MetaPaper: Massively Scaling Minecraft Worlds With Dynamic Load Balancing

Marina López-Alet
Universitat Rovira i Virgili
Spain
marina.lopeza@urv.cat

Stepan Klymonchuk
Universitat Rovira i Virgili
Spain
stepan.klymonchuk@urv.cat

Daniel Barcelona-Pons
Universitat Rovira i Virgili
Spain
daniel.barcelona@urv.cat

Pedro García-López
Universitat Rovira i Virgili
Spain
pedro.garcia@urv.cat

***Abstract*—Minecraft is one of the most popular massively multiplayer environments, supported by an active and open community that continuously works to improve it. Among these contributions, MultiPaper focuses on addressing a critical challenge: the limited scalability of standard Minecraft servers, which restricts the number of users that can interact within a single shared world. Despite its innovations, our novel evaluation of MultiPaper reveals scaling limitations to maintain a good experience for large user counts.**

To address this, we introduce MetaPaper, an extension of MultiPaper that presents novel advancements in Minecraft's multiplayer scalability. We present two key innovations: dynamic scaling, which enables elastic infrastructure and automated server deployment; and dynamic load balancing, allowing seamless player distribution and real-time migration across server instances. Our empirical analysis demonstrates that MetaPaper significantly improves both user experience and system performance, increasing playability and supporting substantially more concurrent users within the same virtual space (from 250 to 800 users).

***Index Terms*—Multi-user virtual environments, metaverse, modifiable virtual environments, Minecraft, cloud computing, load balancing, scalable systems**

I. Introduction

Multi-user virtual environments (MVEs) such as Minecraft, Roblox, or Fortnite share a common limitation: *they do not scale* [1]. Although these platforms have millions of users in total, they are split into many world instances (servers), each limited to a small number of users. For example, Minecraft becomes unusable after 200 users [2], Roblox is API-limited to 700 users, and Fortnite games are constrained to 100 players. This means that users cannot interact in a single massive world as proposed by the *metaverse* idea [3] and they can only see and connect with the local cohort.

The cloud is an ideal substrate for solving this scaling problem, but there are several technical problems that need to be addressed. A critical part of this includes workload distribution, load balancing, and scheduling. To wit, the world simulation must be split into many servers to scale its processing, the load must be correctly balanced since the simulation must progress synchronously, and resource scheduling must be handled very carefully because player load is dynamic and moves around the world whimsically.

Minecraft is particularly challenging since the entire virtual world is highly interactive and malleable by users, also known as a massive modifiable virtual environment [4]. Moreover, the Minecraft ecosystem is particularly interesting because it is supported by an active community that continually works to improve it. In fact, there are efforts from this community that address the scaling problem and have produced several open-source projects, including Mammoth [5] (horizontal scaling with zone-based partitioning), MultiPaper [6] (horizontal scaling with dynamic chunk-based distribution), and Folia [7] and ShreddedPaper [8] (vertical scaling with multithreading).

In this paper, we have analyzed the scalability of these community projects, concluding that only MultiPaper can properly sustain hundreds of users concurrently in a single continuous world (up to 250 with 38 servers). However, we also found significant problems like static resource provisioning and sluggish load balancing that prevent further scaling.

To overcome these limitations, we design MetaPaper as an extension of the MultiPaper framework. MetaPaper introduces two key innovations: (i) *dynamic scaling* to elastically spin up or remove server instances in real time based on simulation performance and player count, and (ii) *dynamic load balancing* to provide responsive player distribution with improved server joining policies and novel real-time migration strategies that enable seamless player transfers across server instances to rebalance load without interrupting gameplay.

MetaPaper presents a novel architecture for Minecraft that integrates dynamic autoscaling, responsive load balancing, and seamless player migration—capabilities not previously available for Minecraft's complex simulated worlds. These features address challenges identified in prior work [1], where traditional approaches like world sharding or instancing disrupt the continuity of metaverse environments. While existing research [2], [9] has focused on single-server benchmarking and

scalability, this work is, to our knowledge, the first to evaluate and optimize multi-server Minecraft deployments. Central to our system is a metric-driven control plane that leverages in-game and system-level indicators—many of which were previously unavailable—to inform real-time infrastructure decisions. These metrics feed a novel heuristic algorithm for load distribution, enabling scalable and responsive gameplay.

We evaluate MetaPaper on diverse workloads with simulated players—idle, movement-heavy, and interaction-intensive. Experiments show that MetaPaper consistently maintains the simulation close to the target of 20 loops per second, even as the player count increases, while MultiPaper quickly drops below 15 and cannot recover. In our largest setup, MetaPaper sustains 800 concurrent players distributed across 38 servers without noticeable gameplay disruption, even during dynamic scaling and live player migration. Instead, MultiPaper's simulation performance starts to degrade irreversibly after 250 connected users in the same setup. To the best of our knowledge, we are the first to scale a single Minecraft world to 800 concurrent players under diverse benchmarks and workloads. MetaPaper improves the performance and responsiveness of the system under load and lays the foundation for building large-scale persistent virtual worlds using commodity game engines.

Our contributions are summarized as follows. We:

- Analyze community projects that scale Minecraft worlds and show their limitations in workload distribution, static resource scheduling, and sluggish load balancing.
- Design MetaPaper, a Minecraft server implementation with elastic resource scheduling and automatic scaling.
- Introduce dynamic load-balancing strategies to MetaPaper for improved workload distribution and real-time load rebalancing with player migration.
- Evaluate MetaPaper against MultiPaper and show that it can support $3\times$ more users in the same setup without disrupting gameplay.

II. Background and motivation

Minecraft is a popular game known for its interactive virtual world, where players break, place, and build with blocks. Multiplayer functionality allows users to host servers on their own machines, but local servers typically support only 10–20 players, and even powerful servers rarely handle over 100 players effectively. This challenge is evident in large public servers that host thousands of players by connecting multiple smaller instances via proxies like Velocity [10] or BungeeCord [11], but players on different servers cannot interact within the same world.

Traditional web autoscaling mechanisms and sharded game server architectures exhibit significant limitations when applied to large-scale multiplayer environments. These approaches typically rely on coarse-grained scaling decisions and isolate players across independent worlds or shards, which restricts interaction and continuity. Moreover, their scaling logic is often decoupled from the game tick, making it difficult to guarantee stable per-tick performance under dynamic workloads.

As a result, existing systems struggle to maintain tick-level stability in a continuous, shared world. They lack high-frequency load evaluation aligned with tick dynamics and do not support coordinated player migration while preserving world consistency. Player transfers, when supported, often rely on heavyweight handoff mechanisms, lacking low-friction coordination and explicit policies that balance performance stability against migration overhead.

Minecraft's simulation is structured around hierarchical spatial and temporal units. The world is composed of *blocks*, individual cubic meters representing terrain and structures. These blocks are organized into 16×16 columns called *chunks*. Chunks extend vertically across the full height of the world and form the fundamental unit of world loading and management. Chunks are loaded dynamically based on player positions. The *simulation distance* defines the area in which chunks are actively simulated, while the *view distance* determines which chunks are transmitted to clients for rendering. Players spawn at a designated *spawn point*, and the *spawn radius* defines the area in which they may appear upon joining.

