Evaluating the Scalability of Minecraft Through Performance Evolution Comparison

Ignacio Taylor
2687687
VU Amsterdam
i.taylor@student.vu.nl

Gabriel Gil Pichelbauer
2726247
VU Amsterdam
g.j.gilyanez@student.vu.nl

Quincy Koper
2901940
VU Amsterdam
q.r.koper@student.vu.nl

Leonardo Marro
2896871
VU Amsterdam
l.marro@student.vu.nl

Marten Fleminks
2676685
VU Amsterdam
m.j.fleminks@student.vu.nl

Albert Bao
2893798
VU Amsterdam
h.bao2@student.vu.nl

Supervisor: Jesse Donkervliet **Group: Lab Group 15**

January 16, 2026

Distributed Systems / Vrije Universiteit Amsterdam / Lab Group 15

Abstract

Video games represent today's most valuable entertainment industry, with Minecraft being the best-selling game of all time. It and other Minecraft-like games (MLGs) feature procedurally generated, player-modifiable worlds. Although it is especially popular to play these games in online multiplayer, these games generally do not scale well to higher player counts, with the quality of experience starting to degrade in the low tens. Prior studies have provided benchmarks for MLGs and otherwise analysed their performance. One aspect of their performance which has been studied is performance evolution: the changes in performance from one release to the next. In this study, we extend an existing benchmarking tool to automatically gather various performance metrics for different releases of MLGs, and use this tool compare recent releases of Minecraft. Based on our experiments, we find that Minecraft 1.18.2 has higher CPU usage than other versions and Minecraft 1.17.2 and 1.18.2 have higher RAM usage than other versions we tested. We also find that as more players join, TPS decreases faster on newer or older versions depending on the workload.

1 Introduction

Video games have become the most popular entertainment medium. On average, 3.32 billion [8] people play video games every day, and the industry grosses $197 billion [9] USD annually. One of the most popular video game genres of recent years revolves around modifiable virtual environments (MVEs). The archetypal example of this genre is Minecraft, which recently became the best-selling game of all time [4]. It features procedurally generated, voxel-based worlds that players can explore and modify as they see fit. They can do so either alone or with other players on a (public or private) server. Other popular Minecraft-like games (MLGs) include Terraria, Factorio and Satisfactory, each of which has also sold millions of copies.

The procedurally generated and mutable nature of the game world in MLGs introduces several performance challenges not commonly found in other genres. This can become problematic in multiplayer. Although many players prefer to play with friends in a shared world, and the open-ended design of these games lends itself particularly well to scaling to larger player numbers, a server's performance can start to degrade in the low tens of players.

Prior research has been performed on the performance of Minecraft. Specifically, Yardstick [12] represents the first benchmark for MLGs, and its extension Meterstick [1] was designed to measure the performance variability of Minecraft. Other research on the performance of video games includes DECAF [2], which considers various genres in the context of cloud gaming, but does not include MLGs. One aspect that prior studies have not analysed is performance evolution. By comparing the performance of different releases of a game, it may be possible to gain better insights into the causes of poor performance.

In this study, we extend Yardstick [12] to apply a

single benchmark to several releases of Minecraft. This benchmark consists of several workloads using AI players, representing typical in-game activities that are especially computationally challenging. We designed these workloads to be reliably repeatable and applicable to a wide range of releases. We analyse the results of running this benchmark on several releases to gain insights into the changes in performance of the relevant game mechanics.

Our contributions in this work can be summarised as follows:

C1 We extend an existing framework for measuring the performance of MLGs (Yardstick [12]) to measure the performance evolution of a game.

C2 We measure the performance evolution of Minecraft across several release versions.

2 Background

Many of today's most popular games feature modifiable virtual environments (MVEs). The archetypical example of this type of game is Minecraft, which is currently the best-selling game of all time [4]. Similar to many other popular games, its gameplay revolves around players exploring and modifying a procedurally generated virtual world and interacting with various AI-controlled non-player characters (NPCs). We refer to such games as Minecraft-like games (MLGs), and to other types of games collectively as non-MLGs.

This section gives a high-level overview of Minecraft's system architecture, as well as common types of workloads. We focus on the aspects that are not featured, or pose fewer challenges, in non-MLGs. This overview largely draws from [12; 1]. Figure 1 provides an overview of the system.

2.1 System Overview

The multiplayer version of Minecraft uses a traditional client-server model. Each client translates player inputs, such as keypresses or mouse movement, into in-game actions like walking forward or looking around. It then sends a list of actions performed by the player to the server. The client also receives the game state from the server and renders it as graphics and audio.

