Vrije Universiteit Amsterdam

Bachelor Thesis

Designing a Protocol-Agnostic Benchmark for MVEs

Author: Alexandr Costei (2771504)

1st supervisor:	Jesse Donkervliet
daily supervisor:	Jesse Donkervliet
2nd reader:	dr. Daniele Bonetta

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

August 24, 2025

Abstract

Modifiable Virtual Environments (MVEs), such as Minecraft, are widely used for entertainment, education, and research, however their performance level remains difficult to compare across different engines. Existing benchmarks are dependent on specific protocols or implementations, which hinders their ability to provide fair evaluations between these various systems. Therefore, this thesis investigates how to design, implement, and evaluate a protocol-agnostic benchmark that enables reproducible performance analysis of MVEs, with a focus on the voxel engine Luanti.

Luantick is an extension of the Yardstick benchmark, that translates high-level player behaviors into engine-specific actions through lightweight protocol adapters. The system orchestrates automated deployments of Luanti servers, imitates hundreds of concurrent player bots, and collects application-level metrics, like tick duration and player counts, alongside system-level metrics, such as CPU and memory utilization. Results are then processed through an integrated analysis pipeline to generate standardized visualizations and comparisons.

The findings demonstrate that Luantick can reliably execute controlled experiments on Luanti without modifying the engine, while also maintaining protocol independence. Moreover, the benchmark exposes how responsiveness, throughput, and scalability are affected under increasing load, and therefore establishes a basis for comparing MVEs that rely on different network protocols. This work done to achieve this, contributes to a practical and extensible toolchain for future research on performance variability and scalability in MVEs.

Contents

1 Introduction **1**

1.1 Context 2

1.2 Problem Statement 2

1.3 Research Questions 3

1.4 Research Methodology 4

1.5 Thesis Contributions 5

1.6 Plagiarism Declaration 6

2 Background **7**

2.1 Modifiable Virtual Environments (MVEs) 7

2.2 MVE Networking Stack 10

3 Design of Luantick: A Benchmark for Protocol-Agnostic MVE's **13**

3.1 System Requirements 13

3.2 Design Overview 15

3.3 Player Emulation 17

3.4 Server Orchestration 19

3.5 Network and Metrics Analysis 20

4 Implementation Overview **23**

4.1 Luantick Framework Overview 23

4.2 Metrics Collection System Implementation 25

4.3 Rust Bot Implementation and Authentication 26

4.4 Deployment Automation and Integration 27

5 Evaluation **29**

5.1 Main Findings 29

5.2 Experimental Setup 30

5.3 Maximum concurrent players (**MF1**, **MF2**) 32
5.4 Movement pattern & Spawn Radius matters (**MF3**, **MF4**) 34
5.5 Placing blocks costs more than walking (**MF5**) 36

6 Related Work **39**

7 Conclusion **41**
7.1 Answering Research Questions 41
7.2 Limitations and Future Work 42
7.3 Closing Remarks 43

References **45**

1 Introduction

Online gaming has quickly evolved into a significant global industry (1). In 2024, the global games market generated approximately $187.7 billion in revenue, therefore reflecting a 2.1% year-on-year growth (1). This is driven by numerous players coming together and engaging on different platforms such as desktops, mobile devices, and consoles, to play games. However, this is not limited to entertainment as online gaming serves as a platform for social interaction and educational purposes. Specifically, the game “Minecraft" has been utilized in educational contexts, mainly "Math Essentials with Minecraft Education" which provides lessons on geometry and algebra, engaging students through interactive gameplay (2). However, the effectiveness of games mixed with education has been a debatable topic for many years. Therefore, a study was conducted to understand the impact of Minecraft in math class. The findings showed that students enjoyed learning math with Minecraft which simultaneously increased confidence in their math abilities (3).

Furthermore, Modifiable Virtual Environments (MVEs) are a subset of online games that give users the ability to alter the game world by modifying terrain, adding content, or scripting new behaviors in real-time (4). This ability enables MVEs to be powerful tools for education, collaborative design, and simulations. This is evident in a virtual STEM tool called the Urban Ecology Kit which allowed students to modify simulated city environments to explore concepts like population density, green space planning, and wildlife conservation (5). This enabled the students to engage in real-world problem-solving situations, enhancing critical thinking and active learning (5). However, MVEs do present some challenges concerning scalability and performance because as user modifications increase, it becomes harder to maintain the systems performance which also hinders the overall consistency (6).

A method that assesses and compares the performance of systems in order to ensure desired standards are met under controlled conditions is benchmarking (7). But in regard to effective benchmarking with MVEs it is important to address their operational demands which involves simulating realistic player behaviors and measuring a spectrum of performance metrics, from low-level packet transmission timings to high-level tick-time distributions. For example, the state-of-the-art benchmark, Yardstick, simulates lifelike player behavior in representative virtual environments, monitoring various systems, applications, and service-level metrics to analyze the scalability of Minecraft-like network protocol implementations (8). Nevertheless, this comprehensive benchmarking is crucial to ensure that MVEs maintain consistent performance, even as users dynamically alter the environment.

1.1 Context

Building on the introduction above, this thesis focuses on how to study performance in MVEs while worlds are actively being edited. The main challenge with this focus is how to measure responsiveness and capacity in a way that reflects real play and can be compared across different systems. Although current research on the topic exists, researchers either focus on a specific MVE or use a custom set up. Because they use different player scripts, track different metrics, and follow different test steps, their results are hard to generalize and fairly compare (4). I see a missing component in this research which is a neutral, repeatable method that drives every MVE with the same types of player actions and reads out the same core metrics, regardless of how the system is built (9).

A solution to this is the benchmarking approach this thesis employes which (i) uses standardized, realistic workloads to create comparable pressure on each system, (ii) relies on non-intrusive metrics that track user-visible responsiveness and capacity, and (iii) follows an automated, documented process so experiments can be reproduced and extended. Considering the aforementioned solution, the aim of this thesis is to produce fair comparisons and actionable insights about how MVEs behave under change, without privileging any single engine.

1.2 Problem Statement

Despite the increasing use of MVEs, there is no benchmark with the ability to consistently evaluate MVEs with different engine and networking designs, while also keeping workloads, measurements, and reporting comparable. Traditional MVE benchmarks often connect the test scenario with how a given engine communicates. To avoid this, we first decouple the workload from the engine's communication style **(P1)**. To do this, there must be a defined workload model regarding what players do (e.g., move, look, interact, place), how quickly they act (e.g.,pacing and jitter), and where those actions occur, meaning spatial locality and contention zones. By keeping the actions the same across engines, we avoid favoring a specific system, allowing us to examine differences solely from the engines instead of how the test was conducted.

Additionally, we must have a non-intrusive, end-to-end measurement **(P2)**. The benchmark should reveal what users feel and what the system sustains without rewriting the system under the test at hand. Therefore observing signals at the system, application, and service layers, emphasizes action-to-effect latency, steady-state throughput, and coarse resource use on servers and drivers. By collecting these metrics externally where possible, we reduce the risk that measurement itself distorts the behavior we aim to capture.

Lastly, an automated and reproducible deployment is required **(P3)**. Thus, the experiments should start on standard clusters with little manual effort, scale from pilot runs to heavy loads, and emit artifacts that others can replay. This process involves setting up worlds from a clean seed, running scenarios at multiple scales with controlled warm-up and cool-down phases, and exporting traces, configurations, and reports in a standard format. Built-in safeguards, such as health checks and clean teardowns, ensures that a failed run doesn't affect the next one, so results stay comparable.

1.3 Research Questions

Our work explores how the design of network protocols shapes the performance of modifiable virtual environments (MVEs). To keep the study focused and reproducible, we translate the broad problem statement into three concrete research questions. Each question focuses on a different layer of the benchmarking pipeline: concept design, practical implementation, and empirical evaluation, so that their combined answers will form a coherent foundation for future research and engineering. The questions are:

RQ1 *How can we design a network-protocol-agnostic benchmark for MVEs?*

There is no benchmark that can fairly compare different MVEs without assuming the way one engine works. A neutral design will make it easy to extend the benchmark to additional games.

To do this, clear requirements, a defined workload model in high-level player actions, a specific layered architecture with a thin connector boundary, standardized metrics, and simple defaults are all required. This will yield the high-level architecture of the benchmark and the required metrics.

RQ2 *How can we implement this benchmark in practice?*

The only way this design will be impactful and effective is if it becomes a tool people can run, extend, and trust. Without an implementation that handles multiple MVEs, we cannot compare systems in a meaningful way.

To do this, I extend an existing benchmark and add per-game translation libraries that mimic the same high-level actions to each MVE's expected inputs. By combining automation, documentation, and non-intrusive measurement newcomers can run experiments out of the box and contributors can add new engines with minimal changes. Therefore creating a working toolchain that researchers can deploy without modifying the target engine.

RQ3 *How can we evaluate MVEs that rely on different network protocols?*

Fair and repeatable evidence about responsiveness and scale are important to understand which designs perform better and handle more players.

Shared metrics, namely time from action to visible effect, sustained throughput, and high-level resource use, are established to then run controlled tests at multiple load levels with repeated trials. This should establish a procedure for collecting, analyzing, and comparing results across various platforms.

1.4 Research Methodology

To answer these RQs, we will design and implement a benchmark and use it to conduct real-world experiments.

To address **RQ1**, I employed a *AtLarge Design Process* (10), which consists of an iteration through (i) collecting requirements, (ii) sketching a design, (iii) implementation, and (iv) testing and validating. This process will be repeated until the design fits the requirements well. I must combine what MVEs need from a benchmark, mainly workload realism, neutrality to engine/networking choices, and ease of measurement. Additionally, a practical element is needed. For this, one part of the design will generate player actions, another translates them for the system under test, a third records measurements, and the final part turns the data into clear reports.

For **RQ2**, a working prototype called Luantick was built by extending an existing MVE benchmark with a connector that can drive a second family of engines in addition to the original. The prototype maintains the workload and metrics modules with the only aspect changing being the connector. When implementing this, there is an emphasis on clarity, specifically small configuration files, non-intrusive measurement, and a quick start which allows newcomers to run it without struggling.

In regards to **RQ3**, metrics are clearly defined in three ways. First, is the response time, where the action to visible effect. Throughput, is the second, with completed actions per unit time. Third being resource use, such as CPU, memory, and network I/O at a coarse level. Then there are tests that cover different numbers of simulated players and the different ways they interact with the world. Each test is repeated to rule out flukes, and all scripts alongside raw data are shared so anyone can reproduce the results.

1.5 Thesis Contributions

This thesis makes three complementary contributions that advance the state of experimental research on modifiable virtual environments. First, it offers a **design** (Section 3) for a network protocol-agnostic benchmark that isolates the effects of the protocol from the internals of the engine. The design specifies a layered architecture with workload generation, protocol adapter, metric externaliser, and analysis modules that can be redirected to any MVE provided its wire format is known. Design decisions are grounded in the requirements analysis from Chapter 3 and distilled into a set of reusable patterns that researchers can adopt when studying emerging engines or proprietary platforms. To the best of our knowledge, this is the first benchmark blueprint that treats protocol diversity as a first-class concern rather than an afterthought.