Temporal progression follows a fixed update loop (*tick*) at 20 *ticks per second* (TPS). During each tick the server simulates world state changes such as entity movement, AI execution, redstone[1] logic, and physics. The duration of each tick computation is measured in *milliseconds per tick* (MSPT), and must remain below 50 ms to maintain real-time responsiveness (the target 20 TPS simulation rate) and avoid lag which impacts player experience.

To improve functionality, third-party implementations such as Spigot [12], Paper [13], and Purpur [14] offer reduced lag, improved tick handling, and plugin support. Meanwhile, Folia [7] and ShreddedPaper [8] introduce multithreading. Nevertheless, despite these improvements, all these projects remain single-server solutions that cannot scale horizontally to support thousands of players in a unified world.

Academic work such as *Manycraft* [15], *Koekepan* [16], [17], and serverless approaches [4] explore world sharding and elastic scaling, but do not provide seamless large-scale shared worlds. Distributed architectures such as MultiPaper [6] and Mammoth [5] maintain a single virtual world across servers, coordinating chunks for load distribution. However, they suffer imbalance and player transfer delay that disrupt gameplay.

Minecraft is attractive for building metaverse-like experiences due to its interactive environment. However, achieving a metaverse-like experience within Minecraft requires sever infrastructure capable of supporting thousands of concurrent players in a seamless, shared world while maintaining smooth performance under variable loads.

Achieving smooth performance under variable loads is particularly challenging in event-driven scenarios where many players may join a server simultaneously, such as during scheduled in-game events or community gatherings. This burst join pattern is frequently mentioned in community discussions,

[1]Minecraft's in-game circuitry and automation system.

which underscores the need for fast, responsive scaling in Minecraft and other metaverse environments.

This motivates the need for a new system that enables dynamic scaling, even load distribution, and seamless player transfers, designed for practical deployment on modern platforms such as Kubernetes.

A. Existing approaches: MultiPaper

After reviewing various community-driven projects, exploring both vertical and horizontal scaling options, we found that Folia [7] and ShreddedPaper [8] (vertical) have extremely low performance, while Mammoth [5] and MultiPaper [6] (horizontal) perform better. In particular, MultiPaper [6] yields the best results for growing player counts. MultiPaper distributes a single Minecraft world across multiple server instances by coordinating chunk ownership, ticking, and real-time synchronization, enabling support for larger player bases. Its main components are the master database, server instances, and a load balancer (e.g., Velocity), provided as JAR files runnable on single or distributed machines.

The *master* manages the world file and data (chunks, player info, etc.) and tracks chunk ownership by servers. Servers ask the master for chunk data, which redirects to the current owner to ensure up-to-date data. The master runs standalone or as a plugin in a proxy server.

The *servers* are individual Minecraft server instances running MultiPaper that own a set of chunks. Each server connects to the master and its peers to sync real-time changes. The master dynamically assigns chunk ownership based on player movement and server availability. When a player connects to a server, the server must obtain information about the chunks around the player. These chunks may or may not be owned by other servers. If they are not, the server requests ownership of the corresponding chunks from the master. If they are already owned by other servers, it subscribes to their updates directly from the other servers. This allows the server to reflect changes made by the owner.

The *load balancer* is included in the MultiPaper master as a built-in proxy to unify servers under one IP. When working as a plugin for other proxy alternatives, it collaborates with them to balance player connections.

Limitations of MultiPaper: Our initial benchmarking of MultiPaper (more details in §VI) reveals important limitations that prevent it from meeting the metaverse-ready goals described above. The key issues are:

- *Uneven player distribution*: MultiPaper's load balancing uses a moving average over 1 min to estimate MSPT. This long window delays the response to load changes, making it unsuitable for dynamic environments—overloaded servers continue to receive players, creating an imbalance in player distribution (§VI-B).
- *Static server deployment*: MultiPaper requires a static server count, preventing adaptation to varying player loads, as seen across all experiments in §VI.
- *Lack of online player migration*: MultiPaper cannot transfer players between servers after connection. Players remain on the assigned server regardless of load. This prevents correct load balancing and limits scalability and responsiveness.

These limitations show MultiPaper's inadequacy for a metaverse-ready Minecraft server, requiring a new approach to meet scalability, load balancing, and performance needs.

III. Design goals

To develop a Minecraft server for a metaverse environment, supporting many concurrent players with seamless scalability and performance, we set the following goals.

1) *Scalability*: The system must dynamically scale resources based on real-time player load, using autoscaling to handle fluctuations without manual intervention, ensuring that the system can handle both sudden spikes and drops in player activity.
2) *Load balancing*: Players should be evenly distributed at any time across multiple server instances to avoid overloading any single server.
3) *Player migration*: Seamless player migration is required to balance loads without disrupting gameplay. The system should be able to move players between servers as needed with no noticeable transition or disruption.
4) *Performance*: The system must maintain high performance under heavy load, measured by TPS or MSPT, ensuring smooth and responsive gameplay to maintain a high-quality user experience.
5) *Flexibility*: The system should support various deployment models, such as Kubernetes, and easily integrate with diverse infrastructures.

This work excludes developing new game mechanics unrelated to scalability, load balancing, or performance, though some supportive features (e.g., server management commands) may be added. The focus is on architecture and infrastructure for a scalable, distributed Minecraft server.

Optimizing single-server performance is not prioritized; this work targets *horizontal scaling*—distributing the game across multiple servers. *Vertical scaling*—enhancing single-server power—is out of scope, as it does not address the challenges of supporting thousands of concurrent players in a continuous world.

IV. System design

To overcome the limitations of MultiPaper and achieve the design goals outlined in §III, we present a novel system architecture that integrates automatic scaling, adaptive load balancing, and dynamic player migration. Built on a distributed, Kubernetes-based infrastructure, the system can handle fluctuations in player population with elasticity and robustness. This architecture prepares the server ecosystem for large-scale, persistent, and dynamic multiplayer environments, such as those found in metaverse-style platforms.

Our proposed architecture design is sourced on an examination of the primary sources of server-side load in Minecraft. Minecraft server performance is primarily driven by chunk management and player distribution. Each server loads and

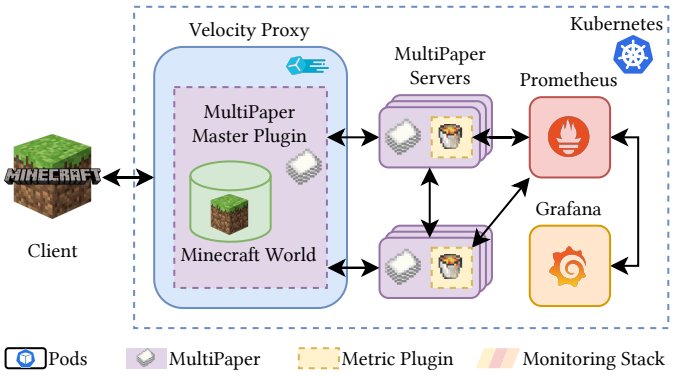


Fig. 1: MetaPaper architecture.

