Meterstick: Benchmarking Performance Variability in Cloud and Self-hosted Minecraft-like Games

Jerrit Eickhoff
J.D.Eickhoff@student.tudelft.nl
Delft University of Technology
Delft, Netherlands

Jesse Donkervliet
J.J.R.Donkervliet@vu.nl
Vrije Universiteit Amsterdam
Amsterdam, Netherlands

Alexandru Iosup
A.Iosup@vu.nl
Vrije Universiteit Amsterdam
Amsterdam, Netherlands

ABSTRACT

Due to increasing popularity and strict performance requirements, online games have become a workload of interest for the performance engineering community. One of the most popular types of online games is the Minecraft-like Game (MLG), in which players can terraform the environment. The most popular MLG, Minecraft, provides not only entertainment, but also educational support and social interaction, to over 130 million people world-wide. MLGs currently support their many players by replicating isolated instances that support each only up to a few hundred players under favorable conditions. In practice, as we show here, the real upper limit of supported players can be much lower. In this work, we posit that performance variability is a key cause for the lack of scalability in MLGs, investigate experimentally causes of performance variability, and derive actionable insights. We propose a novel operational model for MLGs and use it to design the first benchmark that focuses on MLG performance variability, defining specialized workloads, metrics, and processes. We conduct real-world benchmarking of MLGs, both cloud-based and self-hosted, and find environment-based workloads and cloud deployment to be significant sources of performance variability: peak-latency degrades sharply to 20.7 times the arithmetic mean, and exceeds by a factor of 7.4 the performance requirements. We derive actionable insights for game-developers, game-operators, and other stakeholders to tame performance variability.

CCS CONCEPTS

• **Software and its engineering** → **Interactive games**; • **General and reference** → **Performance**; • **Computer systems organization** → *Cloud computing*.

KEYWORDS

Meterstick, Benchmarking, Workloads, Performance Variability, Online Games

ACM Reference Format:
Jerrit Eickhoff, Jesse Donkervliet, and Alexandru Iosup. 2023. Meterstick: Benchmarking Performance Variability in Cloud and Self-hosted Minecraft-like Games. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23), April 15–19, 2023, Coimbra, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3578244.3583724

ICPE '23, April 15–19, 2023, Coimbra, Portugal

ACM ISBN 979-8-4007-0068-2/23/04.
https://doi.org/10.1145/3578244.3583724

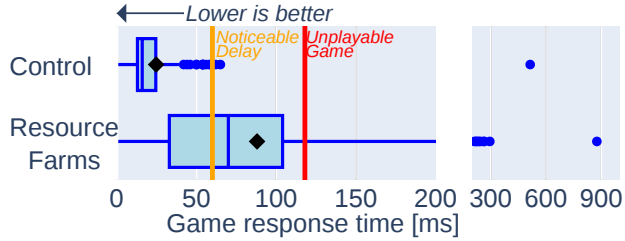


Figure 1: Minecraft response time in the AWS cloud.

1 INTRODUCTION

The gaming industry is the world's largest entertainment industry [39]—world-wide, games engage over 3 billion players and yield over $170 billion in revenue [36]. In this work, we focus on *Minecraft-like Games (MLGs)*, an emergent and highly popular type of game where users can change almost every part of the environment. The canonical example of an MLG is Minecraft, which is already the best-selling game of all time [31]. All MLGs, including Minecraft, present an important challenge to the performance engineering community: although their user-bases can exceed 100 million active users per month, their *scalability* is limited to only 200-300 players even under very favorable conditions [47]. (MLGs support high concurrency by creating separate replicas of their virtual worlds, essentially sharding state and not allowing cross-instance interaction.) *What limits MLG scalability?* In this work, we posit performance variability is a key limit to MLG scalability, and design and use an MLG benchmark focusing on this concept.

MLGs represent an important and unique class of online multiplayer games. Most importantly, MLGs allow players to create, modify, and remove in-game objects (e.g., player apparel) and geographical features (e.g., terrain) [18]. Moreover, some game objects and features are self-acting, that is, they act even when no player input is applied to them. Players can use them to create dynamic elements, by "programming" the environment with combinations of self-acting parts.

Performance variability prevents MLG service providers from giving strict Quality of Service (QoS) guarantees, and simultaneously incentivizes overprovisioning of resources and limiting the number of players that can interact together. For example, Minecraft Realms, a Minecraft *cloud-based service* offered by Microsoft, limits the number of players per game-instance to at most 10 (ten)**!** In contrast, Hypixel, at 216,762 online players [34] the most populated Minecraft server, achieves high player-count by stitching together thousands of (independent) MLG instances using specialized tools, but players in different instances cannot interact.

In this work, *we show for the first time empirical evidence that current MLGs experience significant performance variability.* Figure 1 depicts an exemplary result—even with a single connected

player, the response time varies from good (below 60 ms) to *unplayable* (above 118 ms). We discuss this and similar real-world experiments in §5.2.

Furthermore, *ours is the first study to systematically analyze the effects of performance variability on the operation of MLGs.* By designing and using for this purpose a novel benchmark, we provide an important complement to an emerging body of knowledge. Game researchers and engineers are already aware of the impact of several types of performance variability. Performance variability in networks causes players to stop playing sooner [11], and there are widespread techniques in industry to prevent variability in frame rates [24, 45]. However, the effect of performance variability on the interactive simulation of virtual worlds, and in particular on MLGs, is much less understood.

Prior work in understanding the performance of MLGs [26, 47] and on improving their scalability [15, 17, 23] forms a valuable contribution to the field, but does not currently consider explicitly performance variability. Addressing this important gap, we make a four-fold contribution:

C1 We propose an operational model of MLGs. Ours is the first to consider MLG-specific workloads (§2). Because MLGs allow players to program the virtual environment and terraform the terrain, they support new types of workload not available in most traditional online games.

C2 We design Meterstick, a benchmark that quantifies performance variability in MLGs (§3). To this end, we propose a novel performance variability metric, and define a benchmarking approach to produce it experimentally. Our benchmark supports common deployment-environments for MLGs offered as a service, in particular, both cloud-based and self-hosted. Our benchmark is the first to quantify performance variability in MLGs.

C3 We conduct real-world experiments using Meterstick (§5) and, after analyzing the results, propose actionable insights (§6). We evaluate the performance variability of three popular MLGs, running on two popular commercial cloud providers and one local compute-cluster.

C4 Following open-science and reproducibility principles, we publish Findable, Accessible, Interoperable, and Reusable (FAIR [49]) data, available on Zenodo [21], and Free-access Open-Source Software (FOSS) artifacts, available on GitHub [20].

2 OPERATIONAL MODEL OF MINECRAFT-LIKE GAMES

For contribution **C1**, first, we summarize a state-of-the-art operational model and a reference architecture for MLGs (§2.1). Second, we define MLG-specific *environment-based workloads* that are caused by terrain and entity simulation; §2.2 defines the resulting MLG workload model. Third, we model the operational elements of these workloads (§2.3).

2.1 Reference Architecture for MLGs

We leverage in this work a common reference architecture for MLGs [47]. As Figure 2 depicts, MLGs use a client-server architecture and are commonly deployed in cloud environments. Players run a client on their own device, which connects to a server instance running in the cloud. Some MLG developers publicly distribute their

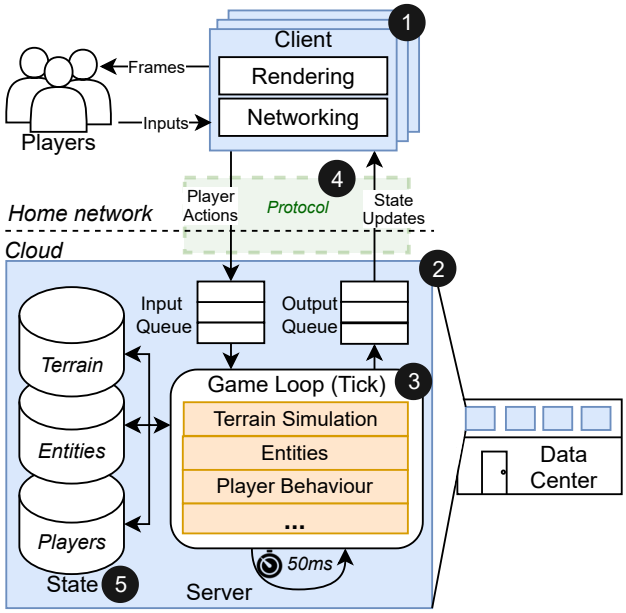


Figure 2: Overview of an MLG.

server software, allowing players to self-host game instances. Popular cloud providers such as Amazon AWS and Microsoft Azure provide tutorials for running these servers on their platform. Microsoft, the company that currently owns Minecraft, markets it as a cloud-based service through *Minecraft Realms*, which offers players a fully-managed Minecraft instance for a monthly fee [4]. Additionally, many smaller companies offer MLGs as a service; an extensive list appears in our technical report [22].