Second, the thesis delivers a working **prototype** (Section 4) called Luantick, an extension of Yardstick that supports the Luanti server and its network protocol in addition to the original Minecraft implementation. Luantick implements a pluggable packet codec layer, a language-agnostic control channel, and an automated deployment script suitable for local clusters and public clouds. The entire toolchain is released under an open source license, accompanied by unit tests, continuous integration recipes, and a quick-start guide. By sharing both source code and packaging artefacts, the prototype reduces the barrier for other researchers to reproduce our results, port the benchmark to further engines, or integrate new workload models.

Thirdly, the thesis describes and **evaluates** (Section 5) an extensive experimental campaign using Luantick. The experiments were executed across tens of runs and multiple load levels on shared-cluster hardware. Using the same networking-agnostic workloads and a common measurement pipeline, I quantified user-visible response time, steady-state throughput, and server resource use for multiple MVEs. These repeated trials meant that any differences were a direct result of engine design instead of the test harness. This will then help identify when and why engines separate in performance and when they behave similarly, and those patterns are translated into practical guidance for scaling and design choices.

The GitHub repository link containing the implementation code: `https://github.com/energet1k/luantick`.

1.6 Plagiarism Declaration

I confirm that this thesis is entirely my own work, and has not been submitted elsewhere for evaluation. However, artificial intelligence (AI) played as a supportive role to help me, understand unfamiliar topics, such as how Luanti was built along with their server functionality. This was the prompt I would use "I do not understand X topic, can you explain and give examples to help me understand it."

2

Background

This chapter introduces the key concepts and system models that underpin a protocol-agnostic benchmark for Modifiable Virtual Environments (MVEs). Figure 2.1 presents a high-level deployment model, while Figure 2.2 details the internal flow of a representative MVE server (Minecraft Java Edition).

2.1 Modifiable Virtual Environments (MVEs)

An **MVE** is an interactive, multiuser virtual world in which participants can alter the environment in real time by *(i)* adding/removing terrain, *(ii)* scripting new behaviors or *(iii)* installing custom modifications. Unlike fixed-content games, MVEs must handle *arbitrary user input* while maintaining a consistent shared world state. Representative examples include *Minecraft*, *Luanti* (formerly *Minetest*), and *Roblox*. MVEs are typically deployed as continuous online services with different client devices and variable network conditions. Their main challenge is to remain responsive under simultaneous edits while preserving world consistency and scalability. Every MVE deployment comprises three logical layers (illustrated in Figure 2.1):

1. **Client Layer**: Runs on the player's Device and includes the MVE Client. It renders graphics, captures user input, and transmits input packets labeled "Player actions" toward the Router; it also receives "State updates" from the Router. The client performs local prediction (e.g., movement smoothing), manages an on-disk cache of world assets, and subscribes to the player's area of interest to render only relevant chunks/entities. Key pitfalls include GPU/CPU contention on the device, clock drift affecting latency measurements, and the cost of reconciling predicted state with authoritative updates received from the server.

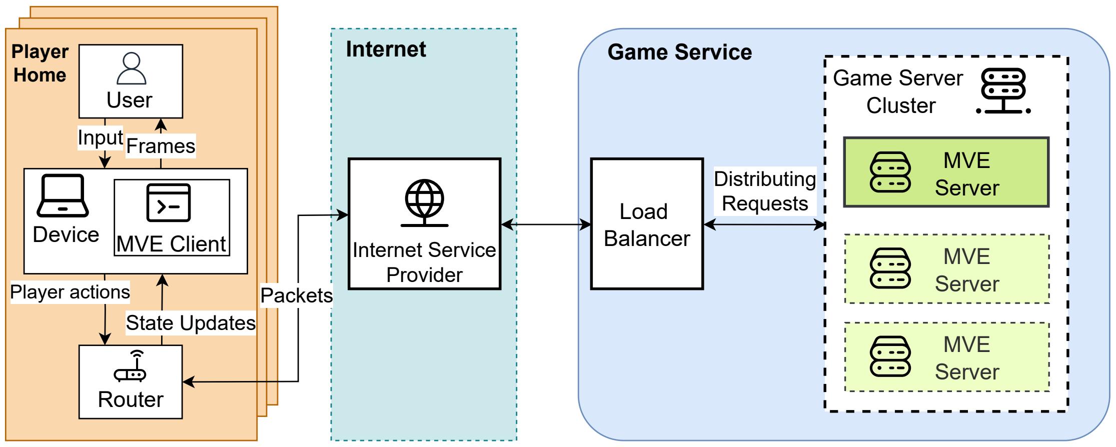

Figure 2.1: System model of an MVE deployment. A player uses their device to run MVE client that connects over custom Protocol through the Internet Service Provided to a load-balanced game-server cluster.

2. **Network Layer**: Spans the Router, the ISP, and the public Internet path to the Game Service. It transports Packets between client and server using either TCP (e.g., PaperMC/Minecraft stacks) or a UDP-based protocol with selective reliability (e.g., Luanti). This layer is where head-of-line blocking (TCP) or per-channel reliability (UDP/RUDP) influences how latency, jitter, reordering, and loss manifest in the observed action-to-effect timing. A protocol-agnostic benchmark must not assume stream (TCP) or datagram (UDP/RUDP) semantics; it should record RTT(duration it takes for packets to go from the client to the MVE server and back), jitter, loss, and reordering while keeping the workload independent of transport details.

3. **Server Layer**: Lives inside the Game Service component and starts at the Load Balancer, which accepts incoming connections and distributes requests to the Game Server Cluster. Each Minecraft Server instance maintains an authoritative world state, processes player actions, enforces game rules, and advances the simulation in a deterministic, tick-based loop. Minecraft-like servers typically target 20 ticks per second (about 50 ms per tick) under healthy load; exceeding the tick budget causes visible slowdowns and degraded responsiveness. The figure highlights that a given player is pinned to exactly one backend instance (the dark-green, solid-border server); the other dashed, pale instances represent additional available servers for other players or future scale-out.

A well-designed benchmark must observe the metrics in all three layers at the same time. At the system level, it watches the CPU and memory (RAM) usage, disk I/O for

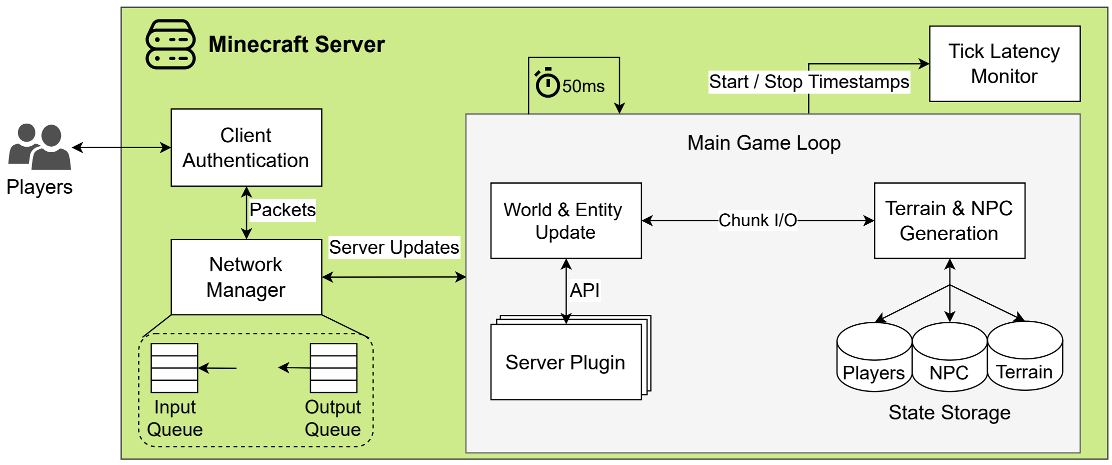

Figure 2.2: Internal flow of the Minecraft server. Packets from clients enter via the Client Authentication check and Network/Session handler, propagate through the Main Game Loop, and are eventually dispatched back to clients.

world saves, and the network throughput rates along with losses and retransmits. At the application level, it times how long each tick takes and how that time is split across major phases like world updates and plugin/mod loads. And at the service level, it checks the big-picture capacity: how many players actions per second the server can complete, how many concurrent clients it can sustain before the latency grows, and how stable the connection stays when the server experiences multiple join/leave or other synchronized edits.

In the reference deployment, the Load Balancer distributes player connections across the Game Server Cluster, while each client maintains a single active session to exactly one server instance (as depicted by the dark-green server). Automated testbeds emulate tens to hundreds of clients and replay realistic workloads (mixed actions, pacing/jitter, spatial locality, contention zones).

It's important to understand that unlike static game worlds, MVEs must deal with many players changing the space at the same time, which demands low-latency synchronization, persistent state storage, and dynamic capacity. Even with a load balancer and a cluster of server instances, players are pinned to a single backend, so hotspots and uneven load remain a challenge. As player counts and edit actions increase, maintaining responsiveness and consistency becomes increasingly difficult unless the system is carefully measured, tuned, and scaled out.

2.2 MVE Networking Stack

The game server is the authoritative simulator for the virtual world. It runs the in-game physics and rules, maintains the global state (players, entities (e.g. NPC), and terrain), and transmits state updates to all connected clients. In Minecraft-like stacks the simulation advances in discrete steps called ticks, where the target is 20 ticks per second (about 50 ms each). When the tick rate drops, players experience stutter and increased game latency. In Figure 2.2, we have a representation of one of the most popular MVE games - Minecraft.

Incoming traffic first hits the *Client Authentication* and *Network Manager*. Together, they accept protocol packets like TCP handshakes (or UDP/RUDP sessions), decrypt, verify identity and permissions, and turn raw player packets into structured player actions. Those actions land in the *Input Queue* so that the simulation can consume them cleanly at the next tick.

The *"Main Game Loop"* is the heart of the server. Each iteration records a start timestamp, processes queued player actions, fires a tick event, and then captures a stop timestamp so the latency monitor can compute per-tick duration and its distribution. If load or slow I/O pushes a tick past 50 ms, the server becomes overloaded, and players see jitter and rollbacks as the authoritative state arrives behind their local predictions. With each tick, the server generates terrain and entities, updates them, and stores them in the *State Storage*. The *World & Entity Update* moves players and NPCs, resolves collisions, applies game rules, and runs plugin/mod hooks. *Terrain and NPC Generation* kicks in as players explore or edit new regions. Once changes are decided, they're saved into to storage.

On the way out, the *Output Queue* prepares updates for each subscribed client it serializes the relevant world differences (placements, breaks, entity movement), and hands them to the networking layer for delivery. This keeps the tick thread free while the network side handles batching, congestion, and retransmissions.

Although many MVEs follow this general shape, real servers differ in ways that can affect performance. Minecraft-like servers, share a similar architecture and rely on Netty for networking, but their internals vary. For example, *PaperMC* introduces asynchronous, worker-threaded chunk loading to reduce tick stalls during exploration (11). Community stacks such as *Bukkit/Spigot/PaperMC* and *Glowstone* expose plugin systems (APIs) that can add functionality and with that, overhead at runtime (12). The world-generation code paths also differ across implementations. For instance, *Glowstone* is a from scratch server whose ecosystem includes plugins specifically to provide vanilla-style world gen on nonofficial servers (13). Under the same workload, these choices generate various performance profiles.