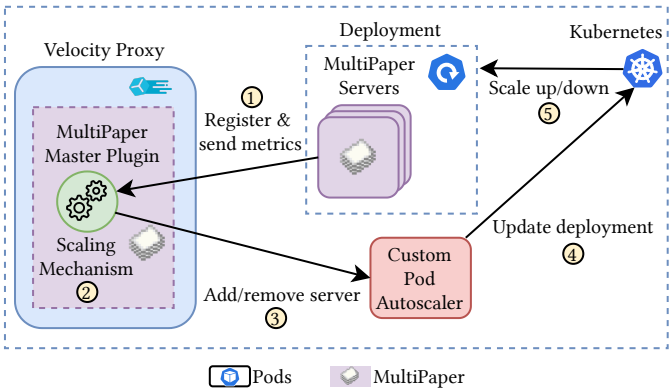


Fig. 2: Dynamic scaling process.

maintains chunks around connected players according to the configured simulation and view distances. While overlapping player regions reduce the total number of loaded chunks, player activity within those chunks still increases computational workload due to entity updates, block events, and interactions. As players move, chunks are dynamically loaded and unloaded, directly impacting tick duration and overall server performance.

In MultiPaper, chunk ownership determines which server instance is responsible for managing and updating a given region of the world. When a player enters a region not owned by its current server, ownership of the corresponding chunks is requested from the MultiPaper master. Conversely, when chunks are no longer required, ownership is released and becomes eligible for reassignment. Servers may also subscribe to non-owned chunks to relay updates to players, although these chunks are not actively simulated.

This chunk-centric execution model underpins both workload distribution and performance behavior in MultiPaper-based deployments and motivates the design decisions presented in the following sections.

An overview of the system's deployment architecture is presented in Fig. 1. This diagram illustrates the key components of the system, such as the master, server instances, custom autoscaler, and monitoring infrastructure, and it highlights the interactions between them within the Kubernetes environment.

At the core of the system is a metric-driven, policy-based control layer. The master collects server and gameplay metrics (MSPT, player count, chunk ownership) to inform policies for server selection, player migration, autoscaling, and instance draining. These policies work together to balance load, redistribute players, scale resources, and maintain stable performance. The following sections detail the implementation of dynamic scaling, player migration, and load balancing within this framework.

A. Dynamic scaling

Dynamic scaling enables automatic adjustment of server instances based on player load, bringing *elasticity* to the system. By scaling out during peak times and in during idle periods, the system optimizes performance and resource usage. This is achieved using Kubernetes, which manages containerized applications and supports dynamic scaling. The mechanism monitors server load and adds or removes instances as needed. The key components are:

- *Server deployment*: Servers run in Kubernetes pods, managed and autoscaled via the Kubernetes API.
- *Custom pod autoscaler*: Works with the master to create or remove pods based on gameplay metrics (MSPT, player distribution). Kubernetes' HPA lacks this load awareness based on gameplay-specific metrics.
- *Scaling algorithm*: Uses MSPT thresholds to scale out or in, adding servers when exceeding a high MSPT threshold and removing them when MSPT falls below a lower threshold, ensuring responsiveness to player activity.

The dynamic scaling mechanism is environment-agnostic and flexible, supporting Kubernetes clusters and standalone infrastructures with minor changes to the autoscaler.

Fig. 2 illustrates the autoscaling process. The key steps are:

1) The servers periodically report their metrics, including MSPT, player count, and chunk ownership, to the master.
2) The master collects these metrics, and the scaling strategy periodically decides whether to add or remove instances.
3) If scaling is needed, the master sends a request to the custom pod autoscaler.
4) The custom pod autoscaler updates the Kubernetes deployment to add or remove instances.
5) Kubernetes manages pod lifecycles, ensuring graceful shutdown by migrating players before termination, applying the drain policy.
6) When instances are added or removed, the master registers or deregisters them from the Velocity proxy. Scaling in requires coordinated player migration.

B. Player migration

Player migration enables dynamic load balancing by redistributing players to balance server load. It is designed to be seamless, allowing uninterrupted gameplay. The key components are:

- *Velocity proxy*: manages player connections and supports player migration via its built-in API for player transfers.

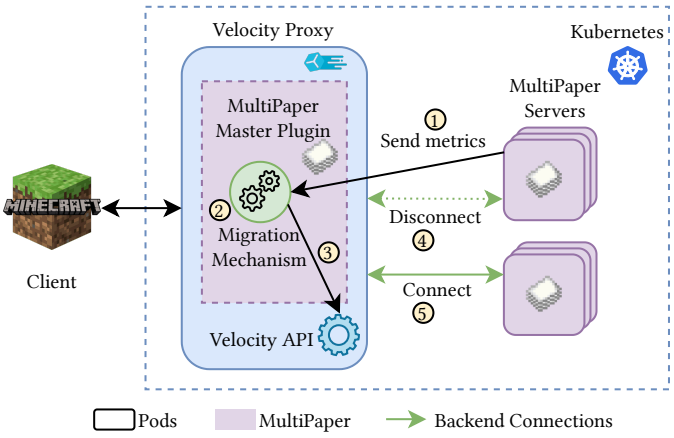


Fig. 3: Player migration process

- *Migration algorithm*: decides when, which players, and where to migrate based on server load (MSPT) and player count; designed for extensibility with alternative migration policies or strategies.

Fig. 3 illustrates the player migration process. The key steps are as follows:

1) Servers periodically report their metrics to the master, which collects them.
2) The migration strategy analyzes the collected metrics and determines how to redistribute players across servers to balance the load. This strategy runs periodically at fixed intervals, ensuring that player distribution is continuously adjusted in response to changing server conditions.
3) The migration mechanism calls the Velocity proxy API to initiate player transfers. The proxy handles the actual transfer of players between servers.
4) Velocity maintains a backend connection to the servers. First, it disconnects the player from the current server.
5) The player is then reconnected to the target server.

a) Proposed player migration algorithm: The player migration algorithm balances the load by moving players from overloaded to underloaded servers. It calculates a target MSPT based on server capacities and derives ideal player counts using per-server efficiency (MSPT per player). To achieve balanced responsiveness and stability, we set a 30 ms MSPT target. Excess players on overloaded servers are redistributed proportionally to underloaded servers' available capacity, ensuring migrations do not overload destinations. This happens iteratively to achieve balanced load.

C. Load balancing

In MultiPaper, load balancing selects the least busy server based on MSPT. However, its long 1200-tick moving average and 20 Hz reporting limit responsiveness and increase network overhead. Sudden changes in load can cause uneven distribution and a degraded user experience. This is further explained in §VI-B.

To mitigate these drawbacks, the MetaPaper load balancing algorithm uses a shorter moving average window. This modification enhances the responsiveness of the system, improving the detection of load fluctuations and more effective distribution of players across servers.

V. Implementation

MetaPaper is a metaverse-ready Minecraft server implemented using MultiPaper and Velocity, with Kubernetes for orchestration. All features are optional via configuration for backward compatibility.

A. Technology Stack

Kubernetes is responsible for the dynamic scaling of MultiPaper server pods. Docker containers utilize existing Minecraft server images. Velocity serves as the proxy layer for player connections and migration. A monitoring stack (Prometheus/Grafana) collects metrics via a custom Bukkit plugin. A custom pod autoscaler component is used to handle scaling requests via REST API, thus avoiding the limitations of Kubernetes' native HPA.