The server manages the game state, which consists of data representing the terrain, players, and other entities. The terrain is represented as a 3-dimensional grid of voxels, called blocks by the community. The movement of players and entities is generally unconstrained by the grid. Based on actions performed by players and entities, as well as environmental effects, the server updates the game state in discrete timesteps. These timesteps are called ticks, and are normally performed at a rate of 20 Hz (i.e. ticks have a duration of 50ms). Under normal conditions, all required calculations can be performed well within the tick duration, leaving some time as a buffer. In this case, the server waits until the full tick duration has passed to ensure a regular passage of time. If the computational load on the server is too high, ticks take longer than the intended duration, leading to a degradation in quality of service (QoS) experienced by the players.

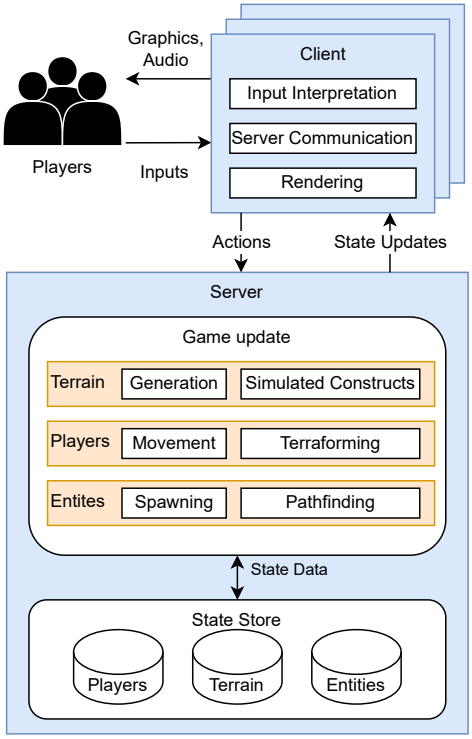

Figure 1: Overview of Minecraft's client-server model, showing examples of each type of workload. Adapted from [1].

2.2 Workload Overview

Minecraft and similar games feature various workloads, several of which are either unique to the genre, or are significantly more challenging to implement efficiently than in non-MLGs. The added challenges are generally caused by the mutable nature of the game world. These workloads can be divided into three categories: those relating to the player(s), the terrain, and entities in the game world. Here, we explain each of these types and, where applicable, give examples of challenging workloads within those categories.

2.2.1 Player Workloads

Challenging workloads generally do not involve player characters directly, but are typically caused by them, either directly or indirectly. For example, by moving near unexplored regions of the game world, the player can cause the game to generate those regions. The player can also create simulated constructs or cause large numbers of entities to spawn, such as livestock

on a farm, which then need to be simulated.

2.2.2 Terrain Workloads

An important type of workload in MLGs, absent in most other game genres, is the terrain workload. The most obvious workload of this type is likely terrain generation, which is done procedurally as a player approaches the edge of the already-generated world. The terrain is generated in constant-sized regions called chunks, which are generated or loaded from memory such that (ideally) players don't perceive any unloaded chunks. Many players moving around quickly in different areas of the world causes many chunks to be generated, loaded and unloaded, which can cause strain on the server.

The workload that is likely the most unique to MLGs involves simulated constructs. These are combinations of interacting blocks and entities that together perform some larger function. For example, a construct made from a pressure plate, wires and turrets can act as a trap for players or entities. Players can build these constructs to be arbitrarily complex. Unlike in non-MLGs, where such systems are prebuilt by developers who can directly link the inputs and outputs using custom game logic, in MLGs the inner workings need to be simulated every time a construct activates.

2.2.3 Entity Workloads

Due to the procedural nature of MLGs, entities cannot appear in predetermined locations. Instead, they are constantly created to ensure the regions that are currently being simulated (i.e. the ones near players) contain enough entities to keep players entertained. This process is called spawning. Because of the terrain is procedurally generated and changeable, it's impossible to spawn entities in predetermined locations, as would be done in non-MLGs. Instead, the game dynamically determines which entities to spawn where, based on players' locations (to ensure entities spawn off-screen) and environmental factors such as light levels, biome type and available space to support the entity.

The changeable terrain also complicates the pathfinding of mobile entities. While non-MLGs can pre-compute a navigation mesh [10] to reduce the computational burden, this is significantly less effective in MLGs, as the navigable space can change at any moment.

3 System Design

This work builds directly on Yardstick, reusing its core idea of controlled, repeatable benchmarking for MLGs while extending the framework to support cross-version evaluation, multi-metric monitoring, and automated multi-run experimentation. Yardstick establishes the foundation consisting of separate nodes for server and agents, controlled workloads, and automated experiment lifecycle management [12]. Our design keeps these principles and makes targeted modifications needed for our study's goal: systematically comparing performance across a wide range of vanilla Minecraft versions using identical workloads and instrumentation.

3.1 Requirements

Because our system operates on top of Yardstick, we adopt its base requirements (repeatability, automation, isolation) without restating them. The requirements below reflect what must be added or modified to enable reliable cross-version performance comparison, which Yardstick itself does not address.