Yardstick's comparative experiments on vanilla, *Spigot*, and *Glowstone* report different scalability limits and show *Glowstone* performing worst among those tested, while vanilla performs the best (8). Measurement at identified instrumentation points allows evaluation of protocol-agnostic performance, independent of the underlying game engine. This thesis underlines the importance of benchmarking: by testing different engines with the same actions and reading out the same cross-layer metrics, we can report real-world capacity and user-visible responsiveness instead of relying on assumptions.

3

Design of Luantick: A Benchmark for Protocol-Agnostic MVE's

This chapter describes the design of Luantick, a benchmark specifically developed to address *"How can we design a network-protocol-agnostic benchmark for MVEs?"* (**RQ1**). First, it outlines and elaborates on the functional and non-functional requirements that guide the development of the benchmark itself, highlighting key design criteria like protocol independence, scalability, and modularity (Section 3.1). Then, it providse a comprehensive design overview, describing the overall process, workloads, and metrics, and how these elements fit together (Section 3.2). Next, outlines player emulation by explaining how realistic play patterns are modeled and arranged (Section 3.3). This is followed by server orchestration which serves as the component that automatically sets up, arranges, and manages game servers across multiple machines, this involves building and installing the Luanti server, applying settings, and starting/stopping servers remotely (Section 3.4). Finally, where tests are monitored and measured for me to collect system metrics (e.g., CPU and memory usage) and application metrics (e.g., server ticks, player count, and server responsiveness) (Section 3.5). This structure ensures that Luantick accurately reflects authentic player interactions and provides protocol-specific performance metrics.

3.1 System Requirements

For this research, functional requirements refer to the features that Luantick must support, whereas non-functional ones are regarding the quality attributes needed to ensure robustness and usability. These requirements are combined leaving the following:

Requirements:

R1 Simulate player behavior across network protocols: To benchmark MVEs with different networking architectures, the system must support simulated clients, that are bots, who communicate over diverse application-level protocols (e.g., TCP-based, UDP-based, or custom hybrids). Many MVEs define their own message formats and state synchronization strategies, so that the bot logic must remain abstracted from protocol details to enable compatibility.

R2 Ease of use: The benchmark should be able to run itself from start to finish. This automation involves setting up a fair test world to start and configure the server, check that it's healthy, run the workload, watch progress, handle crashes or timeouts by retrying and restarting cleanly, alongside shutting down and saving logs, traces, and reports. These steps should also work the same way for different MVEs and world formats, ensuring that runs are consistent and easy to repeat.

R3 Generate diverse and realistic player behavior: To reflect real gameplay and examine how the game engine reacts, bots should be able to act like real players, which involves them walking, idling, navigating toward a fixed coordinate, and block placement. These actions put pressure on different parts of the server, like physics, graphics, and world changes, which then helps in understanding how it performs under different types of load.

R4 Provide protocol-aware logging and metrics collection: To evaluate network behavior and detect bottlenecks, the system must capture relevant protocol-level metrics, including packet counts, latencies, disconnections. In addition, monitoring should be passive and non-intrusive, to avoid significant interference with gameplay or simulation logic.

R5 Low Overhead: In order to test how MVEs behave under realistic pressure, the system must support hundreds of concurrent bots. This requires efficient resource management to avoid introducing overhead when collecting metrics.

R6 Modular and cross-platform core: The benchmark should allow you to swap or add different aspects, like bot behaviors, server connectors, and metrics collectors, without touching the rest. This should run reliably on common operating systems such as Linux, macOS, Windows. Therefore, making it easy to extend to new MVEs and easy to use in different labs and CI setups.

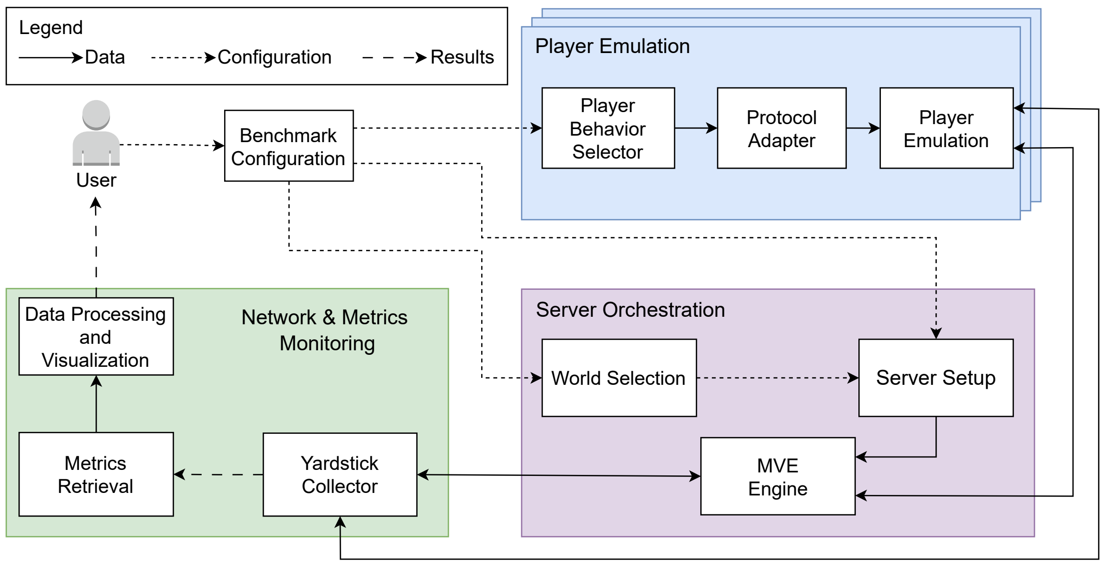

Figure 3.1: Luantick Design Overview

R7 Clarity and Relevancy: The benchmark should select and present results in a way that clearly explains performance. Metrics must be relevant to MVEs and describe system performance and scalability as perceived by real players.

3.2 Design Overview

All benchmarks, including Luantick, define the following three elements (14): The first element is defining a process to conduct the benchmark, while also considering how a single benchmarking experiment should run, and how and what to monitor about the system under test, which is described in Sections 3.4 and 3.5 of this thesis. The second element involves defining the workload given as input to the system under test, outlined in Section 3.3 of this thesis. Third, the metrics to assess the output of the system under test are also defined, further elaborated in Section 3.5 of this thesis. Specifically, Luantick extends these principles to accommodate the protocol diversity essential in MVEs, allowing for a comprehensive performance evaluation across different networking architectures and game engine implementations.

3.2.1 The Benchmarking Process. Luantick's end-to-end workflow directly addresses R1, R2, R4, R6, and R7 mentioned in Section 3.1 of this thesis. It evaluates MVE servers by running experiments that combine protocol-agnostic parameters, such as world configuration, player profiles, pacing, and load levels (**R2**). Each experiment embodies a virtual world and a set of bots defined by these parameters. Since the world and players are configurable, the benchmark supports a wide range of workloads that remain comparable across MVE platforms (**R1**, **R6**). To ensure these runs are repeatable and efficient, the benchmark uses non-intrusive measurement and automated setup and teardown methods (**R4**, **R7**). The workload generated by these bots players is defined by selecting a built-in player behavior pattern, such as random movement, circular navigation, or block placement activities (**R3**). The Protocol Adapter component decouples these behaviors from the network implementations, therefore allowing the same logical behaviors to run across TCP-based, UDP-based, or hybrid protocol architectures (**R1**). The resulting workloads produced from the configuration files, are independent of the specific MVE undergoing the experiment(s) (**R1**). These scenarios are designed to cover a broad set of operating conditions, like varying player densities, complex behavior mixes, and across world interaction patterns (**R3**, **R7**). During each experiment, Luantick monitors the machine that hosts the MVE server alongside the machines that runs the bots. Luantick measures these systems using CPU, memory, network use and application level metrics such as server utilization (**R4**). The monitoring setup is designed to avoid interfering with the system under test while supporting hundreds of concurrent bot connections (**R5**). After each experiment, Luantick stores both the raw data, such as time stamps and resource samples, and the processed data, like tables and plots, which allows me to analyze server performance using metrics that are relevant for MVEs (**R2**, **R7**).

3.2.2 The Benchmark Design. Figure 3.1 showcases the design of Luantick, which consists of three main components that extends *Yardstick*'s architecture to support protocol diversity in MVE systems. The first is the Server Orchestration layer, which manages the MVE service by starting and stopping the server, which exposes the APIs, applies the current configuration, and loads the virtual world for the run. The *World Selection* module enables standardized world configurations across different platforms, while the *Server Setup* automation handles deployment, configuration, health monitoring, and cleanup procedures. This element is seamlessly integrated with the *Yardstick Collector* under the second component, Player Emulation, to provide real-time performance monitoring without manual intervention. Second, *Player Emulation* connects the bots to the system under test via a *Protocol Adapter* component. The adapter enables Luantick's protocol-agnostic behavior (**R1**) by translating player action, such as moving, interacting, and communicating, into the protocol-specific network messages expected by each MVE, including Luanti's UDP protocol and *Minecraft*'s TCP implementation. It is designed to maintain consistent behavioral models across engines while accommodating protocol specifics (**R1**, **R3**). The

Player Behavior Selector offers configurable profiles that reproduce realistic activity, from basic movement patterns to complex ones, for instance building and exploration (**R3**).

Third, the Network & Metrics Monitoring layer observes the bots alongside the server environment to collect and process measurements using a *Yardstick Collector*. The implementation of said *Yardstick Collector* is specific to the MVE under test. Luantick provides collectors for Luanti and offers an extendable interface to add other platforms. These collectors are then implemented as minimal changes to the server source code or as external monitoring agents to keep implementation effort low (**R2**) while still providing player-focused metrics, such as how quickly actions take effect (**R7**). The Collector standardizes the measurements and forwards them to the results database for analysis. Moreover, the modular architecture enables extension to new MVE platforms through pluggable *Protocol Adapters* and collectors (**R6**). The management of the experiment's lifecycle ensures consistent and repeatable benchmarks across different platforms. The collected metrics provide clear insights into the performance characteristics that matter to real players and server operators (**R2**, **R7**).

3.3 Player Emulation

The purpose behind Player Emulation is to reproduce the actions of real players with simulated workloads, this can range from moving and looking around to interacting and placing objects within the virtual environment. The goal of these workloads is not to mimic a single session perfectly, but to generate repeatable behavior that is similar to everyday play activities so that user experience is reflected through meaningful system responses. That said, this section explains the design principles, modeling choices, and validation steps that make these workloads realistic and comparable across different MVE platforms.

Luantick implements workloads for multiple MVE platforms, including Luanti and Minecraft (via *Yardstick*'s existing *PaperMC* support). Each platofrm has its own interaction model, meaning the workloads are not identical. Despite their lack of similarity, they do share a single high-level configuration interface which specifies the number of simulated players alongside their behavioral profiles. Internally, each game uses a separate execution environment, which keeps player behavior faithful to each platform, to ensure that, at the configuration level, users specify "what" happens rather than "how" any engine prefers to receive it (**R1**).