B. Modifications to MultiPaper

To support the system design and achieve the goals in §III, several key modifications were made to the MultiPaper master:

- *Integration with Velocity*: Velocity requires static backend definitions, hindering dynamic scaling. The master was modified to dynamically register and deregister servers with Velocity, keeping the proxy's server list updated without restarts.
- *Extensible policies*: the master now supports pluggable policies for server selection, player migration, and autoscaling. These are defined through configurable interfaces, allowing custom strategies such as variable MSPT windows or adaptive scaling thresholds.
- *Custom pod autoscaler integration*: the master interacts with a custom pod autoscaler via REST, as Kubernetes' HPA proved unsuitable due to its aggregated metrics at deployment level and its scaling behavior. The master makes scaling decisions and the autoscaler applies them.
- *Custom metrics plugin*: a lightweight Bukkit plugin exposes gameplay metrics (TPS, MSPT, player count, chunk ownership) to Prometheus. This plugin enables detailed performance monitoring and visualization in Grafana. The evaluation in §VI relies on these metrics.

C. Modifications to Velocity

Velocity's player transfer mechanism is modified in order to disconnect players prior to reconnecting them to the target server. This is needed to resolve synchronization conflicts with MultiPaper. It introduces minor latency in transfers, which can result in temporary disconnection of players, and it may also result in the potential discarding of chat messages during transfers. A native MultiPaper transfer solution would be optimal; however, this is rendered unfeasible by the constraints imposed by Minecraft's code.

VI. Evaluation

This section evaluates the performance of two distributed Minecraft server solutions: the original unmodified MultiPaper and the enhanced MetaPaper. Each experiment is repeated multiple times to ensure the accuracy of the results. However, due to the time-series nature of the metrics presented, we report a representative execution for each experiment. The evaluation consists of experiments under varying loads, divided into two groups:

- *Original MultiPaper experiments*: assess baseline performance and limitations of the unmodified system with a static number of servers.
- *MetaPaper experiments*: test the effectiveness of the proposed enhancements, focusing on adaptation to workloads via scaling, load balancing, and player migration.

A. Experimental setup and methodology

Experiments were conducted in a Kubernetes cluster with MultiPaper server pods, a master pod acting as the Velocity proxy, and a custom autoscaler. Server pods ran on two nodes (128 CPU cores, 128 GiB RAM each), while the master pod ran on a separate node (48 CPU cores, 32 GiB RAM) with an 8 GiB limit on the Java heap. Player activity was simulated using bots on four additional heterogeneous nodes: 12, 32, 48, and 48 CPU cores, respectively, each with 32 GiB of RAM. The resource usage of each bot pod is dependent on the type and number of bots it manages.

1) Metrics: A monitoring stack with Prometheus and Grafana was used to collect and visualize performance metrics during the experiments. A custom Bukkit plugin on all servers exposes metrics to Prometheus. The collected metrics include:

- *System-level*: CPU and memory usage of master and server pods, and network traffic between pods, gathered via Prometheus querying the Kubernetes API.
- *Application-level*: ticks per second (TPS), tick duration (MSPT), player count and distribution, loaded chunks, chunk ownership distribution, and server response time. These are obtained via the Bukkit plugin, except server response time, which is measured using player bots.

2) Workload generation: The experimental workloads are generated with Mineflayer [18], a library already used in prior studies [9] to create Minecraft bots. Four types of bots were developed to simulate the behavior of real Minecraft players:

- *Idle bots*: connect but perform no actions in the game world; used to measure baseline server load.
- *Simple-walk bots*: move randomly in a bounded area to simulate light activity.
- *Miner bots*: simulate players mining blocks in the game world. They move randomly within a box, but they also place a block and then immediately break it, at fixed regular intervals.
- *PvP bots*: simulate players fighting each other. They move around randomly within a bounded area and engage in combat with nearby bots, incorporating attack cooldowns.

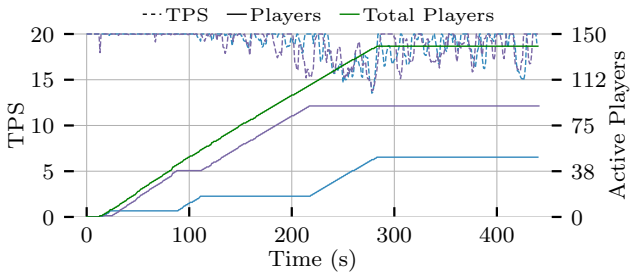


Fig. 4: MultiPaper - TPS and Players per Server (2 servers, up to 140 players): Poor load balancing leads to an uneven distribution of players on each server, causing TPS fluctuations.

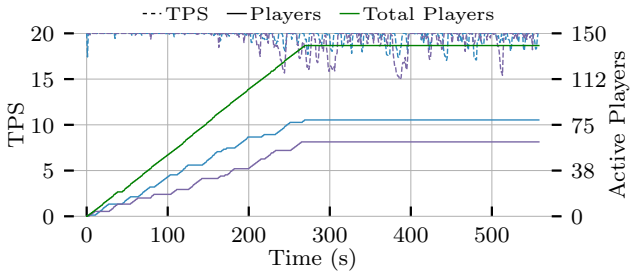


Fig. 5: MetaPaper - TPS and Players per Server (2 servers, up to 140 players): Improved load balancing distributes players more evenly across servers, achieving higher TPS stability.

To simulate different player connection patterns, two different spawn strategies were used:

- *Interval spawning*: bots join steadily at a fixed interval, with a certain number of bots spawned every second.
- *Batch spawning*: bots join in batches (bursts), with a certain number of bots spawned at once and pauses between batches, mimicking real-world player surges.

Bots also measure server response time via timestamped chat messages and are deployed as Kubernetes pods to expose Prometheus metrics.

3) World configuration: To minimize external factors and better reflect MultiPaper's chunk distribution, a flat, peaceful world without structures was used, with a 400-block spawn radius (800×800 blocks). Minecraft servers load chunks around each player according to the configured view distance, dynamically loading and unloading them as players move. Multiple players increase workload due to interactions, though loaded chunks do not scale linearly.

4) Server configuration: All server instances used Aikar's flags [19] for optimized performance and were configured to use 4GB of Java heap memory. All other server settings were left at their default values (including the view distance).

B. Load balancing: MultiPaper vs. MetaPaper

The objective of this section is to present the limitations of the load balancing algorithm implemented by the original MultiPaper. The limitations of the algorithm become evident when considering the duration of the MSPT moving-average window

(1 min). Consequently, MultiPaper experiences challenges in adapting swiftly to load fluctuations. To this end, two experiments were conducted: one using the original MultiPaper, and the other using MetaPaper with the MSPT window optimized, but without performing server scaling or player migration.

Both experiments run on two servers provisioned beforehand. A total of 140 idle bots are introduced using the interval spawn strategy.

a) MultiPaper experiment: Fig. 4 shows the results of the experiment, revealing that the servers begin to show signs of performance degradation when there are around 110 players—the TPS of one of the servers drops below 17. This figure shows the TPS, the number of players connected to each server, and the total number of connected players. Each of the colors represents one of the two servers used for the experiment. Observing this same plot, we can see a problem with the distribution of players across the servers: an imbalance in the number of players, since one of the servers has approximately 90 players, while the other is just below 50 players. This imbalance problem is due to the large moving average window used by the original MultiPaper load balancing algorithm, which is 1200 ticks (1 min). This means that the server selection algorithm takes a long time to react to changes in server load, leading to an uneven distribution of players across servers. In this case, the server with more players has a lower MSPT over the last minute, which is why it is selected by the MultiPaper master to connect new players.