The client (❶) has two main tasks. First, it translates player input into in-game actions, which it speculatively applies to the local state and also sends to the server for validation. The client-server communication uses an implementation-specific protocol (❹) that can be shared between different MLGs. Second, the client visualizes the game state, at a fixed rate.

The server (❷) is responsible for performing all in-game (virtual-world) simulations, maintaining the global state, and disseminating state-updates to clients. Different from simulators in science or engineering, video game simulations tolerate (temporary) inconsistency, and must support modifying the environment via user input. The *game loop* (❸) performs simulations, by applying state-updates to the global state in discrete steps (*ticks*), at a fixed frequency. In MLGs, this frequency is typically set to 20 Hz, or 50 ms per tick. If a tick takes under 50 ms, the MLG waits for the next scheduled tick start. However, if a tick exceeds 50 ms, the tick frequency drops below 20 Hz and the server enters an *overloaded* state. While in this state, *the game fails to meet its QoS requirements* and can cause players to experience game stuttering, visual inconsistency, and increased input latency. Prior work has shown direct causality between increased input latency and reduced player experience [12, 16, 25].

MLGs generate workloads, both data- and compute-intensive, that do not exist in other types of games. In contrast to traditional games, MLGs allow modifications to the terrain. This requires the game server to simulate terrain changes and manage terrain-state alongside the player- and entity-state found in traditional

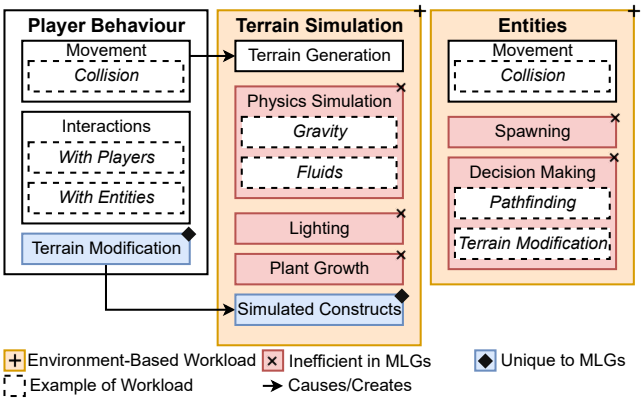


Figure 3: Workload components in MLGs.

games (❺). Unlike other types of state, terrain state can be both *data-intensive* and *compute-intensive*, without direct player input.

2.2 Workloads in MLGs

This section presents our workload model for MLGs, which focuses on players, terrain, and entities. We discuss each of these components, in turn, focusing on unique and challenging aspects. We distinguish novel aspects in our research.

Figure 3 presents a visual overview of our model. Beyond the state-of-the-art, our workload model captures *environment-based workloads*, which are caused by simulating the modifiable environment itself, and scale independently from the number of active players. We argue environment-based workloads are an important part of representative benchmarking workloads for MLGs. However, existing benchmarks do not include this type of workload. Addressing this gap, we propose an MLG workload model which describes a wide range of environment-based workloads. In Figure 3, Terrain Simulation and Entities are examples of environment-based workloads.

2.2.1 Workload from Players (known). Players cause workload for MLGs, and games in general, through their actions. MLGs support player-actions found in traditional games, e.g., player movement and interactions, and also *MLG-specific actions*, e.g., to modify terrain. For player movement, the game computes collisions to prevent players from walking through obstacles such as walls, and disseminates location-changes to other players. Players can also interact with other players and entities (i.e., objects), for example by collecting resources and exchanging them with other players.

An important difference between MLGs and traditional games is support for player-actions that modify the terrain. In MLGs, players can *terraform*—create, modify, and destroy the terrain, as well as the buildings standing on the terrain. This can generate resource-intensive workloads in two ways. First, players can change a large part of the terrain in a short amount of time, for example through the use of explosives. This is both compute- and data-intensive, because the game needs to compute the new terrain, and communicate state updates to keep a consistent view across all players. Second, players can construct dynamic elements such as *simulated constructs*, which increase the complexity of the terrain simulation and are discussed in §2.2.2. The impact of player workloads has been previously examined in both the context of traditional video games architectures and MLGs specifically [35, 47].

2.2.2 Workload from Terrain Simulation (novel). In contrast to traditional games, a significant part of the MLG workload can come from generating and simulating the terrain. MLGs typically present players with an endless open world. This *world* is split into areas, which are lazily generated when players come near them. Once the terrain is generated, the game simulates it and allows players to modify it.

We identify four important components of terrain simulation: physics, lighting, plant growth, and simulated constructs. Although *physics and lighting* simulations are present in traditional games, the modifiable nature of the terrain makes it significantly more challenging to manage such features in MLGs. Unlike static environments, where physics simulation only needs to happen for the relatively few entities that can move through the world, MLGs need to perform physics simulations on the many blocks that compose the terrain itself. For example, a bridge can collapse when a player removes its support pillars, or the terrain underneath them. Once the bridge has collapsed, the bridge no longer casts shadow, so the simulator needs to recompute lighting (frequently) at runtime; static environments do not have this dynamic workload.

Plant growth is an example of a dynamic element unique to MLGs. Plants and trees change over time, reshaping the nearby terrain, thus generating new workload.

Through terrain modification, players can create *simulated constructs*. In a simulated construct, players place together dynamic elements (e.g., plants, automatic croppers) to achieve a certain goal. For example, many players build irrigation systems that grow and harvest vegetables automatically, with high yield. Such systems can leverage tens to hundreds of dynamic elements, whose interaction generates compute-intensive workload for terrain simulation.

In MLGs, as we show in §5, *even a single player can overload the game simulator.* This is in part because, in MLGs, a single player can trigger complex simulations, for example, by building simulated constructs of arbitrary size. By contrast, in traditional games, only the number of concurrent players is strongly correlated with workload intensity.

2.2.3 Workload from Entities (novel). An *entity* is an object that exists in the virtual world but is not a player or terrain. Examples include Non-Playable Characters (NPCs), mobiles (i.e., *mobs*), and items (e.g., a sword). Entities can typically move or be moved by players and collide with each other. Here we describe two important aspects of entity simulation which are challenging for MLGs.

First, games typically instantiate entities at *spawn points*, e.g., spawn an NPC at a spawn point in a dark cave when a player is about to enter. In contrast to static environments, where game developers typically place these spawn points manually, MLGs need to compute spawn points dynamically as terrain modification may obstruct existing spawn points.

Second, NPCs use path-finding algorithms to move around the map. Static worlds pre-compute overlay graphs with viable NPC locations, improving computational efficiency. In contrast, MLGs have changing terrain, so they must compute path-finding graphs dynamically, leading to additional compute-intensive workload.

2.3 Operational Model of MLGs

We detail in this section the game loop used by MLGs. We define the *operational model* as the set of operations, and of events triggering and linking them, of individual components in the implementation of the game loop. Novel, in this work, we analyze the performance implications of the unique aspects of MLG workloads (see §2.2) when executed with the MLG operational model.

In an MLG, the game loop consists of three elements: players, the terrain, and entities. These elements correspond to the workloads specified in §2.2. For each element, its simulation typically requires reading the current game state, and may result in terrain state changes that need to be persisted (i.e., written).

Although, from a performance perspective, it is desirable to run the game loop elements concurrently, there are two challenges with this approach. First, while these elements are in principle independent, they have implicit dependencies through the game state which they access. Individual elements can only run concurrently as long as they do not access the same state. Second, terrain simulation rules can cause a sequence of state changes which cannot be parallelized.