Functional requirements

FR1 *Uniform cross-version execution:* The system must run the exact same workloads, parameters, and measurement procedures on all Minecraft versions under test, despite differences in game mechanics, protocol changes, or server behaviour.

FR2 *Version-agnostic deployment pipeline:* the provisioning layer must be able to fetch, configure, and launch arbitrary vanilla server versions without manual adjustments.

Non-functional requirements

NFR1 *Version-robust workload design:* Workloads must be constructed using stable game mechanics whose operations do not change significantly between Minecraft versions, ensuring that observed performance differences arise from version-level implementation changes rather than workload incompatibilities. This requirement is related to the design of the workloads themselves, as opposed to **FR1** which relates to what the system can do at runtime.

NFR2 *Minimal configuration:* Users should be able to run experiments on multiple game versions by specifying only high-level parameters.

3.2 Design Overview

Figure 2 presents a high-level overview of the experimental system architecture. A Yardstick-based experiment controller coordinates the entire lifecycle of each experiment by selecting the Minecraft server version and workload, allocating dedicated DAS-5 nodes, and starting and terminating executions. On the server node, a vanilla Minecraft server corresponding to the selected version runs the core game loop, handling chunk loading, entity simulation, and redstone logic, while system-level and JVM-level monitoring agents collect resource usage and runtime information. In parallel, a separate client node executes scripted player behaviour through a bot framework based on Mineflayer, which generates controlled, repeatable workloads by

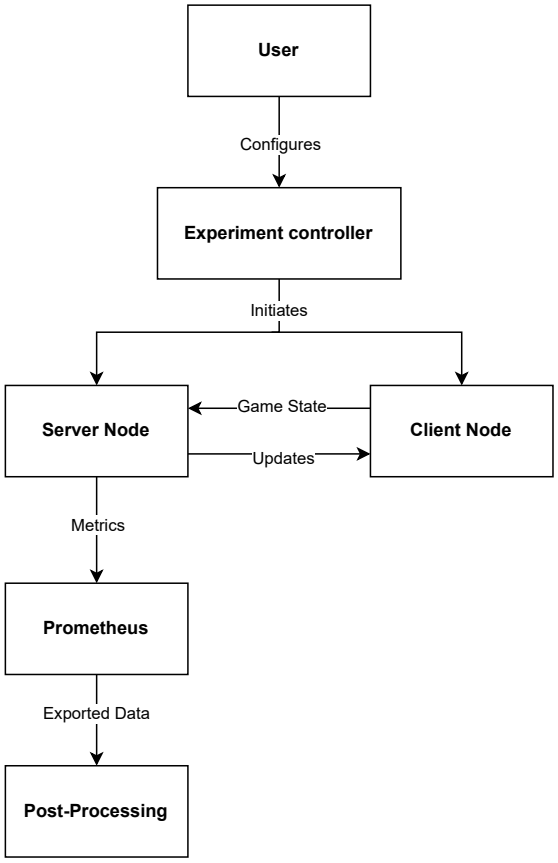

Figure 2: High-level system overview and operation flow.

issuing movement and interaction commands and exchanging game-state updates with the server. During execution, performance metrics from the server are continuously collected and forwarded to a centralized time-series database, enabling synchronized analysis of resource consumption and tick-level behaviour across different workloads and Minecraft versions.

This architectural organization directly reflects the design goals of the system. Its primary goal is to enable fair and reproducible performance comparisons across multiple Minecraft server versions. Minecraft evolves continuously, with changes in internal mechanics, resource management, and implementation details between releases. These changes can significantly affect performance characteristics, even when external configuration and hardware remain identical. Consequently, evaluating only a single version risks drawing conclusions that do not generalize across the lifecycle of the game. This motivates **FR1** and **FR2**, which ensure that identical workloads and measurement procedures can be executed uniformly across versions using a version-agnostic deployment pipeline.

To support meaningful cross-version comparison, workloads must avoid relying on mechanics that behave inconsistently or are deprecated across versions. This requirement directly informs **NFR1**, which emphasizes stability across versions, and **NFR2**, which limits user configuration to high-level parameters. Together, these constraints reduce experimental bias and improve reproducibility by preventing version-specific tuning or manual intervention.

The selected workloads were designed to stress distinct and realistic server subsystems that are known to dominate performance in production environments. Two complementary workload classes were chosen:

- **Memory-intensive workload:** This workload focuses on maximizing the number of simultaneously loaded chunks by spreading players across the world. Chunk loading and retention place sustained pressure on memory usage and garbage collection, especially in newer versions where world height, biome complexity, and lighting calculations have increased. This scenario reflects real-world large servers where players are geographically distributed, for example during exploration or base expansion.
- **CPU-intensive workload:** This workload concentrates activity within a confined area containing large numbers of entities, redstone components, and automated farms (chicken farms). Such setups generate frequent tick updates, entity processing, and redstone evaluations, placing sustained load on the main server thread. This reflects common player behaviour on long-running servers, where players build and remain near resource farms to ensure continuous operation.