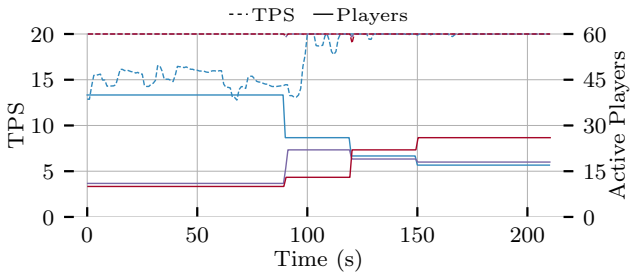
Optimal server performance corresponds to maintaining TPS between 17 and 20, with 20 being Minecraft's target simulation rate (20 Hz). Experimental results show that when TPS drops below 17, server performance degrades: in one test, a server fell below this threshold at around 110 players, with MSPT exceeding 50 ms, indicating inefficient tick processing.

Based on gameplay experience, a sustained TPS below 17 results in noticeable lag, causing actions such as moving, placing blocks or interacting with entities to become delayed or inconsistent and significantly reducing playability. Therefore, while 17–20 TPS ensures a smooth experience, 17 TPS is the practical lower threshold for an acceptable one.

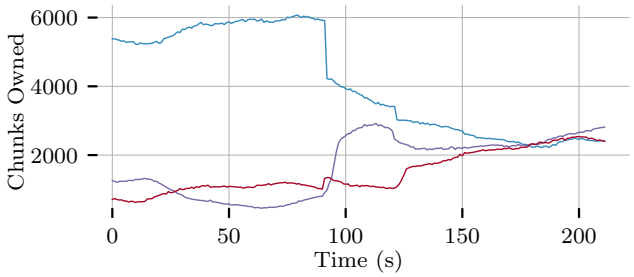
The same problem of imbalance in the number of players connected to the different servers has been observed even with a different number of servers, and it has also occurred regardless of the type of spawn strategy used to connect the bots to the servers.

b) MetaPaper experiment: Fig. 5 shows the results of this experiment using the modified MSPT window. Compared to the original MultiPaper system, these results demonstrate a significant improvement in player distribution across servers. Players are progressively assigned to different server instances without saturating a single server, resulting in a more balanced number of players per server throughout the experiment.

This improvement results from reducing the MSPT moving average window from 60 to 10 seconds, allowing the MetaPaper master to detect load changes more quickly and balance player assignments more effectively. In the original MultiPaper, a large MSPT buffer of 1200 ticks, reported every tick, leads to high communication overhead with poor load distribution and causes some servers to overload while others are underused, limiting scalability. By contrast, MetaPaper employs a significantly smaller MSPT buffer of 200 ticks, which is reported once per second. This results in a 10–second moving average window that enables faster reaction to workload variations while avoiding excessive metric traffic. With MetaPaper's faster feedback loop, load is distributed more evenly. This enables stable performance with more than 120 players, compared to the degradation in the original setup appearing from 90–100 players. This illustrates that improving load balancing is not only about performance symmetry but directly enables higher player capacity, which is a key KPI for any massively multiplayer server infrastructure.



(a) TPS and Players per Server: Player migration balances the number of players and recovers performance to 20 TPS across servers.



(b) Chunk ownership per server: As player migration is applied, the number of chunks loaded by each server equilibrates.

Fig. 6: Player migration in MetaPaper (3 servers, 60 players total).

In all of the following MetaPaper experiments, the same moving average window of 10 s is used for MSPT.

In this experiment, MetaPaper demonstrated a 1.3× improvement in player distribution on servers in comparison to MultiPaper. This efficacy is attributed to MetaPaper's ability to maintain a reduced dispersion of players, thereby attaining a more equitable distribution.

C. MetaPaper: autoscaling and player migration

The experiments presented in this section were conducted using MetaPaper, to show how the new autoscaling and player migration techniques improve server performance compared to MultiPaper, where none of these are applied.

a) MetaPaper with autoscaling: The experiment evaluates the system's ability to dynamically scale out and in based on workload. The experiment starts with a fixed number of

server pods and then increases the number of bots to trigger autoscaling.

The experiment revealed that the scaling strategy can dynamically scale out the number of server pods based on the workload. However, scaling takes some time (around 50 seconds) due to the slow startup time of the Minecraft server instances, which must fully initialize before they can be used. During this period, the TPS of the initial server pod drops significantly as it is overloaded with new player connections.

Moreover, it is a known issue that freshly created server pods initially report high MSPT values, which prevents the MetaPaper master from selecting them for new connections until they stabilize. In some cases, these new pods may even be terminated before being used if they remain underutilized and fall below the MSPT threshold.

These results indicate that although MetaPaper scales effectively under load, startup delays limit immediate relief of overloaded servers. Future work could address this by preemptively scaling based on player arrival rates and spawning pods in parallel. Another option is to combine dynamic scaling with player migration, thus ensuring greater balance of load across newly deployed servers.

b) MetaPaper player migration: This experiment evaluates the system's ability to dynamically migrate players between servers. The experiment starts with three server pods and an unbalanced distribution of players across servers. The MetaPaper master then triggers player migration to balance the load across servers.

At the start, one of the servers is clearly overloaded due to the uneven player distribution, dropping its TPS below 15. As illustrated in Fig. 6a, the migration algorithm is activated at approximately 90 seconds, prompting the MetaPaper master to initiate the redistribution of players. After three migration rounds (30 s apart), the overloaded server players decrease from 40 to 17, while the others increase to 26 and 18. Consequently, the overloaded server recovers 20 TPS, restoring optimal performance. Fig. 6b shows that, due to migrations, chunk ownership across servers also becomes more balanced.

This experiment shows that MetaPaper effectively migrates players between servers, balancing load in both MSPT and chunk ownership. This represents a substantial enhancement over MultiPaper, which does not support player migration. Overall, MetaPaper achieves a $2.8\times$ improvement in player distribution and a $7.4\times$ improvement in loaded chunks distribution compared to MultiPaper.

D. Mixed workloads: MultiPaper vs. MetaPaper

This section evaluates how MetaPaper, through autoscaling and player migration, adapts to varying loads to maintain optimal performance ($\geq$17 TPS), while MultiPaper struggles to maintain optimal TPS. Two experiments have been conducted to compare MultiPaper and MetaPaper under mixed workloads, including movement, building, and PvP, in order to simulate realistic gameplay. Each test used 60 walk bots, 60 miner bots, and 60 PvP bots spawned at intervals. MultiPaper ran on 8 fixed servers, while MetaPaper began with one and dynamically scaled up to 7 as player load increased.

a) MultiPaper experiment: As can be seen in Figs. 7a, 7b, and 7c, the MultiPaper servers cannot maintain optimal performance when handling different types of workloads. In Fig. 7a, the TPS suffers a noticeable decline beginning at 75 players. Specifically, the TPS of one of the servers decreases significantly, reaching lows of less than 15 TPS. Concurrently, MSPT exhibits a substantial increase, as evidenced by Fig. 7b.

Fig. 7c shows that a potential cause of these TPS and MSPT spikes may be the large number of chunks loaded by the server, which puts a significant extra processing stress on that particular server.

In addition to chunk-processing issues, MultiPaper shows poor player load balancing, with some servers hosting nearly twice as many players as others (i.e., one with 41 players, two with 20–25, and the rest with 16–19). As more players join, overloaded servers experience TPS drops and higher MSPT, while others remain underutilized. This imbalance highlights a key limitation of MultiPaper's load balancing under mixed, dynamic workloads.

b) MetaPaper experiment: Fig. 8a shows that MetaPaper handles a total of 180 players, successfully scaling to 7 server pods (one less than the static MultiPaper deployment above). The TPS of the servers is more balanced compared to MultiPaper and stabilizes on all servers around 20 TPS after a few runs of the migration strategy (around 500 seconds into the experiment). The brief drops in player count observed during this phase are caused by seamless player migration, in which players temporarily disconnect from the source server and reconnect to the destination through the proxy.