Using our operational model for MLGs, we formulate two implications for MLG performance variability. First, because environment-based workloads do not rely on the presence of players, large environment-based workloads can cause ticks to exceed their maximum duration, even with few or no players connected. Second, because player simulation and environment-based workloads must be completed sequentially when they access the same state, even small environment-based workloads can affect tick duration given they are spatially clustered.

3 METERSTICK BENCHMARK DESIGN

To address contribution **C2**, we design Meterstick, a benchmark for evaluating performance variability in MLGs. The main novelty of Meterstick relates to its workloads (§3.3) and performance metrics (§3.4 and §4).

3.1 System Requirements

Here we describe our eight requirements for Meterstick. We define the first three requirements specifically for our use case. The last five relate to benchmarking computer systems in general, and are based on existing guidelines [28, 48].

- **R1** *Captures performance variability of MLGs:* Meterstick must be capable of capturing relevant performance metrics at a granularity sufficient for analysis of variability. The specific measure of variability must be applicable and meaningful in the context of MLGs.
- **R2** *Validity of workloads:* The workloads used in benchmarking of the MLG should be representative of real-world use and address the workload types listed in §2.2.
- **R3** *Relevant metrics and experiments:* The benchmark should support relevant experiments to isolate different sources of variability, and collect meaningful metrics to allow suitable analysis of these sources.
- **R4** *Fairness:* The benchmark should provide a fair assessment for compatible systems. In particular, bias towards any one system should be limited.

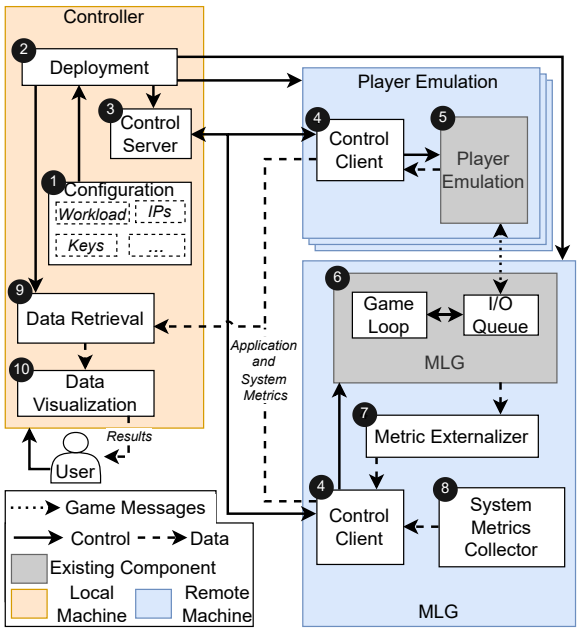


Figure 4: Architecture of Meterstick. Component ❻ is system under test. Component ❺ adapts tool [47].

- **R5** *Ease of Use:* The benchmark should be easy to configure and use with any compatible system.
- **R6** *Clarity:* The benchmark should present results to the user in a way that is suitable for system performance variability.
- **R7** *Portability:* The benchmark should support common deployment environments and be easy to port to others.
- **R8** *Scalability:* Benchmark workloads should be scalable to accommodate benchmarking on increasingly powerful hardware or with more performant systems.

3.2 Design Overview

Here we present the design of Meterstick, our system for benchmarking of performance variability in MLGs. Figure 4 presents Meterstick's high-level design. We discuss the benchmark workloads (addresses **R2**) and metrics (partially addresses **R3**) in more detail in §3.3 and §3.4 respectively.

In our design, the user mainly interacts with Meterstick through its Configuration component (❶). The Configuration allows the user to capture performance variability by specifying the duration and number of iterations of experiments (partially addresses **R1**). The Configuration further allows users to configure benchmark parameters, such as the systems under test and workload, and deployment parameters, such as machine IP addresses (partially addresses **R5**).

After specifying the configuration, the user launches Meterstick. This triggers the Deployment component (❷), which deploys components and software dependencies to remote machines specified in the configuration. This only requires the user needs to specify a set of IP addresses of `ssh`-accessible machines. This makes Meterstick portable (**R7**), and allows users to evaluate MLG performance variability under cloud or self-hosted deployments.

When deployment is complete, the Deployment component hands control to the Control Server (❸). Meterstick follows a Controller/Worker pattern, with the Control Server as the controller, and the Control Clients as the workers (❹). Depending on the configuration, the Control Client runs either *player emulation* or the MLG.

Meterstick uses one or more workers for player emulation (❺). These workers emulate players by connecting the MLG and automatically sending player actions based on programmed behavior. Meterstick implements this by using the player emulation from Yardstick [47], an existing MLG benchmark which we compare to Meterstick in detail in §7.

One worker runs the MLG (❻), which is the system under test. Meterstick captures the MLG's performance variability metrics using the player emulator (❺), the metric externalizer (❼), and the system metrics collector (❽). §3.4 details the operation of these components and the metrics they collect, including our novel metric to capture performance variability.

When the benchmark experiments are done, the Control Server activates the Data Retrieval and Data Visualization components (❾ and ❿), producing basic plots for MLG performance and performance variability. Users can view these plots, and, if desired, provide their own advanced plotting scripts for in-depth analysis (concludes **R5**, **R6**).

3.3 Benchmark Workloads (address R2, R4, R8)

This section presents Meterstick's workloads. Meterstick uses the workload model presented in §2, which divides workloads in three main components: *players*, *terrain simulation*, and *entities*. By using this model, Meterstick's workloads are applicable to MLGs in general, thus avoiding favoring specific systems (partially addresses **R4**). In practice, the user specifies in the Configuration (❶) only the *player* and *terrain simulation* parts of the workload, as *entities* are a result of terrain simulation (spawning, see §2.2).

As future systems may become sufficiently performant to mitigate the impact of Meterstick's workloads, Meterstick supports workload scaling (**R8**). To increase Meterstick's workload complexity, the user can specify an increased number of players to scale the player workload, and use Meterstick's *scale* parameter to select higher-complexity versions of the pre-configured workloads.

While Meterstick supports arbitrary valid Minecraft worlds as workloads, the remainder of this section describes the workloads we design for use in our experiments to highlight performance variability based off our workload model and observable community use (**R2**).

The conceptual challenge of designing the benchmark workloads stems from the vast design space. MLGs give players fine-grained control over the virtual environment, resulting in an endless number of possible world permutations and a large variety in types of simulated constructs.

Additionally, finding evidence to support our selection proved to be challenging, as no peer-reviewed analysis of such artifacts currently exists and game operators do not want to share such information; searching for trustworthy communities and identifying suitable artifacts poses additional challenges. We detail our workload design and the evidence supporting it throughout this section.

Table 1: Minecraft worlds used as workload starting points by the Meterstick benchmark.

Name	Properties	Size [MB]
Control	Freshly generated world	5.4
TNT	Entity actions, terrain updates	6.3
Farm	Resource Farm constructs	26.0
Lag	Complex simulated construct, stress test	4.7

3.3.1 The Environment-Based Workloads. The environment workload is determined by the MLG's terrain generation and the terrain modifications made by players. To obtain representative worlds and workloads, we reconstruct highly popular creations (templates of useful simulated constructs, see §2.2) available on common sharing platforms in the MLG community. Because the MLG community thrives on sharing player-created content with other players, this approach captures essential features of how the community uses these systems.

To cover the range of valid workloads (**R2**), we include two worlds that result in a best-case workload and worst-case workload respectively. During all environment-based workloads, Meterstick connects to the game a single player that performs no actions. This is necessary to correctly capture the response time metric discussed in §3.4. The remainder of this section describes the worlds and their resulting workloads. We list the worlds used in Table 1.

The Control world results in a best-case workload while still being realistic. The Control world is an unmodified world generated by Minecraft version 1.16.4 (seed in the technical report [22]). The measured results of this workload are used as a workload-level baseline to compare the other workloads.

The TNT world contains a 16-by-16-by-14 cuboid filled with TNT blocks which are set to explode around 20 seconds after a player connects. In the systems tested, TNT operates by spawning an entity, which can be interacted with by other entities, including other TNT entities. Thus, when a large section of TNT is activated, the MLG must perform a large number of both entity-collision and physics calculations. Intentionally creating large-scale TNT chain reactions is a popular activity, which can be observed in a plethora of community-made content. For example, a video from 2018 that shows a chain reaction of thousands of TNT blocks has 21 million views [19].