At the design level, these workloads were written in a clear manner. The configuration describes players in terms of high-level actions and the conditions in which those actions occur. Then, the *Protocol Adapter* converts the scenario, of what the simulated players do, how many there are, where they act, how fast they act, and for how long, into engine-specific interactions. This allows me to describe what players do once, and the adapter then handles how each game expects to receive it. This approach keeps the focus on player intent rather than technical protocol details, thereby improving ease of use for benchmark operators (**R2**).

In practice, extending Luantick in a manner to support another MVE means implementing only the adapter, rather than rewriting the behavior model (**R6**). Similarly, most experiments can run again on different platforms by changing the connector but keeping the scenario unchanged, which aligns with the requirement for platform-independent workload generation (**R1**).

Additionally, to enforce the requirement, that the defined behavior remains unchanged across MVE platforms, Luantick defines invariants that every protocol implementation must maintain regarding the order and timing of player actions. The following rules make cross-engine results easier to interpret. First, is order, where it is required for actions to happen in sequence, for instance "move from A to B, then place one block, and look around." After, the connector preserves that sequence within a reasonable time frame.

Second, is where the action (e.g., moving) triggers the corresponding in-game effect as intended. In our example, a movement action must result in visible player displacement, block placement must result in world modification, and camera orientation changes must reflect the specified viewing direction.

Third, is where timing windows arises where it must be specified how often actions should occur, and the connector should meet the given time frame.

However, this raises the question of why there isn't a reliance on one universal script for all games. Such scripts usually blur the differences in how engines handle movement, interaction, and world updates. If the workload is too generic, then it can hinder the exact effects that should be measured. This is why Luantick focuses on what the player is trying to do and allows each game connector to express it in the way the engine expects. This focus keeps things fair, because every engine is asked to do the same set of player actions while allowing each MVE platform to behave in its natural way. Therefore, the benchmark stays neutral to design choices yet still shows the performance impact of those choices(**R1**).

Furthermore, Luantick groups the actions of the players into two families that can be combined to form rich behavior (**R3**). First is movement, which covers waypoint walks, simple random wandering, circular paths, and short idle periods with camera turns. The second is block interaction, where bots create, modify, and remove blocks, from small towers and walls to quick exploratory placements like house. Two parameters control both families. One of which is pacing that sets how often actions happen. The other is locality, which defines where they occur and how long players stay in a region.

Locality matters because real players cluster at points of interest such as spawn areas or build zones. By shaping locality, the benchmark creates natural crowding and contention that stresses scheduling, chunk input and output, and update propagation, revealing how engines behave under spatial pressure (**R3**). At this point, the engine reveals differences regarding how they schedule work and spread world updates.

Each experiment follows a simple phase based schedule with a fixed workload so results are comparable across runs and platforms: warm up to reach steady state, measure to collect data, and cool down for clean tear down. This consistency enables side-by-side comparisons and large automated campaigns (**R2**, **R7**).

In addition, each experiment runs in phases to make results comparable. Movement validation confirms that paths produce the expected position updates, and interaction validation confirms that placements and edits are acknowledged by the server. Moreover, runs that fail these checks are excluded from this thesis, in order to avoid bias results so findings reflect real system behavior rather than experiment faults (**R7**).

3.4 Server Orchestration

As mentioned in Section 3.2.2 of this thesis, the first layer of Luantick is *Server Orchestration* that addresses the complexity of managing different MVE platforms through a protocol-agnostic interface. This component addresses requirements **R1**, **R2**, and **R6** by providing automated deployment, configuration, and experiment management capabilities that effectively work across different MVE implementations.

Regarding protocol-agnostic server management, Luantick implements a unified Server Management API that provides consistent deployment and control interfaces across different MVE platforms. This defines the following four operations: 'deploy()', 'start()', 'stop()' and 'cleanup()'. These operations are implemented by protocol-specific server classes, such as "LuantiServer" for Luanti's UDP-based architecture, with the possibility to extend it to Minecraft servers and other MVE platforms (**R1**, **R6**). Furthermore, the automated deployment pipeline handles the servers lifecycle without manual intervention (**R2**). Building on this API, infrastructure provisioning allocates cluster nodes on demand and prepares the runtime environment by installing required dependencies and configuring networking. The configuration abstraction provides standardized server configuration forms that adapt to different MVE engines. This system keeps the benchmark consistent, by generating protocol specific configuration files, so that the experiments are comparable across platforms. Then, binary distribution builds or retrieves the target MVE server and distributes the same build to all nodes, which ensures consistent versions and stable build settings for the benchmark. Combing these steps, each experiment consist of identical starting points which reduces the chance of configuration drift.

Additionally, world configuration is a crucial part of benchmark reproducibility and cross-platform comparison. Luantick implements standardized world generation mechanisms to create consistent testing environments across different MVE platforms. Therefore, the system is able to support multiple world types, including flat worlds for performance tests, procedurally generated worlds for realistic scenarios, and pre-built worlds for specific benchmark scenarios. Moreover, deterministic world generation using fixed seeds and standardized parameters ensures identical world states across experiment runs. This is important for statistical analysis and performance comparison, because it eliminates world-generation variations as factors in performance measurements. During deployment, monitoring is automatically integrated, so that the system configures protocol specific measurement components, allowing the relevant metrics to be captured without manual intervention.

3.5 Network and Metrics Analysis

As discussed in Section 3.2.2 of this thesis, the Network and Metrics Analysis layer of Luantick implements a comprehensive monitoring and measurement ability (**R4**, **R5**, and **R7**). This element provides protocol metrics collection, low-overhead monitoring infrastructure, and clear presentation of performance results relevant to MVE systems. During experimentation within a protocol-agnostic monitoring framework, Luantick implements a monitoring strategy that captures system level and protocol-specific metrics across different MVE platforms. Each machine runs a local monitoring agent that automatically collects and stores system level metrics, such as CPU utilization, memory consumption, and network throughput. These agents are then organized based on hierarchy with a central monitoring node, called the primary coordinator, that gathers data on all sensor agents across the distributed benchmark to collect their measurements. This method of data collection with these experiments is non-intrusive and it is designed to reduce impact on the system under test (**R5**). For each benchmark run, Luantick buffers samples in memory and writes them to store in efficient formats, which consumes less than 2% of

CPU and memory even under high intensity scenarios with hundreds of concurrent connections. After the collection phase, Luantick performs cross protocol standardization, where protocol-specific measurements are converted into a unified metric, allowing results to be comparable directly across platforms (**R7**). For instance, message transmission is reported as messages per second regardless of the underlying protocol, the connection stability then uses a single success rate definition across all connection management designs. The analysis tool only operates on this normalized dataset, making figures and tables consistent across experiments and systems.

3.5.1 Connection(s) and Network Behavior Analysis. Luantick captures detailed connection lifecycle metrics that provide insights into network behavior and protocol performance characteristics. The system then records connection attempt timestamps, handshake durations, and failure reasons, which allows for a detailed analysis of connection behavior under varying conditions. The session lifecycle monitoring records player session duration, disconnection patterns, and reconnection behavior. It maintains detailed logs of session events, including voluntary exits, network timeouts, and connection failures detected by the protocol layer. These measurements are important because they provide insight into the stability characteristics of different MVE protocols and their behavior under stress conditions, so we can identify patterns that create instability and compare how different systems recover from such faults (**R4**, **R7**). Next, is the behavioral impact assessment which links performance measurements to specific player behavior patterns (**R3**, **R7**) by measuring how simple movement, mixed interaction, and intensive building affect server responsiveness, network use, and resource consumption. Comparing these elements under the same conditions offers insight as to which behaviors trigger contention, which settings remain stable, and where configuration changes yield the greatest improvements. These results are beneficial to developers and benchmark operators as they can now adjust the performance for the behavior mixes they expect in production (**R3**, **R7**). Building on the session and behavior analyses, Luantick derives service level metrics from the system level and protocol-aware measurements it collects. From the message transmission data, it calculates frequency distributions, average message sizes, transmission rates for each message, and statistical summaries. The message sizes reflect application level payloads rather than full network packet sizes, keeping the results independent yet suitable for cross protocol comparison. From server update data, Luantick also estimates the update measure of the game loop to show how efficiently resources are used for a given workload. Together, these metrics provide clear indicators of server performance that align with what players perceive in practice (**R7**). After each experiment, Luantick automatically runs a data processing pipeline that takes the collected metrics and logs, checks their integrity, aligns timestamps, and collected measurements into consistent time windows. Then, it calculates summary statistics later analysis, with the raw traces, supporting detailed inspection and re-analysis, and the processed outputs, providing summaries for quick assessment and comparison, saved (**R2**, **R7**). Building on the mentioned processed dataset, Luantick also provides interactive charts and comparison views to examine the results. Such visualizations help reveal trends across load levels, identify outliers, and compare experimental conditions side by side. After, Luantick generates a complete results report that documents the setup and configuration, defines the metrics, presents statistical summaries and figures, and records versions, seeds, and any deviations from the plan. The report gives operators and researchers a single product that can be reviewed, shared, and archived, which supports fair comparison across MVEs and makes follow up experiments easier to plan and execute.

4

Implementation Overview

This chapter describes the implementation of Luantick, which is a benchmark prototype developed to answer How can we implement this benchmark in practice? (**RQ2**). The goal is to translate the protocol-agnostic design from Chapter 3 into a working system that can be executed, extended, and trusted for experiments.

First, there is an overview of the used implementation approach, explaining how the existing Yardstick framework was extended and adapted to support the Luanti engine (Section 4.1). Next, is the data analysis pipeline where collected metrics are processed through Yardstick's analysis scripts to generate standardized results and visualizations (Section 4.2). This is followed by the bot implementation, where Rust-based clients are built to mimic realistic player actions and interact with the Luanti server through its native protocol (Section 4.3). Then, deployment automation is introduced, where the Luantick Collector is enabled through server configuration templates and operates as an in-game component that records application-level metrics such as tick duration and player activity, exporting them in a structured format for analysis (Section 4.4).

4.1 Luantick Framework Overview

The primary goal of the implementation is to turn the design into an extensible toolchain that remains protocol-agnostic while still handling the realities of benchmarking MVEs. To do that, the framework combines server orchestration, player workloads, and metrics collection. It reuses Yardstick's core ideas for experiment definition and analysis, but extends the implementation of a stack that fits the Luanti's C++, Lua, and UDP foundations.