Variables

The primary variables manipulated during the experiments include: the Minecraft server version (the main independent variable, selected for each run), the workload type (determines the in-game activities performed by the bots, such as exploration, or farm-related behaviour), the bot count and behaviour (influences client-side load and interaction patterns), and the runtime duration, currently fixed at 30 seconds of active workload but extendable for longer measurements. The dependent variables measured throughout the experiments include: tick time, TPS, CPU usage, memory usage, entity count, and the additional metrics exposed through the Jolokia agent.

Instrumentation

To collect metrics and control the system, the experiment uses several components. The DAS-5 supercomputer allocates compute nodes and ensures uniform hardware across runs, so differences in results can be attributed to the Minecraft version and workload rather than to hardware variability.

A Java runtime is installed on the server nodes. Using the same JVM version and configuration across all runs ensures that the effects are not caused by the server itself.

Telegraf, Jolokia, and Prometheus are deployed on the server node to handle monitoring. Telegraf saves CPU, memory, disk, and network usage at regular intervals. Jolokia is attached to the Minecraft server where it collects JMX metrics, such as heap usage

and garbage collection. Then Prometheus gets both system-level metrics from Telegraf and JVM-level metrics from Jolokia and stores them in a time-series format suitable for later analysis and plotting.

On the client side, a custom bot framework written in JavaScript controls the player's actions to mimic human-like behaviour. The framework uses Mineflayer, which is a Minecraft client protocol library used to connect multiple bots, then place them in specific locations and issue movement and interaction commands in a repeatable way. This setup allows for the same workload to be reproduced in every run and version.

Workloads

We define two workload types (W1 and W2), each representing a class of gameplay interactions that commonly arise in multiplayer Minecraft. These workloads are not designed as extreme stress benchmarks, but instead serve as controlled "probes" for evaluating how different versions of the server handle distinct game mechanics under comparable conditions.

- **W1 — Radial Exploration in Spectator Mode**
  This workload evaluates the performance impact of world generation and chunk loading. As illustrated in Figure 3, all players spawn at a shared origin and immediately fly in evenly spaced 2D directions while in spectator mode, to make sure that the players cannot collide with, e.g. mountains. The world is not pre-generated, so newly visited areas are generated on the fly. This workload captures a canonical multiplayer activity: many players exploring ungenerated terrain at once. It is representative of early-world progression, post-update exploration, and SMP migration phases. Chunk generation and memory growth are thus tested under a realistic exploration-expansion pattern.
  Primary mechanics involved: chunk generation, chunk loading/unloading, memory growth with explored radius.
  Performance sensitivity: RAM usage, disk I/O, chunk-generation overhead, long-term growth characteristics.

- **W2 — Multiple Chicken Farms**
  This workload evaluates the performance impact of entity collision, processing, and mob generation. As illustrated in Figure 4, each player is associated with an identical automated chicken farm placed in a creative-mode flat world. This would ensure that the terrain generation mechanism would not influence the measurements. Each farm operates autonomously once initialized and remains active throughout the experiment, while players remain idle and serve only to keep the corresponding farm actively loaded in the server. This workload captures a common multiplayer pattern: multiple players maintaining independent resource farms that continuously generate entities and item flows.

Figure 3: Illustrative plot for Flying Spectator Workload, where players fly outwards in a circle. The true setup has up to 20 players, a radius of 700 blocks, and players in spectator mode.

Figure 4: Illustrative plot for Chicken Farm Workload, where each player is spectating the workload. The true setup has up to 25 players, with corresponding 25 functioning chicken farms.

Primary mechanics involved: entity collision handling, mob generation and lifecycle updates, hopper-based item transport.
Performance sensitivity: CPU utilization, tick duration stability, and memory usage under increasing entity density.

4 Experiments

This section discusses the experiments we ran and the results we obtained from them. The source code of the experiments is available on GitHub.[1] Our main findings are:

MF1 As the number of players increases, latency and TPS degrade more rapidly in newer Minecraft versions when gameplay is dominated by world generation and exploration. In contrast, under workloads dominated by sustained entity interaction and spawning, newer versions exhibit improved scalability compared to older versions.

MF2 Minecraft version 1.18.2 consistently shows higher CPU utilization and significantly larger

[1] https://github.com/gabgilp/MLG-DS-project

variance across both workloads, with frequent utilization peaks under moderate to high player counts.

MF3 A substantial increase in RAM usage, disk activity, and network throughput is observed between versions 1.17 and 1.18 at higher player counts, exceeding the changes seen between subsequent versions.