The Farm world contains multiple *resource farms*, which are simulated constructs built by players to automatically generate in-game resources. The specific designs of the simulated constructs in this workload were sourced from popular community creators and each have 1.6 million views on average. A full list is available in the technical report [22]. These farms rely on entities in their functioning, through spawning driven entities and manipulating their pathfinding, or through the creation of passive entities to represent item resources. A core feature of MLGs is collection of resources from the game environment. The ability for players to construct simulated constructs that automate this process is both an intended and common behavior.

Table 2: Metrics collected by Meterstick. The metric type is Derived, Application level, or System level.

Type	Metric	Description
D	Instability Ratio	Tick instability (see §4)
A	Response time	Round trip latency for clients
A	Tick duration	Duration of each tick
A	Tick distribution	Tick time by workload
S	CPU	CPU utilization
S	Memory	Memory usage
S	Threads	Thread total
S	Disk I/O	Bytes read/written
S	Network I/O	Bytes sent/received

The Lag world results in a worst-case workload. This world contains a simulated construct known in the MLG community as a *Lag Machine*. Lag Machines are a specific subset of simulated constructs that are designed to cause high computational load for the MLG, either for the purpose of stress testing it, or to cause it to crash as part of a denial of service attack. The design of the Lag Machine used in this workload is publicly available and provided by a community-creator with 52 thousand subscribers [44]. It is chosen as it operates based on terrain simulation rules. Specifically, it uses many logic-gate constructs in a small area to cause a high volume of simulation rule activations. Importantly, the simulation rules this Lag Machine uses are generally non-malicious, and are used in many resource farm constructs, as well as forming the basis for simulated constructs such as an operational digital Computer [30].

3.3.2 The Player-Based Workload. Meterstick uses a player-based workload facilitated by the player emulation component (❺). In this workload, Meterstick is configured to connect 25 players which move randomly in a 32-by-32 area. The existing Yardstick benchmark [47] focuses solely on the impact of player workload. So, we include this player workload to represent a high-density area in MLGs and allow Meterstick to compare the impact of environment-based workloads with a traditional player-based workload. We select a player count of 25 based on the Minecraft Wiki's dedicated server recommendation [7], as well as the recommendations from various commercial cloud providers (see §5.1).

3.4 Metrics (address R1, R3, R4)

This section describes the application-level and system-level metrics metrics collected by Meterstick, selected to fulfill **R3**. In §4 we describe our novel *Instability Ratio (ISR)* which quantifies performance variability (concludes **R1**). Our ISR metric and all application and system metrics are general to MLGs to avoid bias for specific implementations (concludes **R4**). Table 2 gives an overview of the collected metrics.

3.4.1 Application-level Metrics. Meterstick collects three application level metrics: *response time*, *tick duration*, and *tick distribution*.

Response time is how system latency becomes visible to the user. Lower values are better, and we use existing latency thresholds for the game becoming noticeable and unplayable at 60ms and 116ms respectively [16, 25].

The response time is measured as the time between a player taking an action and the results of that action becoming visible. Meterstick captures this metric by having a player send *chat* messages to all players (including itself), and measuring how long it takes for the player to receive its own message.

While tick duration and tick distribution cannot be directly observed by players, MLGs typically expose these metrics through interfaces commonly used by debugging tools. Meterstick's Metric Externalizer (❼) uses these interfaces to gain access to these metrics without requiring access to the game's source code. As a consequence, Meterstick is easily configured to work with new MLGs.

The *tick duration* is the amount of time it takes the MLG to complete a single iteration of the game loop, and *tick distribution* is the percent of tick time the MLG spent simulating each workload component, such as simulating entities. Both metrics are directly related to game response time and are important indicators of game performance. More detail about the relationship between these metrics is available in §2.1.

3.4.2 System-level Metrics. Meterstick captures system-level metrics to allow users to perform a more in-depth performance analysis. Meterstick collects system-level metrics using the System Metrics Collector (❽), which queries the operating system twice per second.

Meterstick collects CPU utilization, memory usage, the number of operating-system threads associated with the MLG, disk I/O, and network I/O. These metrics allow users to analyze causes of high tick duration, and check for potential performance bottlenecks.

4 INSTABILITY RATIO METRIC

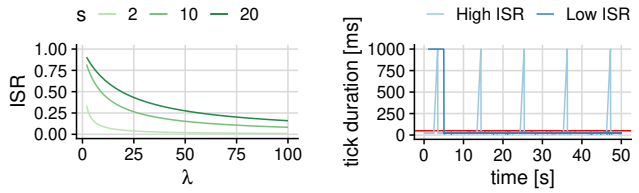
In this section we present our novel Instability Ratio metric. We present a definition, analyze its properties, and compare it to existing metrics.

4.1 Instability Ratio Definition

In the context of online gaming, players prefer stable performance to unstable, but on average faster, performance [11, 37, 40]. Stability facilitates predictability, allowing players to acclimate to game update delays, up to a point. Thus, it is beneficial to quantitatively analyze the stability of game performance by analyzing the variability of game cycles, or ticks (see §2.1). However, existing measures of variability are insufficient, because they do not capture the order of ticks, outlier values, the duration of the trace, or a combination of these elements.

We describe here our novel Instability Ratio (ISR), a normalized metric based on *cycle-to-cycle jitter* [13, 29]. In the context of MLGs, we measure cycle-to-cycle jitter as the difference in delay between consecutive ticks (see §2.1). This delay starts to vary when the game becomes overloaded. We compute the ISR as the normalized sum of MLG cycle-to-cycle jitter. The cycle-to-cycle jitter considers only the difference between two consecutive ticks; reports for this metric include the maximum or moving average value. Novel in this work, our ISR metric sums the differences, and normalizes the result.

The full metric equation is shown in Equation 1, where N_e is the expected number of ticks, t_i is the duration of the i^{th} game tick, b is the delay between ticks when the MLG runs at its intended



(a) Behavior of ISR for varying outlier periods (λ).

(b) Example traces resulting in different ISR values.

Figure 5: Numerical analysis of Instability Ratio. Higher values indicate higher performance variability. s indicates the outlier scaling factor. λ indicates the period between outliers in number of ticks.

frequency, $\max(b, t_i)$ is the period of tick i, and N_a is the actual number of ticks.

When a tick lasts longer than the b value, the proceeding tick is delayed. Thus, if the game meets its performance requirements $N_a = N_e$, but if it becomes overloaded $N_a \leq N_e$ (i.e., $\exists i : t_i > b \implies N_a \leq N_e$).

$$\text{ISR} = \frac{\sum_{i=1}^{N_a} |\max(b, t_i) - \max(b, t_{i-1})|}{N_e \times 2b} \quad (1)$$

Using this metric as a measure of variability, the range of possible values is 0 to 1. A ISR of 0 indicates no variability: the tick period is constant for all ticks in the trace. A ISR of 1 indicates maximum variability, and is reached when the sum of differences between consecutive ticks is equal to twice the duration of the trace (i.e., $N_e \times 2b$). This value is reached when tick periods alternate between their intended value and extremely large values.

4.2 Analysis of ISR Behavior

We analyze the behavior of ISR by modeling a trace where every λ ticks, one tick has a duration of sb, while the others all have duration b. This means 1 in λ ticks exceeds the performance requirement by a factor s. This allows expressing ISR as $ISR = \frac{s-1}{s+\lambda-1}$. A plot of this function and an example trace based on this model are available in Figure 5.

Figure 5a shows how ISR responds to outlier scaling and frequency. The horizontal axis shows λ, which is the number of ticks between outliers, and the vertical axis shows the value of ISR. The curves show the value of ISR for three values of s. The three values of s (2, 10, 20) indicate that all outliers exceed the latency requirement by a factor 2, 10, or 20, respectively.

The plot shows that ISR increases when outliers become larger (increasing s), and when outliers are more common (lower δ). For example, a tick exceeding b by a factor 10 ($s = 10$) every 25 ticks ($\lambda = 25$) results in an ISR value of 0.26.