The implementation follows a modular approach so that each concern evolves without affecting the rest of the benchmark, which is achieved by a high-level architecture con-

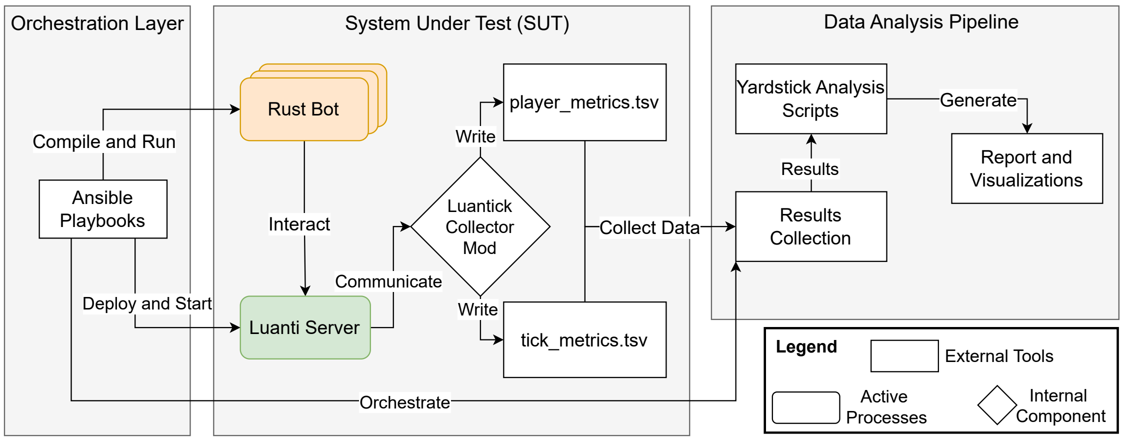

Figure 4.1: Luantick Implementation Overview

sisting of three main elements. First, is an Ansible-based orchestrator that manages the experiment lifecycle end to end. Secondly, a Luanti server instrumented with a lightweight Lua collector. Third being a fleet of Rust based bots that generate player workloads. This means that layers communicate together through explicit files, networks, or process boundaries, which maintains clear and repeatable responsibilities.

These implementation aspects, shown in Figure 4.1 ensures stable relationships. Rectangles with sharp edges denote external tools and systems, such as Ansible playbooks, TSV files, and result collection. Rounded rectangles mark active processes and services like the Luanti server and the Rust bots. Diamond shapes indicate internal modules embedded within larger systems, for example, the Luantick collector mod. The technological choices reflect trade-offs between portability, overhead, and development speed. For instance, Ansible is used because of its agentless model, which simplifies setup in shared research clusters. Declarative YAML playbooks give a versioned and readable description of provisioning, configuration, execution, and teardown. Therefore, supporting automated and reproducible deployment while avoiding background operations.

For metrics collection, Lua is used because Luanti already exposes Lua hooks as APIs, where a collector mod reads timing and player state via

`minetest.register_globalstep()` and `minetest.get_connected_players()`. This direct access removes the need for sidecars or polling exporters to get data. The result is precise tick and event timing with negligible added latency. That said, we keep the collector minimal to respect non-intrusive monitoring goals.

For workload generation we lean on Rust and the texmodbot ecosystem (15). Rust offers predictable performance and memory safety, plus asynchronous concurrency that scales to hundreds of simulated players. Here, helper libraries such as `mt_auth`, `mt_net`, and `mt_rudp` summarize protocol details so the high-level behavior logic can focus on realistic scenarios rather than packet details (16).

The monitored metrics are appended into TSV files. The reason why TSV was chosen is because the format is simple and cheap to parse, which works well with the existing Yardstick scripts. For example, TSV handles well partial writes and remains easy to inspect during debugging. By contrast, JSON made debugging slower when something went wrong.

Together, these choices implement the mentioned design requirements from Section 3 into practice. Specifically, modularity for easy extension (**R6**), low overhead so that the benchmark does not disturb the system under test (**R5**), clear and relevant metrics for interpretation (**R7**), and protocol neutrality from client to wire to server (**R1**).

4.2 Metrics Collection System Implementation

The server side metrics collection system is one of the most important parts of the framework, because instead of relying solely on the Java plugin model, a Lua native Luantick implementation is added. The goal behind this is to observe timing and activity patterns such as tick duration, player actions, and interactions with the server without altering the engine execution path or introducing blocking I/O. For example, timestamps are read, and compact records are then written, but the tick thread never waits on the file system.

To implement this collection system, a Lua mod, called the Luantick Collector Mod, integrates with the tick scheduler. The orchestration layer deploys the mod and the server entirely, in a manner where no manual server edits are required to make this mod work. Here, the collector attaches a single global step callback that performs minimal arithmetic, where each tick is appended to the metrics file. As a result, measurement remains predictable even when the world grows or many players connect.

The core loop presented in Listing 4.1 records three things. First, `minetest.get_us_time()` is used to record in append mode per tick timing. This is done by each tick emitting one formatted line for each to join, leave, or interact while also leaving visible small latency spikes. However, if a process unexpectedly stops, the written lines are still valid, therefore yielding durable logs with low overhead and simple inspection. If a mod fails, the test continues to run as if the collector were absent. Second, player join and leave events, which are registered from the engine callbacks, which then provide a timeline allowing the correlation of player connections with tick variance. Third, block interactions such as place and dig are recorded to later support action to effect analysis once client side timestamps are available.

```lua
-- Define file paths
    local metrics_file = minetest.get_worldpath() .. "/mod_storage/tick_metrics.tsv"
    local player_file = minetest.get_worldpath() .. "/mod_storage/player_metrics.tsv"

    -- Record tick performance
    minetest.register_globalstep(function(dtime)
        local now = minetest.get_us_time()
        local duration_us = now - last_time
        last_time = now
        tick_count = tick_count + 1

        local timestamp_s = now / 1e6  -- seconds since epoch
        local duration_ms = duration_us / 1000  -- milliseconds
        local players_online = #minetest.get_connected_players()

        local file = io.open(metrics_file, "a")
        if file then
            file:write(string.format("%.3f\t%.3f\t%d\t%d\n",
                timestamp_s, duration_ms, tick_count, players_online))
            file:close()
        end

        if duration_ms > 100 then
            minetest.log("warning", string.format("YARDSTICK: High tick duration: %.2fms (players: %d)",
                duration_ms, players_online))
        end
    end)
```

Listing 4.1: Example of the function that registers server ticks

4.3 Rust Bot Implementation and Authentication

Luantick runs the bots that drive the workload, which sign on to the server using the expected handshake, where the client sends the server a message for the server to respond to. For Luanti, there is an added adapter alongside the existing tools for Minecraft rather than replacing them, so that both engines can run the same scenarios. However, some network transports do not guarantee delivery or ordering. The bot implementation handles re-sends and messages that arrive out of order, but this is done without affecting the schedule of actions defined by the workload. For example, it will resend when required and reorder only as needed while keeping the original timing for move, interact, and place actions.

The mentioned authentication aspect is derived from the Secure Remote Password handshake. Here, the `mt_auth` library involves verifier computation and proof exchange so that higher layers only react to success or retry states. This flow is straightforward, as a bot connects and sends an initial message with its username. The server, as aforementioned, then replies with SRP parameters such as the server public key. After, the bot combines the parameters with the user password to compute a proof of knowledge and sends that proof back. Once the server validates the proof, access is then granted and normal game play begins.

To keep the bot connection reliable and authentic, the library `mt_auth` changes in a manner that allows bots to auto-register in the server. This is crucial as Luanti serves require users to log in alongside having unique usernames, in order to play, creating challenges when implementing bots. That said, bots now have distinct usernames which ensure that they succesfully connect to server.

The behavioral realism system provides two bot classes, being Walkbot and Blockbot, that target different subsystems. On one hand, Walkbot focuses on movement, while offering four movement archetypes. First, the static mode provides a baseline with minimal network interference. Second, the random mode involves simulating casual exploration by randomly choosing directions and speeds. Third, the circular mode produces a predictable yet consistent load by moving around a fixed center. Lastly, the follow mode drives bots towards specific coordinates by using trigonometric functions to calculate the direction of movement. On the other hand, Blockbot focuses on world mutation while emphasizing mutation paths through structured build patterns such as tower, wall, platform, and spiral.

Finally, position management maintains the visibility and use of bots. To avoid falling through terrain or getting stuck, the bots fix their vertical coordinates at Y equal to 8.5. This places them above typical ground but below common build heights, which makes local observation easy while keeping positions consistent for measurement.

4.4 Deployment Automation and Integration

The deployment system of the Luanti Server and Rust Bots handles machines, preserves reproducibility, and reports failures clearly. With this, each trial is defined by declarative playbooks that record the exact scripts and settings, so that one can trace any result back to its inputs and replicate runs.

Additionally, orchestration also uses playbooks for each phase of the experiment lifecycle, ranging from prepare and launch, to measure and teardown. Such a modular layout allows us to test or debug one phase without touching the others, therefore keeping the responsibilities separate. Another function of orchestration is that it handles the realities of building Luanti from source on mixed cluster nodes, managing libraries, and keeping build environments consistent. For example, `luanti_deploy.yml` provisions the server by installing dependencies, compiling source, and generating configuration files.

Since building Luanti from source code requires multiple dependencies and is time consuming, the aforementioned playbook file that deploys said Luanti server copies the executable file from the head node where it was complied in advance, which is evident in the Figure Screenshot. It begins by preparing the environment with tool chains and libraries. Next, it gathers and compiles the source to validate that the binary produced exists and runs. If any step were to fail, the system reports the issue early, so nothing is launched against a broken server.

Furthermore, when configuring the server the implementation uses *Jinja2*[1] templates to produce server configures with experiment specific parameters. Here, the user sets network options, world parameters, security policies, and mod-loading instructions, making the Luantick Collector present and active. The template approach provides consistency across runs while still allowing controlled variation for different scenarios.

Bot deployment is managed by `rust_walkaround_deploy.yml`, which sets up the Rust toolchain and compiles the bots across all nodes. The *texmodbot* stack expects a nightly rust toolchain[2], where the playbook then ensures that the correct compiler is available everywhere. The source is then synchronized, which resolves dependencies, and builds binaries optimized for the target hardware. To keep builds reliable, cargo fetch steps use throttling and controlled serialization keeping logs readable while reducing failures. This eliminates deployment issues when bots attempt to deploy on multiple machines.

Finally, results collection gathers metrics and logs into a time-stamped directory. Simple structural checks look for the presence of tick, player, and interaction files and for the main server log. As a result, missing artifacts are immediately noticed, allowing for a correct analysis.

[1]`https://jinja.palletsprojects.com/en/stable`
[2]`https://rust-lang.github.io/rustup/concepts/channels.html`

5

Evaluation

This chapter presents the evaluation of Luantick, helping to answer the third research question *"How can we evaluate MVEs that rely on different network protocols?"* (**RQ3**). The goal is to apply the implemented benchmark to structured experiments that reveal protocol-specific performance, scalability limits, and resource utilization patterns across different engines.

First, the main findings are introduced, highlighting protocol-level performance differences, resource utilization trends, and the scalability boundaries of the tested environments (Section 5.1). Next, the experimental setup is outlined, describing the controlled environment where Luantick was deployed and the variables selected for systematic testing (Section 5.2). This is followed by the presentations of the main findings **(MF)** from their introduction in Section 5.1. As **MF1** and **MF2** address the impact of protocol differences on responsiveness and throughput, they become interconnected and are discussed together (Section 5.3). **MF3** and **MF4**, also build on each other, since they expand the analysis to cover scalability under increasing workloads and resource utilization trends (Section 5.4). Finally, **MF5** describes a distinct aspect of the evaluation (Section 5.5).