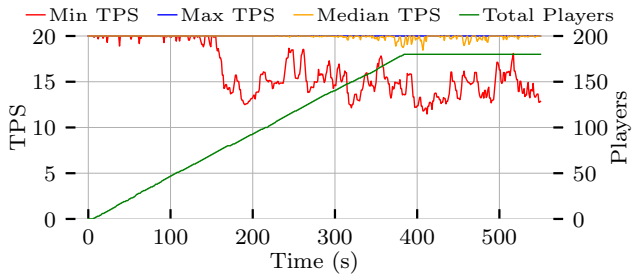
In Figs. 8b and 8c, we can see that the MSPT and chunk ownership are also more balanced across the servers, with MSPT values below 50 ms after the migrations and chunk ownership being more evenly distributed.

This demonstrates that MetaPaper outperforms the original MultiPaper system in handling mixed workloads, as it can effectively scale and balance the load between servers and achieve a $4\times$ reduction in TPS dispersion (range) compared to MultiPaper, resulting in more stable and balanced server performance.

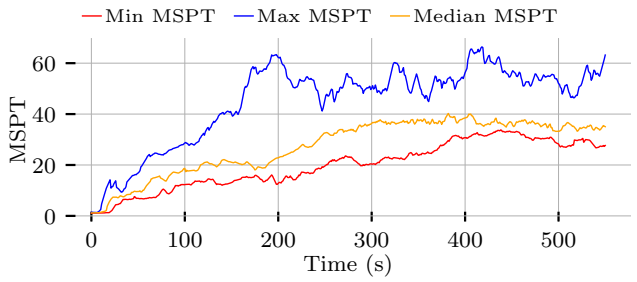
E. High Load Comparison: MultiPaper vs. MetaPaper

In this section, we present the behavior of MetaPaper and MultiPaper under high player load conditions. To this end, we conducted two experiments: one involving MetaPaper, where we employed scaling and player migration strategies, starting with a single server and scaling up to 38; and another involving MultiPaper, which uses 38 static servers. In both scenarios, a total of 800 players were introduced. This scale comes from testbed constraints, which required balancing the number of servers and bots to stress the servers without exhausting resources.

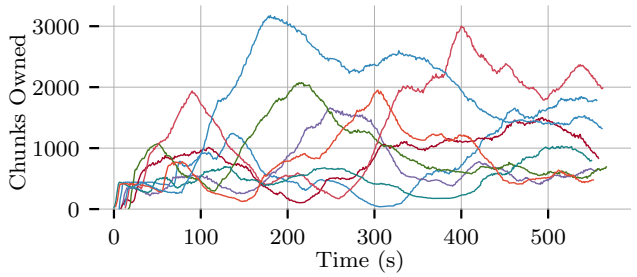
Fig. 9a illustrates a 100-second interval from both experiments, along with the minimum, maximum, and average MSPT values. Additionally, two reference lines have been



(a) TPS and number of players connected: TPS shows significant fluctuations and poor stability as more players join.

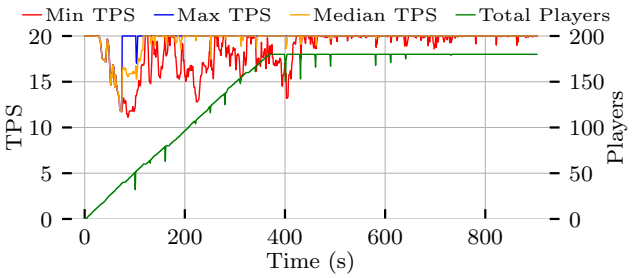


(b) Median MSPT, Minimum MSPT and Maximum MSPT: MSPT exhibits high variance across servers.

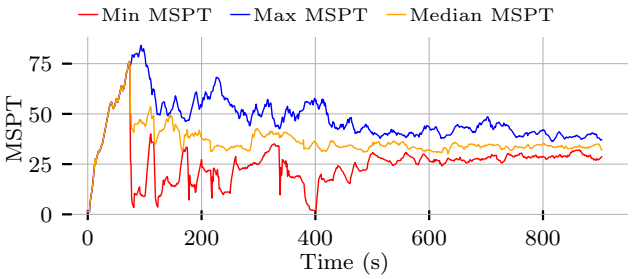


(c) Chunk ownership per server: Ownership distribution remains highly uneven throughout the experiment.

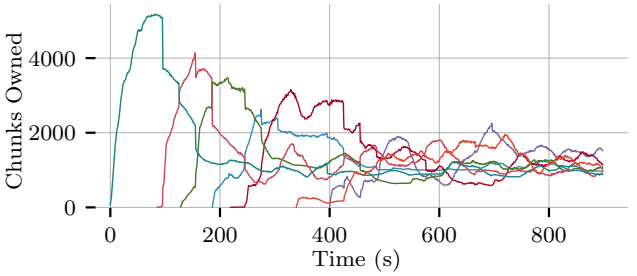
Fig. 7: MultiPaper mixed workloads experiment (8 servers, up to 180 players).



(a) TPS and number of players connected: TPS shows fluctuations as players join but recovers stability thanks to migration. Drops in player count correspond to seamless player migrations between servers.



(b) Median MSPT, Minimum MSPT and Maximum MSPT: MSPT variation decreases over time thanks to player migration.



(c) Chunk ownership per server: Chunk distribution converges to a more balanced state.

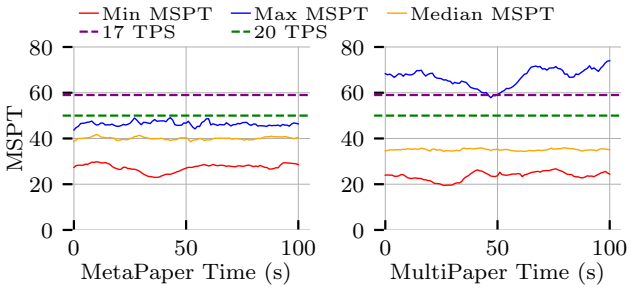
Fig. 8: MetaPaper mixed workloads experiment (up to 7 servers, up to 180 players).

included: one at 50 ms, representing the ideal tick duration required for a server to operate at Minecraft's 20 TPS game loop; and another indicating the upper MSPT threshold (60 ms or 17 TPS) that a server should not exceed in order to ensure a satisfactory gameplay experience.

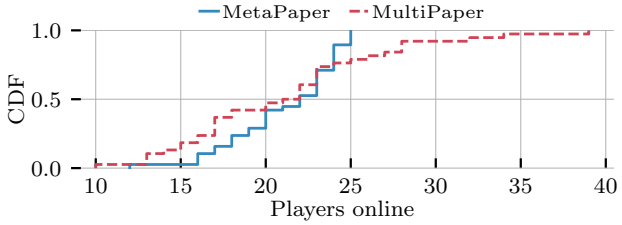
The results show that MultiPaper fails to maintain MSPT within the desired range, and some servers drop below 17 TPS. Its poor load balancing and lack of player migration cause certain servers to overload while others remain idle. In contrast, MetaPaper keeps MSPT below 50 ms by dynamically redistributing players, achieving stable performance. Fig. 9b further illustrates this balance, showing MetaPaper's uniform player distribution (approximately 20-25 per server) compared to MultiPaper's skewed spread.