Figure 5b shows two example traces: High ISR and Low ISR. Both traces contain 1000 ticks. The horizontal axis shows time, and the vertical axis shows tick duration. Most ticks have a duration below 50 ms (b), but each trace has five outliers with a scaling factor of 20, resulting in a 1 second spike. For the Low ISR trace, all outliers are the start of the trace, whereas in the High ISR trace the outliers are evenly distributed over time. While the statistical distributions of the traces are identical, the ISR for the Low ISR trace is 0.009, the ISR for the High ISR trace is 0.15, an order of magnitude higher.

Table 3: Comparison of ISR with existing variability metrics.

Metric	Order Dependent	Irregular Sampling	Normalized
Standard deviation	✗	✗	✗
Allan variance [41]	✓	✗	✗
Jitter [42]	✓	✓	✗
ISR	✓	✓	✓

4.3 Comparing ISR to Alternative Metrics

Table 3 compares ISR to existing measures of variability. Standard deviation captures spread from an average value. It is not order dependent, and thus is not a measure of stability, but dispersion.

Allan variance is used in the field of electrical engineering as a time domain measure of frequency stability, typically applied to clocks or oscillators [41]. Allan variance is order dependent, but relies on a constant sampling frequency and a continuous sampling domain. Neither property is applicable to the duration of tick values.

Jitter is defined in the domain of networking as the smoothed absolute difference between consecutive packets [42]. While most similar to ISR, which is based on cycle-to-cycle jitter, it is not normalized, but rather reported as an average and defined for any packet, rather than an entire sampling duration.

5 REAL-WORLD EXPERIMENTS

To address contribution **C3**, we present here the setup and results from our real-world experiments. We show here the first four **M**ain **F**indings, further results are available in our technical report [22].

MF1 Performance variability can make MLGs unplayable (§5.2). We find that the maximum response time can be up to 20.7 times higher than the arithmetic mean, and exceed by more than a factor of 7.4 the threshold for playable games.

MF2 Environment-based workloads cause significant performance variability (§5.3). We find that environment-based workloads introduce significant performance variability, increasing ISR by 0.04 up to 0.92. This variability can overload popular MLGs by 58 times the normal tick duration and even crash the game.

MF3 MLGs exhibit increased variability in commercial cloud environments compared to self-hosted environments (§5.4). We show that both clouds, AWS and Azure, introduce additional performance variability between iterations of the same workload compared to the local environment, DAS-5. The choice of cloud causes a 1.39 up to 15.44 times increase in ISR IQR and a 1.09 up to 5.61 times increase tick time IQR. The minimum observed ISR for both clouds exceeding the maximum observed ISR on DAS-5.

MF4 The common hardware resource recommendations are insufficient to avoid performance variability (§5.5). The recommended node size exhibits high performance variability and high mean tick duration. Larger node sizes result in lower

Table 4: Hardware recommendations from companies that offer MLGs as a service. *NP* means information is not provided to consumers, *V* means time-varying. Full table, with references to providers, available in the tech report [22].

Service	RAM [GB]	vCPU [#]	CPU Speed [GHz]
Server.pro	4	2	2.4
Skynode	4	2	3.6
Hostinger	3	3	NP
Ferox Hosting	4	NP	NP
MelonCube	4	NP	3.4
Azure	4	2	V
AWS	1	1	V

values of both, such that a node with 8 vCPUs reduces performance variability and mean tick duration to acceptable levels.

5.1 Experimental Setup

In this section we describe our experimental setup. In our experiments, we evaluate three MLGs, i.e., the systems under test, in three different environments.

5.1.1 System under test. We use in our experiments three MLGs that use the Minecraft protocol: the original Minecraft as developed by Mojang [1], *Forge*, and *PaperMC*. We select these services because of their popularity and utility.

Forge is the most popular MLG for operating modified (i.e., *modded*) services [3]. Of the top-50 most downloaded Minecraft mods, 45 work exclusively with Forge. Of the 5 mods that are not exclusive to Forge, only one is incompatible with it [5]. *PaperMC* is marketed as a high-performance alternative to Minecraft [6]. While the PaperMC project does not quantify its performance improvement over Minecraft, it does provide documentation of its optimizations, which include extensive changes to threading models and virtual environment processing.

5.1.2 Deployment Environment. We evaluate the MLGs in two commercial cloud environments, Amazon AWS and Microsoft Azure, and DAS-5, a supercomputer for academic and educational use [9]. We choose AWS and Azure because they are the two cloud environments with the biggest market share, with 32% and 20% respectively [38]. We use DAS-5 to evaluate how commercial cloud environments affect the performance variability of MLGs, compared to self-hosting these games on dedicated hardware.

Our experiments on cloud environments use *T3.Large* and *Standard_D2_v3* nodes respectively. Both node types are equipped with 2 vCPUs and 8 GB memory. We choose these nodes based on the default hardware configurations recommended by Minecraft service providers as well as guidelines published by AWS and Azure [8, 33].

For an overview of these hardware recommendations, see our technical report [22].

On DAS-5, we use a regular node, which is equipped with a dual 8-core 2.4 GHz processor and 64 GB memory, and limit the number of CPU cores available to the MLG by setting its CPU affinity to two cores, unless indicated otherwise. Because the MLGs used in our experiments run on the Java Virtual Machine (JVM), we limit memory available to the MLG in both cloud an DAS-5 nodes by setting the JVM's maximum heap size to 4 GB.

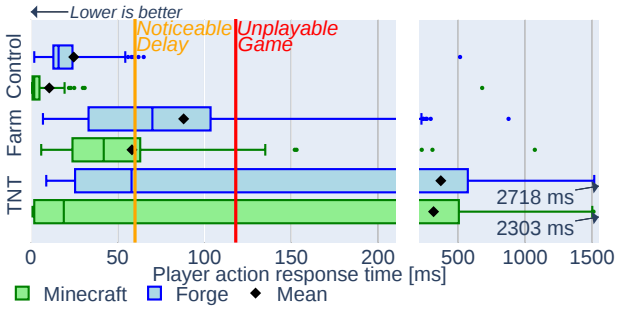


Figure 6: Game response time in AWS environment when running separate environment-based workloads. Whiskers indicate 5th and 95th percentile respectively. The black diamonds indicate arithmetic mean.

Table 4 lists a sampling of hardware recommendations from commercial cloud companies that offer Minecraft-like game hosting. If no plan was marked "recommended," data is taken from plans that are comparable to recommended plan on other services. From these recommendations we find that 2 vCPU and 4 GB RAM is the most common configuration. On AWS and Azure it is possible to select specific configurations of hardware, however, to ensure that the Benchmark metric tools have sufficient memory during the experimental duration, we use nodes with at least 8 GB RAM and limit the heap memory of the Minecraft-like game to 4 GB using the `-Xmx` JVM argument in all experiments.

5.2 MF1: Performance variability can make MLGs unplayable

Due to significant performance variability, the median and mean game response times give an optimistic view of game performance, and is worse than the performance observed by players. Figure 6 depicts the result, and shows that the 95th percentile of game response time can be up to 4.1 times higher than the arithmetic mean, and exceed by more than a factor of 12.8 the threshold that makes the game unplayable. In real-time games, a temporary spike in delay can significantly affect the user's experience, similar to a temporary freeze in a phone call or video stream.

Figure 6 shows the response time (horizontal axis) for two MLGs (Minecraft in green, and Forge in blue) under three different workloads (vertical axis). The workloads and response time metric are described in §3.3 and §3.4.1 respectively. The whiskers extend to the 5th and 9th percentiles, respectively,

and the black diamond indicates the arithmetic mean. The *Noticeable Delay* line (at 60 ms, in orange) and *Unplayable Game* line (at 118 ms, in red) indicate high game-latency which respectively marks the values where latency becomes noticeable to players and makes the game so unresponsive it becomes unplayable [16, 25].