5.1 Main Findings

We derive the following main findings from the experiments summarized in Table 5.1:

MF1 On the `minetest` game mode, an official default Luanti server **hits a retention ceiling around ≈180 concurrent players**. Beyond that, performance is limited by the game engine, which points to a connection/session management limit rather than a resource limit, further elaborated on in Section 5.3.

	Parameters				
Focus	Players	Behavior	Game Mod	Environment	Dur. [m]
System scalability (walk)	5–300	circular	`minetest`	DAS-5	4
SC: Movement pattern (walk)	5–500	random	`minetest`	DAS-5	4
Spawn radius (walk)	5, 100, 150, 200	random (r=60)	`minetest`	DAS-5	4
Spawn radius (walk)	100, 200	random (r=120)	`minetest`	DAS-5	4
Build pattern (block)	5–300	tower (r=0)	`minetest`	DAS-5	4
Build pattern (block)	100, 200	tower (r=120)	`minetest`	DAS-5	4
Pattern variation (block)	100	wall (r=0)	`minetest`	DAS-5	4
Game mode (walk)	5–150	random	`extra_ordinance`	DAS-5	4

Table 5.1: Benchmark families grouped by focus, parameters, and environment. Spawn radius is shown in the *Behavior* column as $r = \{0, 60, 120\}$.

MF2 **Game mode selection materially changes capacity**. The `minetest` mode consistently supports more players at target TPS than the community `extra_ordinance` mode under similar loads.

MF3 **Movement pattern matters.** At the same concurrency, Walkbots with 'random' workload reduce TPS slightly and increase tick-time variance compared to circular Walkbots. These circular movement represents an "idealized" steady load, while random movement produces short micro bursts, which are described in Section 5.4.

MF4 **Spawn radius changes I/O and performance**. Distributing players over larger areas (radius 60 or 120) impacts workloads in different ways. For Random Walkbots, this results in a minor performance penalty, with slightly reduced TPS, longer tick durations, and increased network throughput, while CPU and memory usage remain low. On the opposite end, Blockbot runs benefit from this distribution, achieving higher TPS alongside reduced CPU consumption, as discussed in Section 5.4.

MF5 **Placing blocks costs more than walking.** For comparable amount of connected players, Blockbots consume more memory than Walkbots while CPU and average TPS stay similar, outlined in Section 5.5.

5.2 Experimental Setup

All experiments run on the DAS-5 distributed medium-sized supercomputer, which is part of the VU cluster (17). The nodes of DAS-5 are equipped with two 8-core Intel E5-2630v3 CPU @2.4 GHz with 64 GB RAM that give a lot of computing power for the experiments. Unless stated otherwise, one node hosts the Luanti server and the remaining nodes host the bot workload.

As mentioned in Chapter 4, the experiments ran on Luantick, which is an extension of *Yardstick* for the MVE called **Luanti** [1]. The benchmark reserves one or more of the 68 DAS-5 nodes and deploys server and workload processes using custom Ansible playbooks that (i) copy shared libraries from the head node to the server node, (ii) copy the Luanti server binary and its data directories, (iii) install a nightly Rust toolchain on each worker node, and (iv) copy the bot workloads and metric collectors. Therefore, there is no need for custom VM (Virtual Machine) images since everything runs on the host Linux installation.

First, the *system scalability* test was conducted to bracket the feasible concurrency for the server on `minetest`. In practice, this establishes the safe operating window that subsequent experiments should target. Next, the *movement pattern* is varied because path diversity directly affects the per-tick workload mix which includes entity updates, collision checks, and visibility calculations. After, the *spawn radius* is adjusted to reflect how public servers often concentrate players in hubs influencing spatial locality and streaming pressure. Finally, *build patterns* like tower vs. wall are compared, because different world mutation pathways stress the engine instead of pure movement.

System metrics such as CPU, memory, and network and disk throughput are collected with Telegraf[2] on all active nodes. For instance, CPU and memory come from `cpu` and `mem` measurements, while per-interface network counters (`net`) and per-device disk counters (`diskio`) are separated into rates. Application metrics are then exported by a Lua collector, including per-tick timing, player joins/leaves, and block interactions. All application streams are append-only TSV files, with one line per event, making partial runs recoverable in case the metric collector stops working or the server shuts down unexpectedly. In the analysis step, each benchmark run gets its own ID, so then (i) timestamps are normalized, (ii) concurrency whether peak or unique are obtained, and (iii) join network/disk summaries are linked back to the run ID. This benchmark is designed in a manner where the entire pipeline is reproducible, inputs are present in the per-run directory and the notebooks/scripts write explicit CSV outputs alongside figures.

All experiments were carried out using the implemented protocol adapters for Luanti and unless otherwise noted, follow this environment and setup.

[1]`https://www.luanti.org`
[2]`https://github.com/influxdata/telegraf`

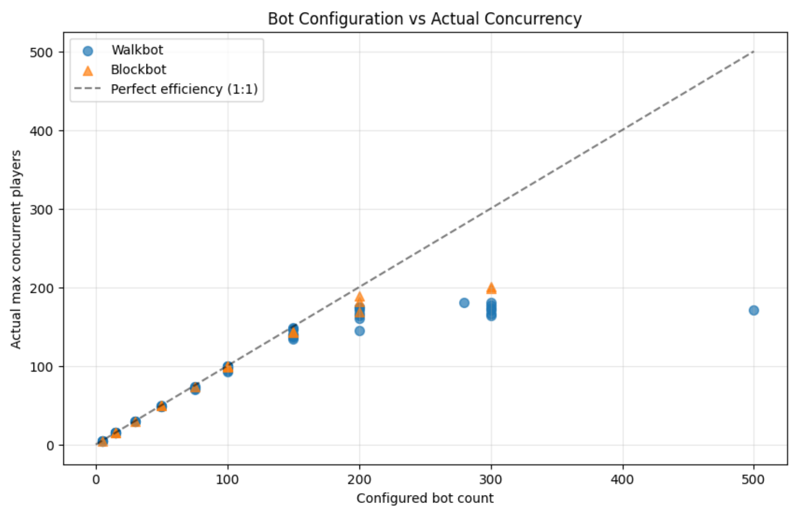

Figure 5.1: Maximum concurrent players against the desired number of emulated players

5.3 Maximum concurrent players (MF1, MF2)

Prior works show scalability limits ranging from a few dozen to a few hundred concurrent players, depending on the Minecraft-like game and workload used. For example, the serverless prototype *Servo*, reached around 150 players during its scaling evaluation, whereas Luanti achieved 180 players under similar conditions as seen in Figure 5.1. The theoretical limit for Minetest, and thus Luanti, is 65,535 players via `max_users` setting in `minetest.conf`. That said, the practical and playable maximum depends on world content, mods, and server configuration, with community reports of lag above $\approx$ 200 players being common.

During the runs on `minetest`, the server maintains 20 TPS, with the general range being from 19.4 to 19.9. The CPU and memory are below saturation, yet concurrency stays around 180. This maximum of 180 concurrent players is concrete because at the 175-180 mark, if new players join the server, that results in almost immediate disconnection of existing peers, which looks like a retention ceiling rather than a throughput collapse.

During multi-bot runs on the community `extra_ordinance` mode, the server logs emit the warning shown also in Figure 5.3:

```
Third person will break aiming and many other things, turn back while you
still can. This is an engine limitation
```

. As demonstrated in Figure 5.2, the *Minetest* game mode remains stable close to 19.9 TPS even as the number of players increases. However, *Extra Ordinance* shows a decline,

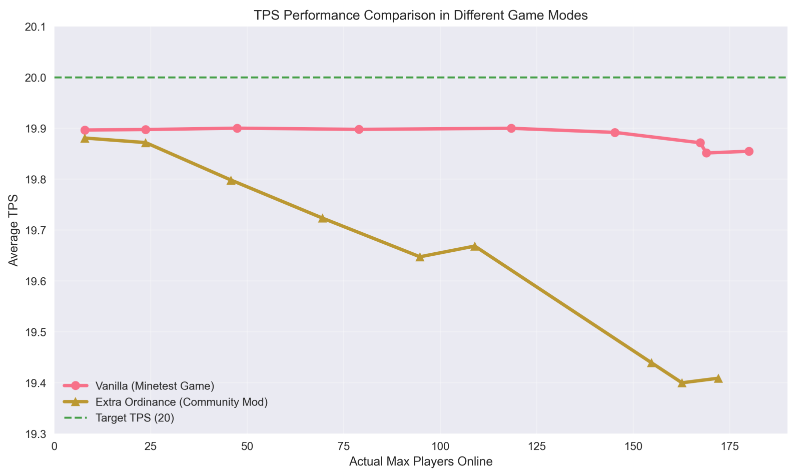

Figure 5.2: Average Ticks Per Second (TPS) variation between the 2 tested game modes: Minetest and Extra Ordinance

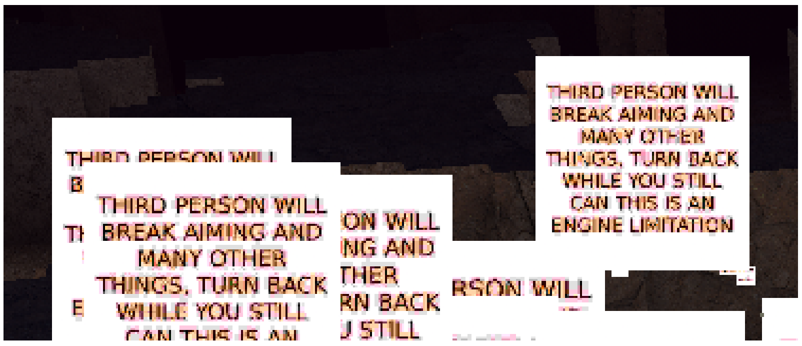
THIRD PERSON WILL
BREAK AIMING AND
MANY OTHER
THINGS, TURN BACK
WHILE YOU STILL
CAN THIS IS AN
ENGINE LIMITATION

Figure 5.3: Descriptive message shown in the game mode 'Extra Ordinance' when more than 2 players join the server.

dropping below 19.5 TPS as the player number approaches 150 to 175 which is beyond the recommended number of active players according to the server warnings, meaning that while the added complexity of the community mod reduces scalability and introduces overhead that directly impacts tick stability.

Luanti uses a per-iteration packet budget which is divided fairly across active peers. In simplified form:

```
peer_quota = max(1, total_iteration_packets / active_peers)
```

In our configuration, `max_packets_per_iteration = 2048` (twice the default). At ≈ 171 peers this yields roughly 12–13 packets per peer per iteration, matching warnings such as:

```
WARNING[ConnectionSend]: Packet quota used up for peer_id=64003, was 13 pkts
```

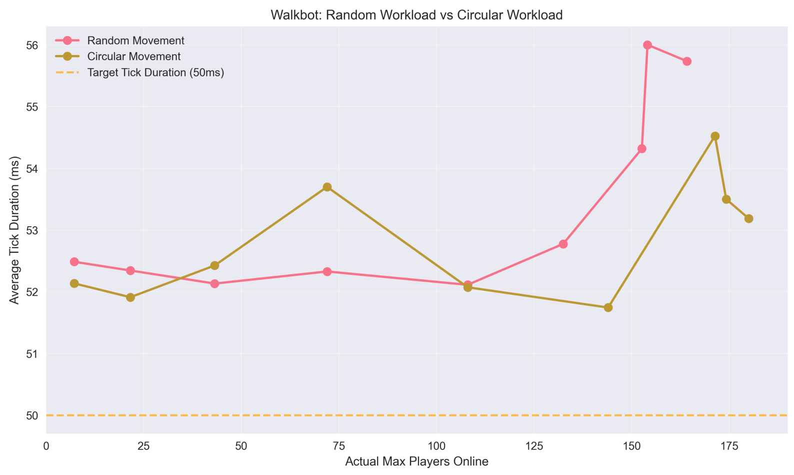

Figure 5.4: Comparison of average tick duration (ms) between the 'random' workload and 'circular' workload

Here, the limiter is not the overall packet capacity but instead the portion of the fixed budget allocated to each player. As the active set grows, each client receives fewer packets per tick, therefore balancing the send loop but also makes marginal peers more fragile during player action overload.