Fig. 9c illustrates the cumulative distribution function (CDF) of the normalized tick jitter across servers, a metric we refer to as the Instability Ratio, as introduced in [9]. This metric quantifies the stability of game performance by measuring the variability in the durations of consecutive game ticks. Specifically, the Instability Ratio computes the normalized sum of the differences in delay between each pair of consecutive ticks, capturing not only the magnitude of performance fluctuations but also their temporal sequence throughout the trace. This approach enables a more accurate assessment of in-game stability compared to traditional variability measures. In this context, lower jitter values are indicative of superior user experience, as elevated jitter reflects increased variability and reduced system stability. While Fig. 9b captures load balance, Fig. 9c directly reflects its effect on user experience and system robustness.

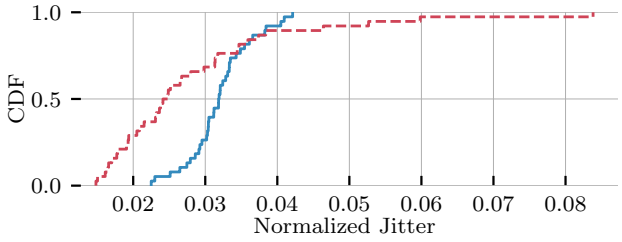
The CDF plot reveals a clear divergence in the distribution of normalized jitter across the two implementations. MultiPaper exhibits a wider spread, whereas MetaPaper offers significantly more consistent tick durations and thus better game experience and overall stability. This shows that MetaPaper's



(a) MetaPaper achieves MSPT under 50 ms with 800 users, enabling gameplay at 20 TPS. MultiPaper fails to keep TPS at 20 for all servers in this scenario.



(b) Player distribution across servers is more balanced in MetaPaper, contributing to better user experience.



(c) Server jitter clearly shows user experience. While MetaPaper keeps jitter low on all servers, some servers in MultiPaper have much higher jitter due to load imbalance.

Fig. 9: Performance of MultiPaper and MetaPaper with 800 players and 38 servers.

balanced player distribution combined with higher-frequency load evaluation, elastic scaling, and migration effectively reduces jitter and avoids the long tails seen in MultiPaper.

The findings demonstrate that MetaPaper consistently maintains optimal performance under high loads by implementing enhanced load balancing, dynamic scaling, and player migration. In contrast, MultiPaper experiences challenges in achieving this. Despite the limitations imposed by cluster capacity, MetaPaper demonstrated its ability to scale further with additional hardware. Future work will assess the scalability limits of the system and, in particular, the master component.

VII. Discussion

This work significantly impacts the design of scalable distributed systems in multiplayer environments, extending beyond Minecraft toward metaverse-like platforms. By enabling servers to support more concurrent players in a unified world, it demonstrates practical strategies for dynamic scaling, load balancing, and player migration beneficial for multiplayer games and distributed applications. Emphasizing the balance between player distribution and computational load offers insights applicable to many distributed systems.

However, there are several limitations. The current player migration process discards in-transit chat messages to prevent disconnections, which can disrupt communication. Also, testing has been primarily in controlled environments with simulated bots, requiring further validation with real players. Additionally, reliance on Kubernetes for scaling may limit deployment in contexts where it is impractical, despite the system's adaptability to other platforms. Finally, during sudden bursts of player connections, server startup latency can temporarily overload existing servers before newly spawned instances become available. Although MetaPaper mitigates load imbalances through migration and elastic scaling, slow server initialization can still cause short-term performance degradation during extreme bursts.

It is also important to note that the scale achieved in our experiments is limited by the available testbed infrastructure and not by inherent limitations of the MetaPaper architecture. All evaluations were performed using a fixed hardware configuration, and the system was successfully scaled up to the maximum number of pods, including both server pods and bots, that the testbed could support without exhibiting stability issues or performance degradation attributable to MetaPaper itself. Therefore, the upper limits observed in our experimental results reflect the constraints of the testbed, not the architectural scalability limits of the proposed system.

Several directions for future work could enhance system capabilities. These include improving seamless world migration and chunk ownership transfers between servers, and enhancing in-transit message handling to support uninterrupted communication during player transfers. Research into advanced load balancing approaches, such as using machine learning for server load prediction, could further optimize resource allocation. Comprehensive testing with actual players remains crucial to uncover real-world challenges. Future efforts may also investigate location-aware policies, such as region-aware assignment, soft affinity, and chunk clustering, to optimize placement decisions further.

Proactive and reactive mitigation strategies could also be explored, such as pre-warming servers based on MSPT thresholds or incorporating flow control mechanisms at the MetaPaper master to rate-limit player admission during scaling events.

Lastly, enhancing observability for detailed activity tracking and integrating machine learning for adaptive autoscaling may improve system performance under varying workloads, paving the way for a more robust infrastructure in Minecraft as a metaverse platform and providing insights into scalable distributed multiplayer systems.

VIII. Related Work

The development of scalable multiplayer online games (MOGs) and massively multiplayer online games (MMOGs)

has been a focal point of research, particularly as virtual environments evolve toward metaverse-like experiences. To this day, there is limited research focused in scalable multi-user virtual environments for Minecraft-like games (MVEs), as they introduce unique challenges due to their persistent, user-modifiable worlds.

Benchmarking tools such as Yardstick [2] and Meterstick [9] simulate realistic workloads using bots, supporting our evaluation strategy. [20] similarly utilize Minecraft bots to optimize server performance. Earlier systems such as Donnybrook [21] and Colyseus [22] address scalability in fast-paced games through peer-to-peer architectures and predictive replication but they are less applicable to persistent MVEs.

Distributed approaches in systems such as Manycraft [15] and Koekepan [16], [17] experiment with sharding Minecraft across multiple servers but lack seamless cross-server integration. Peer-to-peer strategies (e.g., zoned federations) offer decentralization benefits but suffer from consistency and security challenges [23], [24]. Load balancing techniques [25] such as locality-aware region reassignment [26] and neural network-driven autoscaling [27], [28] highlight the importance of dynamic resource management, which MetaPaper addresses using real-time in-game metrics.

Although MetaPaper does not directly adopt prior interest or consistency management models [29] (e.g., vector-field consistency [30], interest filtering [31], [32]), its architecture—based on dynamic chunk ownership and server subscriptions—lays groundwork for future integration of such mechanisms.

Commercial platforms such as Roblox [1] and Improbable's [33] SpatialOS [34], and open-source systems such as Overte [35] and Vircadia [36] demonstrate large-scale virtual world deployment, but lack Minecraft-specific support for world partitioning and modded gameplay. MetaPaper advances the state of the art by combining dynamic autoscaling, load balancing, and live player migration within Minecraft's architecture, offering a scalable and immersive platform for metaverse-like experiences.

IX. Conclusions

This paper has explored the challenge of scaling Minecraft servers towards a metaverse-like environment. Through a detailed analysis of the MultiPaper framework, we identified critical problems in resource management, workload distribution, and online load balancing, hindering the scalability of the system and limiting distributed deployments to just 250 concurrent users in a single virtual world. We introduced MetaPaper, an enhanced system that integrates dynamic resource autoscaling, responsive load distribution, and seamless player migration across servers.