Under the Control workload (top two boxes), the 95th percentile is below the noticeable threshold for both Minecraft and Forge. However, the maximum value for Forge (514 ms) is 20.7 times larger

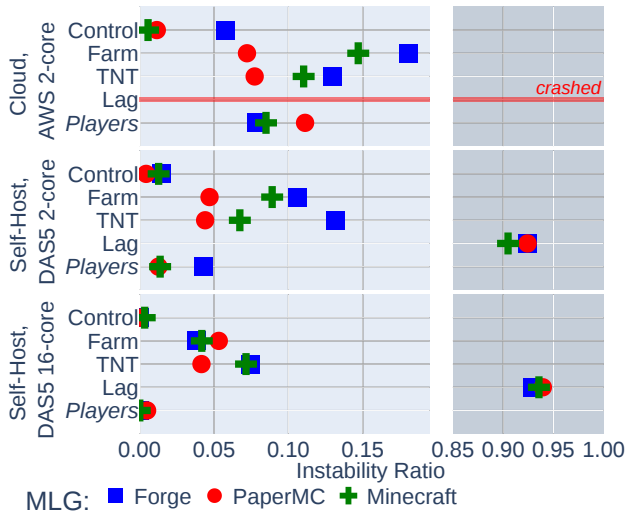


Figure 7: ISR for each MLG on the AWS and DAS-5 environments. The Lag workload crashed all MLGs on AWS; see text for explanation. §3.3.2 defines the "Players" workload.

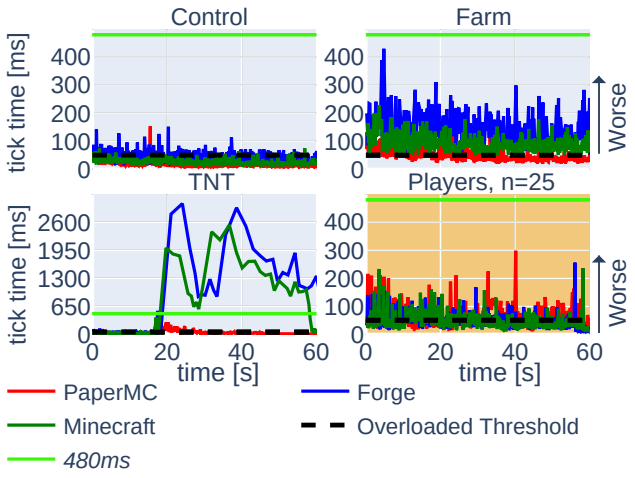


Figure 8: Tick time over time for each MLG in the AWS environments running the Control, Farm, TNT and Players workloads. The Lag workload on AWS is omitted as each MLG crashes. §3.3.2 defines the "Players" workload.

than the mean, and the maximum value for Minecraft (679 ms) is exceeds by 7.4 times the Unplayable threshold at 118 ms. These outliers occur directly after a player connects to the game. This means that, even with good average performance, the game can still be unplayable if players frequently connect, which is a common occurrence in online multiplayer games.

Compared to the Control workload, the Farm and TNT workloads show significantly more performance variability, pointing to a further degradation of player experience. *In all cases, the mean and median values give an overly optimistic view of the game's performance.* For the Farm workload, the mean and median values for Forge (third bar from the top) indicate the response time is noticeable, but not unplayable. However, the plot shows a 95th percentile of 225.8 ms, which is 1.9 times as high as the Unplayable threshold. For Minecraft (fourth bar from the top), the mean and median values indicate that the response time is not noticeable for players. However, the plot shows that performance variability causes the response time to exceed the Noticeable threshold more than 25% of the time (box's right edge exceeds the Noticeable threshold), and exceeds the Unplayable threshold more than 5% of the time. The TNT workload causes the highest performance variability for both Forge and Minecraft (bottom two boxes, 547 ms interquartile range for Forge and 503 ms for Minecraft). In both cases, the median response time is below the noticeable threshold, while the 95th percentiles are 12.7 times the unplayable threshold, and the maximum observed values (indicated with black arrows) are at least 19 times larger than the unplayable threshold.

From results in this section, we conclude that the mean and median values give an overly optimistic view of MLG performance, and that performance variability in MLGs results in noticeable and unplayable game latency, impacting players.

5.3 MF2: Environment-based workloads cause significant performance variability

Environment-based workloads cause significantly increased performance variability on each game and in each environment tested, and can overload or crash the game. Figure 7 shows the performance variability of each MLG when running environment-based workloads on AWS and DAS-5. Compared to the control workload, each MLG on each environment exhibits higher performance variability when operating environment-based workloads.

Figure 7 shows performance variability, quantified using ISR (see Equation 1). The three top-level rows show three environment configurations, each containing five workloads. The color and shape of the marker indicate one of three MLGs.

Environment-based workloads (i.e., Farm, TNT, Lag) cause significantly higher performance variability than the player workload and control workload for all games in all environments, with the exception of PaperMC on AWS (red circles in the top row). This provides evidence that environment-based workloads cause significant performance variability. Further analysis into the behavior of PaperMC reveals that it contains performance optimizations specifically for handling TNT explosions, improving its performance on the TNT workload, and Redstone, a simulated block type which is used in the Farm workload (analysis of PaperMC given in technical report [22]). This provides evidence that the performance variability caused by these environment-based workloads are known to the MLG community and can (at least partially) be addressed through engineering.

Of all workloads, the Lag workload causes the most performance variability. Further analysis reveals that this happens because this workload consists mainly of parts which are only simulated every other tick, causing the game to alternate between extremely short and extremely long ticks. This maximizes the value of ISR, which is based on the difference in duration between consecutive ticks.

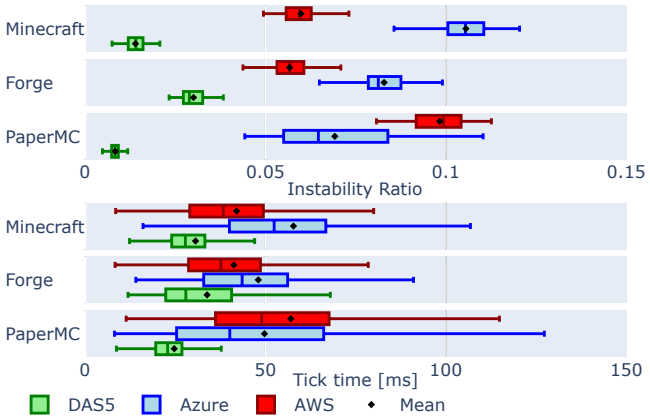


Figure 9: Distribution of tick times and ISR from 50 iterations of the Players workload. Whiskers extend to ± 1.5 × IQR, bounded by the minimum and maximum values.

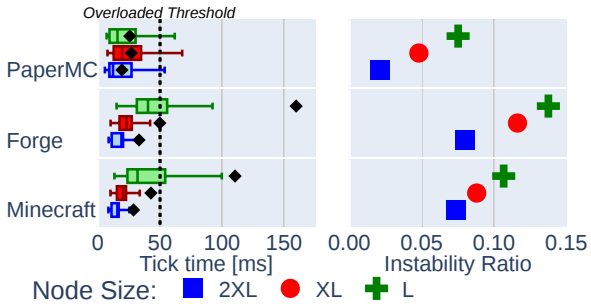


Figure 10: Tick time distribution and ISR during TNT workload on various AWS node sizes. Whiskers extend to ± 1.5 × IQR, bounded by the minimum and maximum values. Black diamond indicates arithmetic mean.

There are no results for running the Lag workload on AWS because all three MLGs crash when a player joins and the environment simulation begins. The corresponding increase in tick duration causes the player's connection to time-out, forcing each MLG to stop.

Figure 8 shows the game's tick duration over time for each game when running on AWS. The dashed black line indicates the overloaded threshold at 50 ms, and the green line allows calibrating the vertical axis across the four sub-plots.

The stability observed when running the Control workload in Figure 7 is visible in the top-left sub-plot in Figure 8 as three relatively stable curves with few spikes. In contrast, the high performance variability observed for the Farm and TNT workloads is visible in the top-right and bottom-left sub-plots as jittery curves. The Farm workload depicted in the top-right shows curves which change value at high frequency, resulting in high ISR. PaperMC's tick durations are frequently below the 50 ms threshold, resulting in lower ISR. The TNT workload depicted in the bottom-left shows curves which change value at a much lower frequency, but reach significantly higher values, exceeding 2500 ms for both Minecraft and Forge. Similar to the Farm workload, PaperMC's tick durations are often below 50 ms, resulting in lower ISR.