It is likely possible to push beyond 180 with tailored network settings or multi-instance designs, but the evidence indicates a connection/session management ceiling rather than a tick-rate or CPU/memory limit. This means that the main bottleneck at that point is who gets how many packets per iteration rather than whether the main loop can keep 20 TPS.

5.4 Movement pattern & Spawn Radius matters (MF3, MF4)

Random Walkbots introduce uneven bursts in movement and entity updates, such as many bots change direction at once, therefore colliding with world features, or cluster temporarily. On the other hand, circular Walkbots, produce a steady, predictable stream of updates at constant speed and curvature. This is evident in the Figure 5.4, that show slightly higher tick average and visibly larger variance for random movement at the same connected players. The CPU deltas are small, which suggests the main effect is scheduling and short-term contention rather than sustained compute load as shown in Figure 5.5

Figure 5.4 shows that at lower player counts, both workloads stay fairly close, but as concurrency increases the differences become clear. Random movement, produces higher variance and sharp spikes, especially over 150 players, because bots frequently change

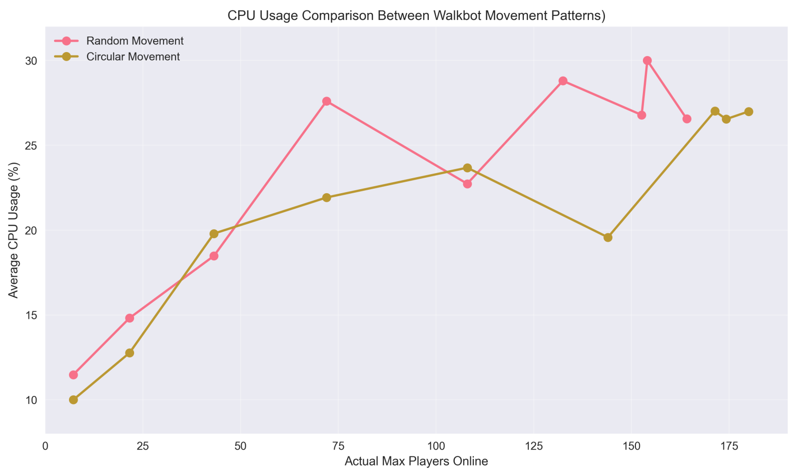

Figure 5.5: Comparison of CPU usage between the 'random' and 'circular' workloads of Walkbot. Small deltas suggest scheduling/contention effects.

direction at the same time. On the other hand, circular movement creates a smoother and more predictable load, although it still rises above the 50 millisecond threshold at higher scales. This means that the type of workload matters, because despite the same number of players, less predictable behavior generates more pressure on the server which reduces stability

Spawn placement changes spatial locality. With radius 0, all players spawn at the same location, therefore allowing the server to reuse nearby mapblocks and neighbor lists while keeping streaming paths short. Then, with a radius of 60 or 120, players spread out and the server interacts with more distinct regions per minute, which increases world streaming and event fan-out. The experimental data shows that (i) a modest TPS reduction and higher tick variance, and (ii) increase network and disk activity which is most evident with Blockbots as they continuously modify the virtual world. However, with Walkbots the effect is present but smaller, meaning that concentrated spawns are more efficient, whereas large spawn radii provide greater spacial fairness for slightly higher disk I/O usage, and reduced tick stability.

Figure 5.6 shows how changing the spawn radius impacts disk I/O and performance for Walkbots and Blockbots. When all players are concentrated in a hotspot, disk usage is lower and more predictable because the server repeatedly accesses the same nearby map blocks. However, as the spawn radius increases to 60 and then 120, players are spread across regions, which increases streaming and I/O demand. This is more evident with Blockbots, where larger radii push disk activity consistently higher, since building actions continuously modify new areas of the world. Walkbots, on the other hand, also show an

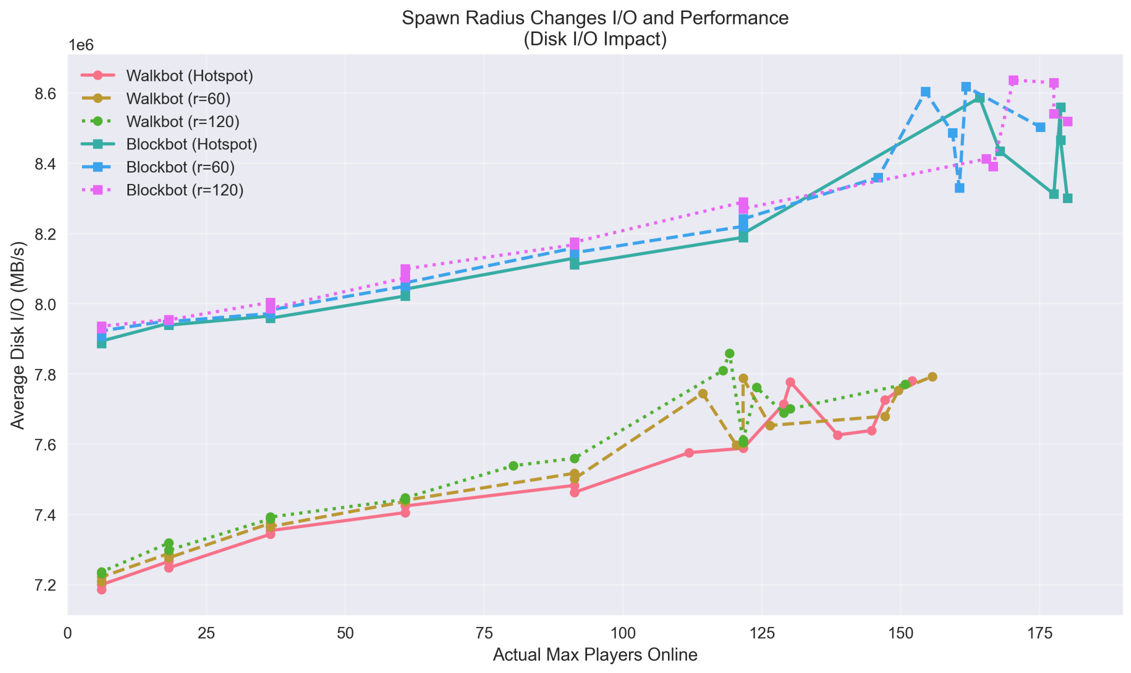

Figure 5.6: Comparison of Disk I/O usage between the Walkbots and the Blockbots

increase, but the effect is smaller because simple movement generates fewer world changes. Therefore, proving that spatial locality does matters since concentrated spawns are more efficient for the server, while spreading players across the map increases I/O load and reduces stability.

5.5 Placing blocks costs more than walking (MF5)

The two subplots in Figure 5.7 compare CPU and memory usage for Walkbots and Blockbots. On the left, CPU usage gradually risies with player counts for both workloads, but the difference between walking and block placement is not significant. Yet, on the right, the memory usage it is evident that Blockbots consistently consume more memory than Walkbots, often staying close to 90 percent, whereas Walkbots remain around 65 to 70%. This proves that placing and modifying blocks is significantly more expensive than just moving, because each action requires the server to track and update additional state. These results stay consistent across tower workload runs and remain stable even when the spawn radius is changed, confirming that world mutations have a higher resource cost than pure movement.

World mutations trigger allocation and retention in several places: (i) mapblock caches,

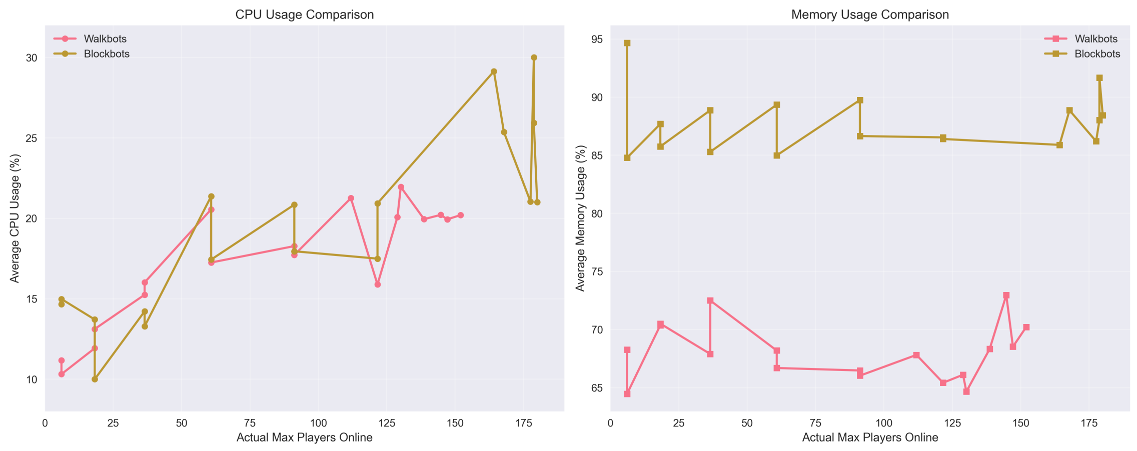

Figure 5.7: CPU and Memory comparison between Walkbots and Blockbots

(ii)) metadata for active blocks, and (iii) server-side data structures that queue and propagate changes to clients. For example, dig and place events increase the amount of modified regions alongside expanding the set of loaded mapblock, even if the tick loop itself stays short. In practice, Lua-side instrumentation and engine behavior allow these changes to be committed without stretching the main step beyond target, so that TPS stays flat while the memory footprint rises.

6

Related Work

Research on benchmarking and scalability of MVEs has gained significant attention across the computer science field especially concerning distributed systems and gaming communities. MVEs enable interactive yet consistent virtual worlds but also present significant challenges when scaling beyond a few hundred concurrent players. Previous research in the field explored multiple approaches to improve performance from server-side optimizations and distributed architectures to benchmarking methodologies. This section is a literary review of related works across four elements (i) scalability techniques for MVEs, (ii) benchmarking frameworks and methodologies, (iii) alternative architectures for game engines, and (iv) open-source engines and protocols relevant to Luanti.