Taken together, these findings reflect the impact of major gameplay and engine changes introduced across Minecraft versions. The terrain generation overhaul and expanded world height in version 1.18 substantially increased the computational, memory, and I/O demands of chunk generation, which explains the sharper scalability degradation observed in exploration-heavy workloads and the elevated resource usage in 1.18.2[6]. At the same time, later versions introduced incremental optimizations to entity simulation, redstone processing, and tick scheduling, leading to improved scalability for entity-dominated workloads such as automated farms. Overall, the results indicate that performance evolution across Minecraft versions is closely tied to which game mechanisms dominate server activity, with world generation and entity simulation exhibiting distinct scaling behaviors following major engine updates.

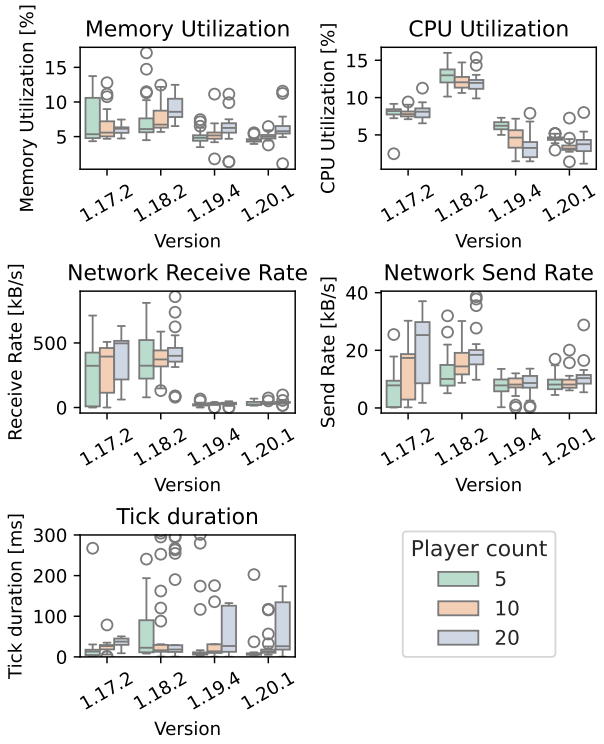

Figure 5: Memory, CPU, network send/receive, and tick duration per player count while running the Flying Spectator workload.

4.1 Setup

Each experimental run reserves two dedicated nodes on DAS-5. The main Python controller runs on the host machine and requests these nodes through the DAS-5 job scheduler. Once the nodes are allocated, the controller deploys the environment: on the server node, Java is prepared, the required Minecraft server version is downloaded, and Telegraf, Jolokia, and Prometheus are installed and configured. On the client node, the bot executor is initialised, and NodeJS is set up to run JavaScript-based bot scripts. After deployment, the server is started with Jolokia collecting runtime metrics. As soon as the server signals readiness, the bots begin connecting from the client node and executing the assigned workload. While the server is running, data is being collected. All metrics collected are then written to Telegraf-compliant CSV files for later analysis and review. After a fixed activity window, the controller terminates the server and stops bot activity. This setup ensures that each performance measurement is executed under identical conditions, with node isolation and automated provisioning removing variability between runs.

4.2 Flying Spectator

The Flying Spectator experiment evaluates how the different Minecraft versions handle CPU load in the case of large-scale exploration executed by the players. This scenario is one of the most common cases that stresses chuck generation and loading. The workload is designed as follows: all the bots loaded for the experiment are set in a radius of 700 blocks around the world origin with a fixed height of y=90. This radius size makes sure that all bots are at least 12 chunks apart from each other, which is the default render distance. All bots are also set in spectator mode to avoid terrain collision and allow them to fly outwards at a constant speed of 10.92 meters/second [5].

The test is split into three phases of one minute with an increased player population. During the first minute, only 5 bots are active. After the second minute, 5 more are added to explore the map in different angles from the starting 5. In the third minute, another 10 bots are added to explore simultaneously. In other words, this workload runs one minute with 5 bots, then one minute with 10 bots, and finally one minute with 20 bots. For each version under test, we record server-side CPU utilization, memory utilization, tick duration, and network send and receive rate over time. This experiment is run 30 times.

The recorded data are plotted as a boxplot over player count and Minecraft versions in Figure 5. The timestamps are offset according to the first bot joining, at which $T = 0$. This offset was determined by looking at the spike in CPU usage. The other metrics use the same offset. For the boxplot, the data is split into 3 separate minutes of measurement, each associated with a player count. For every iteration, the average of the measured data in the entire minute is calculated and used as a value in the boxplot.