5.4 MF3: MLGs exhibit increased variability in commercial cloud environments

In our experiments, all MLGs show increased performance variability in terms of both variability (i.e. ISR) and tick times, when run on the AWS and Azure cloud environments, compared to the self-hosted DAS-5. Figure 9 shows ISR and tick time distribution across 50 iterations of the Player workload (see §3.3.2) of all three games (on the vertical axis) in DAS-5 (green), Azure (blue), and AWS (red).

The results show that all three games are the most stable, with the lowest median ISR (line inside boxes) and the lowest ISR overall, when run on DAS-5. The *maximum* ISR observed on the DAS-5 is 0.021 (Forge), which is smaller than 0.029, the *minimum* ISR observed in AWS and Azure (PaperMC). Distribution of tick time follows a similar trend, with each game exhibiting lowest mean and median tick time on the DAS-5, as well as the smallest interquartile range (IQR).

From this result, we highlight two surprising observations. First, no game performs best in all environments. On DAS-5, PaperMC performs best, slightly outperforming Minecraft with a median ISR of 0.007 and 0.010 respectively. Although PaperMC also has the lowest median ISR on Azure, it simultaneously has the highest IQR of both ISR, 0.028 compared to Forge's 0.009 and Minecraft's 0.011, and tick time, 40.75 to Forge's 23.25 and Minecraft's 26.71. Moreover, on AWS, PaperMC is the worst performing game, with a median ISR of 0.094 and a median tick time of 48.98. Second, neither cloud performs best for all games. While AWS performs better for Minecraft and Forge, Azure performs best for PaperMC.

Increased performance variability in commercial cloud environments is a well-documented phenomenon [10, 32, 43, 46], with a wide variety of sources identified for the cause of increased variability, including hardware manufacturing differences, shared tenancy of hardware and networks, specific software configurations, and resource allocation and scheduling systems. With so many variables operating in the context of commercial cloud hosting, it is infeasible to identify a single source responsible for the variability of these games, especially since commercial cloud hosting companies do not make internal data on resource allocation and shared tenancy publicly available. However, we can conclude that this variability observably impacts the performance of MLGs, and can be compared between MLGs and commercial cloud services.

5.5 MF4: Using recommended hardware results in significant performance variability

Recommended hardware configurations in cloud environments result in unacceptable levels of performance variability, which degrades player experience. By using more powerful cloud hardware, performance variability can be limited to acceptable levels. Figure 10 shows this result, showing both the mean tick duration and ISR for varying VM sizes in AWS. We use the notation *2XL*, *XL*, *L* to denote AWS VM sizes *t3.large*, *t3.xlarge*, and *t3.2xlarge* respectively.

Companies that specialize in cloud hosting of MLGs commonly list recommended hardware configurations, with the most frequent recommendation being 2 vCPUs and 4 GB memory. An overview of these recommendations is available in Table 4. These recommended

values are significantly lower than those listed on the community-driven Minecraft wiki, which recommends a dedicated full CPU (e.g., Intel i5 or i7, or AMD Ryzen 5 or 7) and 6 GB memory [7]. This indicates that players experienced performance problems with the recommended hardware configuration.

Figure 10 shows that using the recommended hardware configuration as listed by cloud-hosting companies, which corresponds to the *L* node type, results in poor performance and significant performance variability. On this node size, each MLG becomes significantly overloaded by environment-based workloads and exhibits high performance variability.

The larger node types *XL* and *2XL* have 4 and 8 vCPUs respectively [2]. While *XL* provides better performance and less performance variability than *L*, it remains insufficient to keep the mean tick time below 50 ms. The *2XL* node type is required to provide sufficiently low mean tick duration. However, this node type still shows significant performance variability for Minecraft (green cross) and Forge (blue square), which means these games can still become overloaded temporarily, as shown in §5.3.

Interestingly, we observe that the benefit of more powerful hardware varies per MLG. Specifically, while PaperMC's (red circle) performance instability (i.e., ISR) increases significantly when decreasing hardware resources, from 0.025 to 0.08 in the top-right sub-plot, it is the only game whose mean and 75th percentile tick duration stays well below the 50 ms threshold. Further analysis shows that, while PaperMC becomes overloaded and its tick duration exceeds 50 ms, the number of extreme outliers is low, preventing this performance problem from becoming visible in the mean tick duration.

6 ACTIONABLE INSIGHTS AND LIMITATIONS

The main findings in §5 lead to *actionable insights*:

I1 Game developers and hardware producers should report performance variability when evaluating the performance of online games, using measures of variance such as Instability Ratio (see §3.4) and the distribution of game response time and frames per second (FPS). Games must provide consistently good performance to their users. Our experiments show that MLGs can be overloaded and become unplayable, even when mean and median performance values indicate good performance (**MF1**, Figure 6).

I2 Game developers and hardware producers should include environment based workloads in their benchmarks for MLGs. It is not sufficient to evaluate the performance of MLGs using only large numbers of players (i.e., player-based workloads). Environment-based workloads cause significant performance variability in MLGs and make them unplayable (**MF2**, Figure 7), and must therefore be included in MLG benchmarks.

I3 Players should choose their cloud environment depending on their MLG, and should consider self-hosting their game. Our results indicate that choice of best cloud provider depends on the MLG. Minecraft and Forge obtain lower performance variability on AWS, while PaperMC obtains lower performance variability on Azure (**MF3**, Figure 9). Moreover, self-hosting remains a valuable alternative, resulting in significantly lower performance variability overall.

I4 MLG service providers should increase their hardware recommendations. For full findings relating to hardware recommendations see the technical report [22].
Similarly, users who seek to avoid adverse performance variability when operating MLG cloud environments should choose node sizes comparable to the 8 core t3.2xlarge node, or use our benchmark to compare both various cloud providers and the specific MLG implementations.

I5 Game developers should engineer MLGs to reduce impact of environment-based workloads. See technical report [22].

Here we discuss limitations of our work related to the Instability Ratio metric and the workloads. ISR cannot be used as a singular performance metric, but rather is designed as an additional axis by which to quantitatively appraise the performance of a game server, by capturing behavior that other metrics cannot. Because ISR is a measure of variability over an entire sampling duration, it does not capture extremely poor but stable performance, or the occurrence of singular relatively small outliers. Thus, other measures, such as percentiles, are necessary to observe the magnitude of tick durations and detect lone outliers. It is not currently understood how ISR directly relates to player-perceived quality of experience and quality of experience, and should be explored in future work, for instance through player studies.

The workloads included in Meterstick are intended to cover a wide range of realistic environment-based workloads, from common to extreme cases. However, there is no publicly available analysis of environment-based workload prevalence. Thus, we select environment-based workloads using proxy metrics such as total views and downloads in online MLG communities, which may not be representative of all MLG players. Additionally, our experiments measuring the impact of environment-based workloads utilized a purposefully minimal player-based workload component. Finally, because our experiments focused on environment-based workloads, our player-based workloads ("Players") uses random avatar movement. Although real player behavior is likely more complex, no player-behavior models exists for MLGs.

7 RELATED WORK

We summarize in this section a developing overview of related work. Overall, this study is the first to evaluate performance variability in MLGs. This is challenging because there is neither a generally accepted set of relevant workloads for MLGs, nor a standardized metric to quantify performance variability in computer systems.

Closest to our work, Yardstick is an MLG benchmark used to show the limited scalability of MLGs [47]. The authors use Yardstick to evaluate the scalability and network characteristics of several MLG services. However, Yardstick does not quantify performance variability, resulting in optimistic results. Moreover, the authors do not evaluate MLG performance under environment-based workloads or in the cloud.

The MineRL competition [26] provides a dataset of Minecraft player recordings. This dataset provides demonstrations to train artificial intelligence systems to complete a challenging in-game task. In contrast, the workloads used in this work focus on commonly observed patterns in the MLG community.

There exist several systems that aim to improve the scalability of MLGs. Manycraft [15] increases the maximum number of players in a Minecraft instance by using Kiwano. Kiwano [14] allows horizontal scaling of virtual environments through Voronoi partitioning, but requires a *static environment*, which disables the MLG's modifiable world and is incompatible with environment-based workloads. Similar in many ways to Manycraft, Koekepan [23] uses zone-partitioning and scales horizontally. Dyconits [17] are a distributed architecture that scales MLGs vertically, through the use of dynamically managed consistency-units. None of these approaches considers explicitly performance variability.