Our implementation targets cloud-native environments, enabling elastic horizontal scalability and real-time adaptability to changing workloads. Experimental evaluation demonstrates that MetaPaper improves resource utilization and is able to maintain the target simulation loop for much larger player counts, supporting 800 concurrent users in the same interactive environment in our largest setup, a $3\times$ improvement over the baseline.

As future work, we plan to reach the scaling limits of MetaPaper. Our architecture can theoretically scale horizontally to many more servers and users, although we foresee that the master component could become a communication bottleneck.

Our findings demonstrate the potential of distributed, autoscalable architectures to support persistent, user-driven virtual worlds. Although significant challenges remain, this work makes a foundational step toward enabling massive modifiable virtual environments with thousands of concurrent users engaging with one another and with the virtual space. Although MetaPaper is grounded in the Minecraft ecosystem, its architectural principles offer broader relevance to scalable multiplayer systems and hold promise for advancing future metaverse platforms.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable feedback and constructive comments, which have helped to improve the quality of this paper. This work has been partially funded by the European Union through the Horizon Europe NEARDATA project (101092644).

References

[1] A. Haeberlen, L. T. X. Phan, and M. McGuire, "Metaverse as a service: Megascale social 3d on the cloud," in *Proceedings of the 2023 ACM Symposium on Cloud Computing*, ser. SoCC '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 298–307. [Online]. Available: https://doi.org/10.1145/3620678.3624662

[2] J. van der Sar, J. Donkervliet, and A. Iosup, "Yardstick: A benchmark for minecraft-like services," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 243–253. [Online]. Available: https://doi.org/10.1145/3297663.3310307

[3] N. Stephenson, *Snow Crash*. Bantam Books, 1992.

[4] J. Donkervliet, A. Trivedi, and A. Iosup, "Towards supporting millions of users in modifiable virtual environments by redesigning Minecraft-Like games as serverless systems," in *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association, Jul. 2020. [Online]. Available: https://www.usenix.org/conference/hotcloud20/presentation/donkervliet

[5] WorldQL, "Mammoth," 2025. [Online]. Available: https://github.com/WorldQL/mammoth

[6] MultiPaper, "Multipaper," 2025. [Online]. Available: https://github.com/MultiPaper/MultiPaper

[7] PaperMC, "Folia," 2025. [Online]. Available: https://github.com/PaperMC/Folia

[8] ——, "Shreddedpaper," 2025. [Online]. Available: https://github.com/MultiPaper/ShreddedPaper

[9] J. Eickhoff, J. Donkervliet, and A. 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)*, 2022, pp. 147–149.

[10] PaperMC, "Velocity," 2025. [Online]. Available: https://github.com/PaperMC/Velocity

[11] SpigotMC, "Bungeecord," 2025. [Online]. Available: https://github.com/SpigotMC/BungeeCord

[12] ——, "Spigotmc," 2025. [Online]. Available: https://www.spigotmc.org/

[13] PaperMC, "Paper," 2025. [Online]. Available: https://github.com/PaperMC/Paper

[14] PurpurMC, "Purpur," 2025. [Online]. Available: https://github.com/PurpurMC/Purpur

[15] R. Diaconu, J. Keller, and M. Valero, "Manycraft: Scaling minecraft to millions," *2013 12th Annual Workshop on Network and Systems Support for Games (NetGames)*, pp. 1–6, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:15048224

[16] H. A. Engelbrecht and G. Schiele, "Koekepan: Minecraft as a research platform," *2013 12th Annual Workshop on Network and Systems Support for Games (NetGames)*, pp. 1–3, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:8422090

[17] ——, "Transforming minecraft into a research platform," *2014 IEEE 11th Consumer Communications and Networking Conference (CCNC)*, pp. 257–262, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:10917284

[18] PrismarineJS, "mineflayer," 2025. [Online]. Available: https://github.com/PrismarineJS/mineflayer

[19] PaperMC, "Aikar's flags tuning guide," 2025. [Online]. Available: https://docs.papermc.io/paper/aikars-flags

[20] M. Cocar, R. Harris, and Y. Khmelevsky, "Utilizing minecraft bots to optimize game server performance and deployment," *2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pp. 1–5, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:25292999

[21] A. R. Bharambe, J. R. Douceur, J. R. Lorch, T. Moscibroda, J. Pang, S. Seshan, and X. Zhuang, "Donnybrook: enabling large-scale, high-speed, peer-to-peer games," in *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2008. [Online]. Available: https://api.semanticscholar.org/CorpusID:7983556

[22] A. R. Bharambe, J. Pang, and S. Seshan, "Colyseus: A distributed architecture for online multiplayer games," in *Symposium on Networked Systems Design and Implementation*, 2006. [Online]. Available: https://api.semanticscholar.org/CorpusID:7660194

[23] A. Yahyavi and B. Kemme, "Peer-to-peer architectures for massively multiplayer online games: A survey," *ACM Comput. Surv.*, vol. 46, pp. 9:1–9:51, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:6250370

[24] T. Iimura, H. Hazeyama, and Y. Kadobayashi, "Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games," in *Network and System Support for Games*, 2004. [Online]. Available: https://api.semanticscholar.org/CorpusID:14475758

[25] F. Lu, S. E. Parkin, and G. Morgan, "Load balancing for massively multiplayer online games," in *Network and System Support for Games*, 2006. [Online]. Available: https://api.semanticscholar.org/CorpusID:15161540

[26] J. Chen, B. Wu, M. DeLap, B. Knutsson, H. Lu, and C. Amza, "Locality aware dynamic load management for massively multiplayer games," *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2005. [Online]. Available: https://api.semanticscholar.org/CorpusID:1345996

[27] V. Nae, A. Iosup, S. Podlipnig, R. Prodan, D. H. J. Epema, and T. Fahringer, "Efficient management of data center resources for massively multiplayer online games," *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2008. [Online]. Available: https://api.semanticscholar.org/CorpusID:11382917

[28] V. Nae, A. Iosup, and R. Prodan, "Dynamic resource provisioning in massively multiplayer online games," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 3, pp. 380–395, 2011.

[29] H. Liu, M. Bowman, and F. Chang, "Survey of state melding in virtual worlds," *ACM Comput. Surv.*, vol. 44, pp. 21:1–21:25, 2012. [Online]. Available: https://api.semanticscholar.org/CorpusID:8741314

[30] N. Santos, L. Veiga, and P. Ferreira, "Vector-field consistency for ad-hoc gaming," in *International Middleware Conference*, 2007. [Online]. Available: https://api.semanticscholar.org/CorpusID:488899

[31] J.-S. Boulanger, J. Kienzle, and C. Verbrugge, "Comparing interest management algorithms for massively multiplayer games," in *Network and System Support for Games*, 2006. [Online]. Available: https://api.semanticscholar.org/CorpusID:5782826

[32] E. S. Liu and G. K. Theodoropoulos, "Interest management for distributed virtual environments," *ACM Computing Surveys (CSUR)*, vol. 46, pp. 1 – 42, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:18561662

[33] Improbable, "Improbable official website," 2025. [Online]. Available: https://www.improbable.io/

[34] ——, "Spatialos," 2025. [Online]. Available: https://www.spatialos.co/

[35] Overte, "Overte project," 2025. [Online]. Available: https://overte.org/

[36] Vircadia, "Vircadia project," 2025. [Online]. Available: https://vircadia.com/