The aggregated results in Figure 5 show that CPU utilization grows with bot player count and version

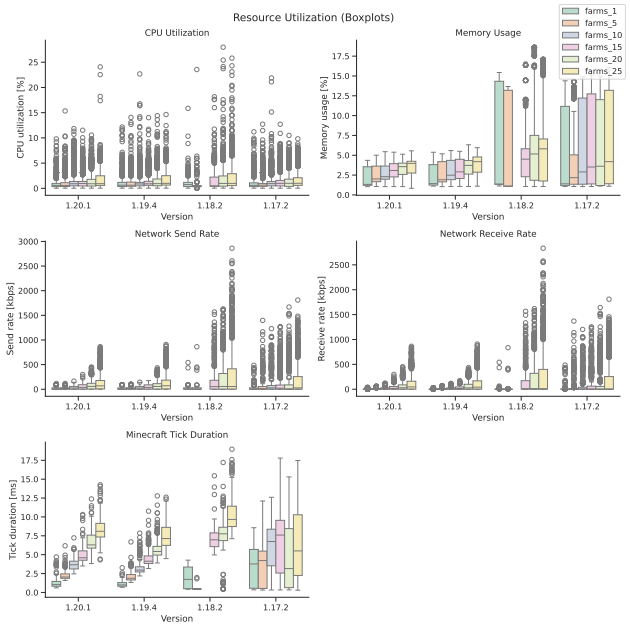

Figure 6: Memory, CPU, network send/receive, and tick duration per farm count while running the Chicken Farm workload.

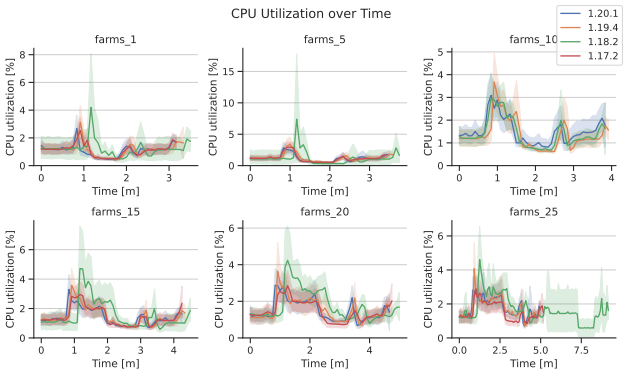

Figure 7: CPU utilization over time across different versions for 1, 5, 10, 15, 20, 25 farm counts while running Chicken Farm workload

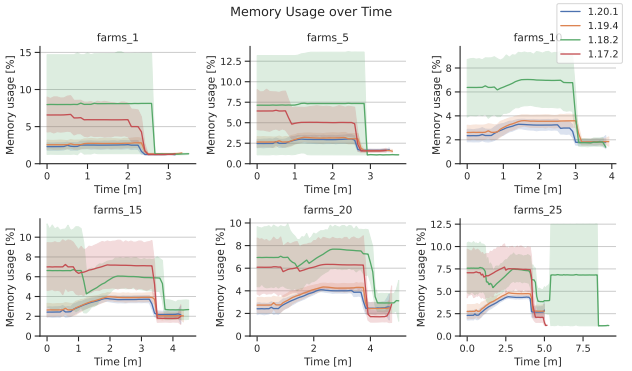

Figure 8: Memory Usage over time across different versions for 1, 5, 10, 15, 20, 25 farm counts while running Chicken Farm workload

complexity. The highest sustained CPU usage is found in version 1.18.2 with a pronounced peak after the first 5 bots joined the server. This peak is likely caused by the terrain generation overhaul introduced in that version [6]. Version 1.19.4 and 1.20.1, on the other hand, display lower CPU usage than 1.18.2, showing that optimisation has been applied to mitigate the exploration cost, although still scaling with additional players. What the results also show is that the network receive rate is much lower in version 1.19.4 and 1.20.1 than in 1.17.2 and 1.18.2. The send rate is also somewhat lower in these versions. This may be related to possible changes in the network protocol introduced in version 1.19.4.

4.3 Chicken Farm

For the chicken-farm workload, we deploy a compact automatic chicken farm with a fixed footprint of 7×3×6 blocks. The farm design relies only on core mechanics: entities, hoppers, dispensers, and simple redstone. Consequently, the farm functions consistently across all tested Minecraft versions. All experiments are conducted in a creative-mode flat world to remove terrain-related variability. Each farm is initialized with the same number of adult chickens and operates autonomously without player interaction. Workload intensity is controlled by the number of concurrently active farms, with experiments conducted at six scales: 1, 5, 10, 15, 20, and 25 farms, where each farm is active by an idle player to ensure that chunks are loading. For each configuration, the workload is tested for a fixed duration of 5 minutes. This workload is designed to test entity collision handling, entity update logic, and related server-side simulation mechanisms.

Across all workload sizes, we observed clear differences across different Minecraft versions in terms of resource utilization and tick duration. In Figure 6, we have a summary of CPU utilization, memory usage, network send/receive rate, and tick duration across all workload sizes and Minecraft versions. CPU utilization increases with the number of farms for all versions, but the distribution varies, and the number of outliers is high, mainly due to the interference of other system tasks. Memory usage also scales with workload size, with version 1.17.2 and 1.18.2 showing consistently higher memory footprints. Network send and receive rates increase as more farms are added as well, but the dispersion differs across versions, especially for version 1.18.2. Tick duration boxplots show increasing medians and widening distributions as the number of farms grows.