To begin with, research on the scalability of online games has consistently focused on overcoming bottlenecks in simulation and networking. Earlier studies showed that limitations of single-threaded event loops in commercial servers, where CPU and network saturation constrains player concurrency. Expanding this, more recent work at the Vrije Universiteit (VU) examined new approaches to address this limitation. Last year, a lock-step simulation model for MVEs was introduced by Kamberi, which enforces deterministic progression to ensure fairness across clients at scale (6). This method maintains consistency while reducing responsiveness in highly interactive environments. Similarily, Servo is a serverless approach to scaling MVEs, that was proposed in 2023 by Donkervliet et al. (4). Their design uses function-as-a-service platforms to spread simulation tasks across multiple invocations, decreasing idle capacity while increasing elasticity. However, despite these advantages, the approach requires some alterations to the game engine, which then complicates its adoption in practice.

Second, benchmarking has become an important method to evaluate the performance of MVEs under controlled conditions. Gray's Benchmark Handbook (14) established systematic practices for benchmarking in databases and transactional systems, which later informed approaches in gaming contexts. For MVEs, van der Sar et al. introduced Yardstick in 2019, which is the first benchmark specifically designed for Minecraft-like services (8). Yardstick generates workloads based on real player behavior, derives service-level indicators, and measures metrics on a system and application level. The results of their study showed that Minecraft servers are limited by weak parallelization, therefore restricting their ability to scale. Building onto this, Eickhoff et al. (2023) designed a benchmark called Meterstick to capture variability across cloud-hosted and self-hosted MVE deployments (18). Their findings demonstrated that variability plays a significant role in limiting scalability, with latency spikes exceeding acceptable thresholds. This emphasizes that benchmarking is necessary for identifying hidden performance issues that undermine responsiveness.

Third, alternative designs for interactive simulations have been widely explored in recent research. The AtLarge Vision on distributed ecosystems outlines general principles for large-scale system design, focusing on modularity, openness, and reproducibility (10). These principles have influenced benchmarking by encouraging the development of reproducible pipelines. However, these designs are dependent on adaptive consistency to achieve scalability without sacrificing responsiveness. For example, dyconit-based systems adjust consistency bounds according to player interactions which reduces unnecessary synchronization while maintaining interactivity. These studies show the architectural strategies proposed for MVEs, ranging from adaptive middleware to serverless execution models.

Lastly, open-source game engines and their protocols provide an essential foundation for experimentation and benchmarking. Luanti is a voxel-based engine written in C++ with Lua scripting support, unlike Minecraft which operates on a proprietary Java-based server. Luanti offers openly documented networking protocols (9) that allows direct benchmarking without the need for reverse engineering. Community-driven implementations, such as Spigot and Glowstone (12), (13), extend the variety of server designs developed to address the limitations of the Minecraft server, which serves as practical alternatives for players and as research platforms for scalability studies. These opportunities are extended by recent research from open-source communities, which is evident in rust-based libraries for Minetest (16). These libraries introduce modern performance enhancements, whereas lightweight tools, like texmodbot (15) highlight the interest in automated interaction with MVEs. When combined, these ecosystems provide the adaptability and transparency required to advance benchmarking research, supporting integration across diverse scenarios.

7

Conclusion

In this thesis I set out to design, implement, and evaluate a benchmark that can fairly compare MVEs across different network protocols. My inspiration derived from existing state-of-the-art benchmark, *Yardstick*, providing strong insights into Minecraft-like servers, but their tight coupling to a single protocol limited their generality. With Luantick, I explored whether a protocol-agnostic approach could be made practical and whether such a system could reveal meaningful differences in performance across engines.

7.1 Answering Research Questions

In this section, we will address the main Research Questions introduced in Section 1.3.

For **RQ1** regarding the design, Chapter 3 highlighted the essential requirements for building a protocol agnostic benchmark for MVEs. I found that the main challenge of the design is is by separating what players do from how each engine expects those actions on the wire. To overcome this, I introduced a layered architecture with workloads, protocol adapters, and metrics collection. This ensures that the same high-level behaviors, such as walking, idling, and building are consistently applied, while each engine's connector handles the details. In my opinion, this design shows that neutrality can be achieved without losing realism, which is done by centralizing requirements such as automation, non-intrusive measurement, and clear metrics. By doing this, there now exists a type of blueprint for future benchmarks that need to stay fair across different MVEs.

For **RQ2** on the implementation, in Chapter 4 I translated the design into a working system, by extending Yardstick with new components altered to Luanti. This meant creating a Lua collector to capture tick-level data, Rust-based bots to simulate hundreds of players, and and Ansible playbooks to automate deployments on DAS-5 cluster. In practice, I believe that this structure and form of implementation was crucial because it allowed Luantick to integrate with Luanti without modifying the engine, which can also be extended to other engines. As a result, we now have a useable toolchain that researchers can run out of the box, which makes protocol-agnostic benchmarking a reality rather than a theoretical concept.

For **RQ3** about the evaluation, Chapter 5 revealed how Luantick can be used to evaluate MVEs under different network protocols. Using Luantick, I was able to test Luanti under various workloads and scaling conditions. These experiments showed that Luanti's UDP protocol is efficient at lower scales but reaches a ceiling at about 180 concurrent players. Movement diversity and spawn locality shaped performance in different ways, while block placement consistently consumed more resources than walking. I think that these findings highlight the trade-offs between efficiency and scalability, because they clearly showed where Luanti performs well and where it struggles. Additionally, by repeating experiments in a structured manner, Luantick provided reproducible evidence that can be compared across platforms, which is something that earlier studies could not guarantee.

7.2 Limitations and Future Work

All research, like mine, comes with its limitations. This section describes the limitations that I faced during my thesis alongside possible challenges with furture work.

First, a significant limitation that arose was related to server compilation issues, specifically from repeated failures when compiling the Luanti server directly on DAS-5 clusters. To address this, the server was successfully built once on the head node, and the Ansible playbooks were adjusted to copy the executable during deployment. Although this workaround ensured progress, it also reduced flexibility by preventing per run builds. Therefore, I believe that future work in this field could explore stronger compilation pipelines or crammed environments that would avoid such obstacles.

Another limitation arose concerning the Secure Remote Password (SRP) protocol required for Luanti authentication. The handshake involves cryptographic operations that proved difficult to implement from scratch, however this was resolved by adopting an existing Rust library that was maintained by the community, allowing me to focus on implementing the workload instead of implementing the authentication layer for the bots. For future research could add support for other authentication methods or make SRP integration smoother, therefore making it easier to benchmark MVEs that use stricter or custom login systems.

Third, managing distributed experiments across multiple DAS-5 clusters introduced difficulties in verifying server startup. My early tests often failed because the orchestration only confirmed process launch rather than active server readiness. Yet, this was managed by implementing health checks confirming that the server process was active before allowing bots to connect. I think that future work could build on this aspect by adding more advanced coordination tools, such as centralized experiment dashboards or automated recovery in case of cluster failure.

Lastly, the current experiments focused on movement and block placement, and while this captures core interactions, MVEs also involve activities such as crafting, terrain modifications, and combat systems that can create different load patterns. Moreover, future research could expand Luantick with more workloads to capture these behaviors and better reflect the diversity of real gameplay.

7.3 Closing Remarks

Luantick shows that benchmarking MVEs does not need to be connected to a specific engine or protocol. By separating workloads from network details, it became possible to drive Luanti with the same type of player actions and collect comparable metrics without modifying the engine. This work showed that protocol agnostic benchmarking is not only possible but actually useful, because it exposes the strengths and limits of engines when placed under realistic pressure. Together, the answers to the three research questions in Section 7.1, highlight that protocol-agnostic benchmarking of MVEs is possible and valuable. In my opinion, Luantick demonstrates a path forward, especially considering its neutral design, extensible implementation, and an evaluation method that produces fair and reproducible results. My experiments revealed concrete trade-offs between efficiency and scalability, and their results suggest that future work can build on this foundation in a manner to extend workloads, engines, and protocols to to better capture the performance level of MVEs.

References

[1] Michiel Buijsman. **Global Games Market Revenue: 2024 Estimates and Forecasts**, 2024. Blog post, accessed 6 June 2025. 1

[2] Minecraft Education. **Math Essentials with Minecraft Education**, 2025. Resource page, accessed 6 June 2025. 1

[3] Minecraft Education. **New Study: Understanding the Impact of Minecraft in the Math Classroom**, 2020. Blog post, accessed 6 June 2025. 1

[4] Jesse Donkervliet, Javier Ron, Junyan Li, Tiberiu Iancu, Cristina L. Abad, and Alexandru Iosup. **Servo: Increasing the Scalability of Modifiable Virtual Environments Using Serverless Computing**. In *Proceedings of the 43rd IEEE International Conference on Distributed Computing Systems (ICDCS 2023)*, pages 829–840. IEEE, 2023. 1, 2, 39

[5] Hannah Daniels. **STEM Lesson Idea: Using Virtual Learning to Drive Environmental Change**, 2020. Blog post on Mimio Educator, posted 3 Nov 2020, accessed 6 Jun 2025. 1

[6] Diar Kamberi. **Lock-Step Simulation for Modifiable Virtual Environments (MVEs)**. Bachelor thesis, Vrije Universiteit Amsterdam, submitted 20 Aug 2024, 2024. 1, 39

[7] Innovature Inc. **What is benchmarking?** `https://innovatureinc.com/what-is-benchmarking`. Accessed: 2025-08-16. 2

[8] Jerom van der Sar, Jesse Donkervliet, and Alexandru Iosup. **Yardstick: A Benchmark for Minecraft-like Services**. In *Proceedings of the International Conference on Performance Engineering, Mumbai, India, April, 2019*, 2019. 2, 11, 40

[9] Luanti Developers. **Network Protocol**. Luanti Documentation, `https://docs.luanti.org/for-engine-devs/network-protocol/`. Accessed: 2025-08-16. 2, 40

[10] Alexandru Iosup, Laurens Versluis, Animesh Trivedi, Erwin van Eyk, Lucian Toader, Vincent van Beek, Giulia Frascaria, Ahmed Musaafir, and Sacheendra Talluri. **The AtLarge Vision on the Design of Distributed Systems and Ecosystems**. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1765–1776, 2019. 4, 40

[11] r/admincraft. **Paper 1.13.1 async chunk loading/generation [BETA]**. 10

[12] SpigotMC Wiki. **What is Spigot/CraftBukkit/Bukkit/Vanilla/Forge?** 10, 40

[13] GlowstoneMC. **Glowstone**. 11, 40

[14] Jim Gray. *The Benchmark Handbook for Database and Transaction Systems*. Morgan Kaufmann, 1993. 15, 39

[15] Lizzy Fleckenstein. **texmodbot**. `https://github.com/LizzyFleckenstein03/texmodbot`. GitHub repository. 24, 40

[16] Unknown Github User. **minetest libraries**. `https://https://github.com/minetest-rust`. GitHub Repository for Minetest in Rust. 25, 40

[17] Henri E. Bal, Dick H. J. Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank J. Seinstra, Cees G. M. Snoek, and Harry A. G. Wijshoff. **A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term**. *IEEE Computer*, **49**(5):54–63, May 2016. 30

[18] Jerrit Eickhoff, Jesse Donkervliet, and Alexandru Iosup. **Meterstick: Benchmarking Performance Variability in Cloud and Self-hosted Minecraft-like Games**. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2022. 40