Figure 7 shows CPU utilization over time for each workload size. At small scales (1 and 5 farms), CPU usage remains relatively low and stable for all versions, with occasional peaks. As the workload increases to 10 farms and beyond, CPU utilization rises and fluctuates more frequently. For higher farm counts, we discovered a sustained period of high CPU usage. Across different versions, we observed that version 1.18.2 shows a high variance with an overall higher CPU utilization rate, mainly due to the major updates on the gaming mechanism.

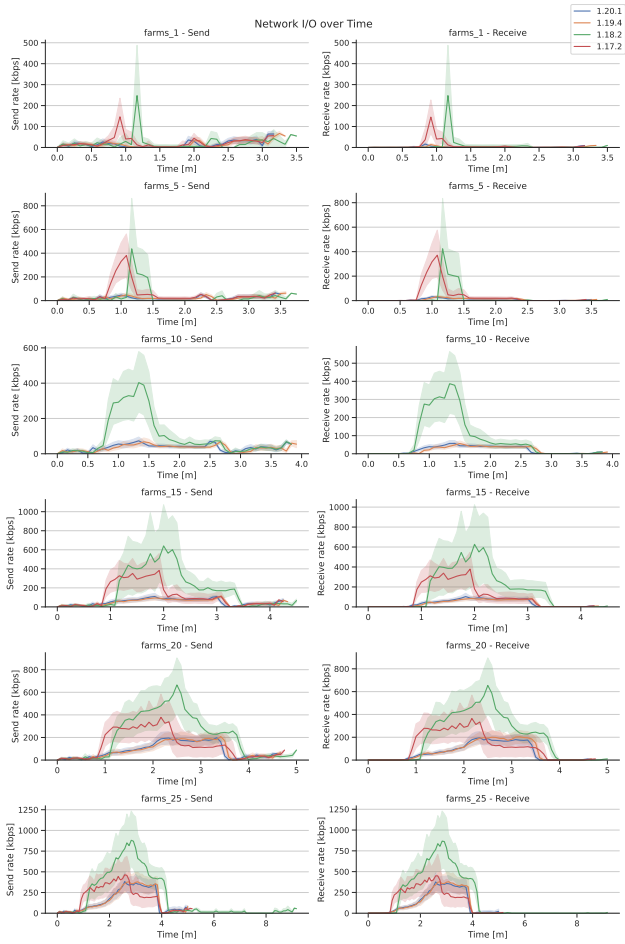

Figure 9: Network Send and Recv rate over time across different versions for 1, 5, 10, 15, 20, 25 farm counts while running Chicken Farm workload

Figure 8 shows memory usage over time. For all versions, memory consumption increases as more farms are introduced, and higher workload configurations maintain high memory levels throughout the experiment duration. Some versions show a gradual upward trend before stabilizing, while others maintain relatively flat memory usage. At larger scales (20–25 farms), memory usage differences between versions become more obvious, with version 1.18.2 consistently occupying more memory and exhibiting larger variability over time.

Figure 9 illustrates network send and receive rates over time. Network activity increases sharply as the number of farms grows. For small workloads, network usage remains low with shorter bursts, whereas for larger workloads, send and receive rates rise and persist for longer intervals. The intervals of peaks are relatively consistent across different versions, but version 1.18.2 demonstrates a higher network rate compared to other versions.

Figure 10 reports Minecraft tick duration over time. Tick duration increases with workload size across all versions, and higher farm counts lead to both higher average tick times and increased variability. For smaller workloads, tick durations remain relatively low and stable, while for larger workloads, the period of

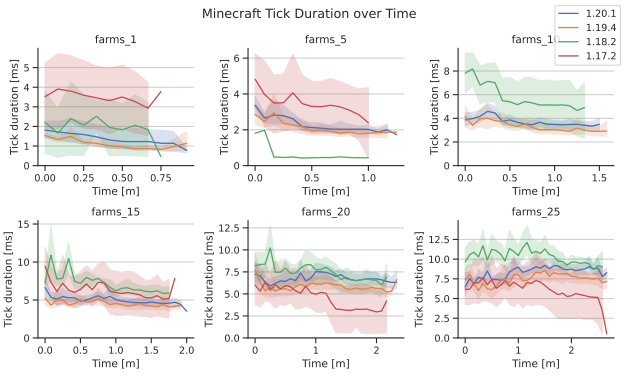

Figure 10: Tick duration over time across different versions for 1, 5, 10, 15, 20, 25 farm counts while running Chicken Farm workload

high tick duration is longer. Differences between versions become increasingly visible at higher scales, version 1.18.2 showing consistently higher tick durations and greater fluctuations than others.

5 Threats to validity

As with any empirical evaluation, several factors may affect the reliability and generality of the results.

Internal Validity: A first source of uncertainty arises from the randomness of in-game entities. Mobs and other simulated elements may behave unpredictably, introducing variation across the runs. Although we perform multiple runs of each experiment, some stochastic behaviour remains unavoidable [7].

Another internal source of noise is the JVM's Just in Time compilation and warm-up, which may temporarily alter CPU usage or tick latency. Our experiment design partially mitigates this by including a warm-up phase before workload execution, though residual effects may still influence short measurement windows [11].

Finally, the DAS-5 nodes introduce an environment that differs from typical server hosting. This difference may raise the possibility that observer behaviour may be partially shaped by the hardware characteristics rather than by the game server alone [3].

External Validity: The realism of the experiment is also limited by differences between vanilla hosting and real Minecraft servers. Many modern Minecraft servers rely on an optimised server software such as Paper, Spigot, or Purpur, which implement advanced scheduling, region-based simulation, and asynchronous operations. Similarly, real servers often implement anti-lag mechanisms and entity caps, reductions not present in our controlled tests. As such, while the experiments capture the scalability of vanilla servers, our observations may not generalize to community-made servers, which are also popular.

6 Conclusion

In this study, we evaluated the scalability of the vanilla Minecraft server across multiple versions using controlled and repeatable workloads. By running automated player behaviour and simulating server stress scenarios, we measured how CPU usage, memory consumption, network activity and tick duration behave on such a workload. Our experiments show that server performance degrades drastically as the number of concurrent players/entities grows, with more recent releases showing higher resource usage under similar conditions.

Two workloads, namely exploration and chicken farm, show clear differences between releases of Minecraft. Most notably, version 1.18.2 consistently shows higher resource usage than all other releases. Although more recent releases show lower resource usage, limitations in scalability remain visible for larger workloads, especially when those workloads involve high numbers of entities.

Overall, the results indicate that the current server architecture struggles to maintain stable tick rates under increasing load, negatively impacting the player experience in large multiplayer settings. These findings highlight the challenge of scaling modifiable virtual environments and provide insight into how performance evolves across versions. Future work could extend this evaluation to additional workloads, alternative server implementations and aimed at newer and updated versions.

References

[1] Jerrit Eickhoff, Jesse Donkervliet, and Alexandru Iosup. Meterstick: Benchmarking performance variability in cloud and self-hosted minecraft-like games extended technical report, 2023. URL: https://arxiv.org/abs/2112.06963, arXiv:2112.06963.

[2] Hassan Iqbal, Ayesha Khalid, and Muhammad Shahzad. Dissecting cloud gaming performance with decaf. *Proc. ACM Meas. Anal. Comput. Syst.*, 5(3), December 2021. doi:10.1145/3491043.

[3] Nils Japke, Furat Hamdan, Diana Baumann, and David Bermbach. Investigating the impact of isolation on synchronized benchmarks. *arXiv preprint arXiv:2511.03533*, 2025.

[4] Asher Madan. Minecraft (likely) becomes the best-selling game of all time on its 10th birthday, 2019. URL: https://www.windowscentral.com/minecraft-becomes-best-selling-game-all-time-its-10th-birthday.

[5] Minecraft Wiki. Flying, 2025. URL: https://minecraft.wiki/w/Flying.

[6] Minecraft Wiki. Java edition 1.18, 2026. URL: https://minecraft.wiki/w/Java_Edition_1.18.

[7] Cristiano Politowski, Fabio Petrillo, and Yann-Gäel Guéhéneuc. A survey of video game testing, 2021. URL: https://arxiv.org/abs/2103.06431, arXiv:2103.06431.

[8] Ravina. Gaming statistics: How many gamers are there in 2026? https://www.affiliatebooster.com/gaming-statistics/, 2026. Accessed: January 15, 2026.

[9] Probaho Santra. Newzoo raises global games market 2025 forecast to $197 billion, December 2025. Accessed: January 15, 2026. Article discusses Newzoo's upward revision of the 2025 global games market forecast to $197 billion (up 7.5% YoY), with breakdowns: mobile $108B (+7.7%), PC $43B (+10.4%), console $45B (+4.2%). URL: https://respawn.outlookindia.com/gaming/gaming-news/newzoo-raises-global-games-market-2025-forecast-to-197-billion.

[10] Greg Snook. Simplified 3d movement and pathfinding using navigation meshes. In Mark DeLoura, editor, *Game Programming Gems*, pages 288–304. Charles River Media, 2000.

[11] Luca Traini, Federico Di Menna, and Vittorio Cortellessa. Ai-driven java performance testing: Balancing result quality with testing time. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 443–454, 2024.

[12] Jerom van der Sar, Jesse Donkervliet, and Alexandru Iosup. Yardstick: A benchmark for minecraft-like services. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19, page 243–253, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3297663.3310307.