Iosup et al. [27] find that commercial cloud environments exhibit significant yearly and daily performance variability patterns. The authors show that performance variability varies per cloud operator, and use simulation experiments to show that this can affect the performance of applications, including a social online game. In contrast, our benchmark uses real-world experiments to evaluate the effect of performance variability on MLGs, which are real-time online games.

8 CONCLUSION

Online gaming is a popular and lucrative part of the entertainment industry, but raises important performance challenges. In this work, we posit performance variability is an important cause for the lack of scalability in MLGs.

We make a four-fold contribution to better understand the behavior of MLGs. *First*, we propose a novel workload model for these systems, which identifies important sources of performance variability not considered elsewhere. *Second*, we design and implement Meterstick, the first benchmark to evaluate performance variability in MLGs. Meterstick uses realistic workload types; novel, it considers environment-based workloads, and can evaluate MLGs running both in self-hosted and cloud environments such as Amazon AWS and Microsoft Azure. *Third*, we use Meterstick to perform real-world experiments and analyze the results. We find that performance variability negatively affects players in MLGs, that both environment-based workloads and cloud environments can cause significant performance variability. This leads us to formulate four actionable insights. *Fourth*, we release FAIR and FOSS artifacts that enable reproducibility for this work.

In future work, we aim to conduct user studies to directly link our Instability Ratio (ISR) values to player-perceived quality of experience. To encourage community adoption, we aim to create a public score-board where operators of MLG-as-a-service can publish benchmark scores.

ACKNOWLEDGMENTS

This work is supported by the NWO grant OffSense, and by structural funds from VU Amsterdam.

REFERENCES

[1] 2020. Minecraft Server Download. https://www.minecraft.net/en-us/download/server [accessed Oct. 2021].
[2] 2021. Amazon EC2 T3 Instances – Amazon Web Services (AWS). https://aws.amazon.com/ec2/instance-types/t3 [accessed Oct. 2021].
[3] 2021. Minecraft Forge downloads. https://files.minecraftforge.net/net/ [accessed Oct. 2021].
[4] 2021. Minecraft Realms for Java. https://www.minecraft.net/en-us/realms-for-java [accessed Oct. 2021].
[5] 2021. Mods - Minecraft - CurseForge. https://www.curseforge.com/minecraft/mc-mods [accessed Oct. 2021].
[6] 2021. PaperMC – The High Performance Fork. https://papermc.io [accessed Oct. 2021].
[7] 2021. Server/Requirements/Dedicated. https://minecraft.fandom.com/wiki/Server/Requirements/Dedicated [accessed Oct. 2021].
[8] Amazon. 2021. Run your own Minecraft Server. https://web.archive.org/web/20201126074839/https://aws.amazon.com/getting-started/hands-on/run-your-own-minecraft-server/ [accessed Dec. 2021].
[9] Henri E. Bal, Dick H. J. Epema, Cees de Laat, Rob van Nieuwpoort, John W. Romein, Frank J. Seinstra, Cees Snoek, and Harry A. G. Wijshoff. 2016. A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term. *Computer* 49, 5 (2016), 54–63.
[10] Zhen Cao, Vasily Tarasov, Hari Prasath Raman, Dean Hildebrand, and Erez Zadok. 2017. On the Performance Variation in Modern Storage Stacks. In *FAST*. 329–344.
[11] Kuan-Ta Chen, Polly Huang, and Chin-Laung Lei. 2006. How sensitive are online gamers to network quality? *Commun. ACM* 49, 11 (2006), 34–38.
[12] Mark Claypool and Kajal T. Claypool. 2006. Latency and player actions in online games. *Commun. ACM* 49, 11 (2006), 40–45.
[13] N. Da Dalt and A. Sheikholeslami. 2018. *Understanding Jitter and Phase Noise*. Cambridge University Press. 15–37 pages.
[14] Raluca Diaconu and Joaquín Keller. 2013. Kiwano: A scalable distributed infrastructure for virtual worlds. In *HPCS*. 664–667.
[15] Raluca Diaconu, Joaquín Keller, and Mathieu Valero. 2013. Manycraft: Scaling Minecraft to Millions. In *NetGames*. 1:1–1:6.
[16] Matthias Dick, Oliver Wellnitz, and Lars C. Wolf. 2005. Analysis of factors affecting players' performance and perception in multiplayer games. In *NetGames*. 1–7.
[17] Jesse Donkervliet, Jim Cuijpers, and Alexandru Iosup. 2021. Dyconits: Scaling Minecraft-like Services through Dynamically Managed Inconsistency. In *ICDCS*.
[18] Jesse Donkervliet, Animesh Trivedi, and Alexandru Iosup. 2020. Towards Supporting Millions of Users in Modifiable Virtual Environments by Redesigning Minecraft-Like Games as Serverless Systems. In *HotCloud*.
[19] DudeItsRocky. 2018. "Huge Minecraft Tnt World Explosion With Aftermath" (Minecraft TNT Explosion, Minecraft Explosion). https://www.youtube.com/watch?v=9ZbenrBO_wI [accessed Oct. 2021].
[20] Jerrit Eickhoff. 2021. Meterstick. https://github.com/atlarge-research/Meterstick
[21] Jerrit Eickhoff, Jesse Donkervliet, and Alexandru Iosup. 2021. *Meterstick Benchmark: Source, Documentation and Data*. https://doi.org/10.5281/zenodo.5567720
[22] Jerrit Eickhoff, Jesse Donkervliet, and Alexandru Iosup. 2023. Meterstick: Benchmarking Performance Variability in Cloud and Self-hosted Minecraft-like Games Extended Technical Report. *CoRR* abs/2112.06963v2 (2023). arXiv:2112.06963v2 https://arxiv.org/abs/2112.06963v2
[23] Herman Arnold Engelbrecht and Gregor Schiele. 2013. Koekepan: Minecraft as a Research Platform. In *NetGames*. 16:1–16:3.
[24] Epic Games. 2021. Dynamic Resolution | Unreal Engine Documentation. https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/DynamicResolution/ [accessed Dec. 2021].
[25] Valentin Forch, Thomas Franke, Nadine Rauh, and Josef F. Krems. 2017. Are 100 ms Fast Enough? Characterizing Latency Perception Thresholds in Mouse-Based Interaction. In *EPCE*, Vol. 10276. 45–56.
[26] William H. Guss, Brandon Houghton, Nicholay Topin, Phillip Wang, Cayden Codel, Manuela Veloso, and Ruslan Salakhutdinov. 2019. MineRL: A Large-Scale Dataset of Minecraft Demonstrations. *CoRR* abs/1907.13440 (2019).
[27] Alexandru Iosup, Nezih Yigitbasi, and Dick H. J. Epema. 2011. On the Performance Variability of Production Cloud Services. In *CCGrid*. 104–113.
[28] Raj Jain. 1991. *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling*. Wiley.
[29] David Lee. 2002. Analysis of Jitter in Phase-Locked Loops. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 49 (2002).
[30] legomasta99. 2018. [Minecraft Computer Engineering] - Quad-Core Redstone Computer v5.0 [12k sub special!]. https://www.youtube.com/watch?v=SbO0tqH8f5I [accessed Oct. 2021].
[31] Asher Madan. 2019. Minecraft (likely) becomes the best-selling game of all time on its 10th birthday. https://www.windowscentral.com/minecraft-becomes-best-selling-game-all-time-its-10th-birthday [accessed Sep. 2021].
[32] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, Robert Ricci, and Ana Klimovic. 2018. Taming Performance Variability. In *OSDI*. 409–425.
[33] Microsoft. 2021. Basic Game Server Hosting on Azure. https://docs.microsoft.com/en-us/gaming/azure/reference-architectures/multiplayer-basic-game-server-hosting [accessed Dec. 2021].
[34] Minetrack. 2021. Minetrack. https://minetrack.me/ [accessed Oct. 2021].
[35] Vlad Nae, Alexandru Iosup, and Radu Prodan. 2011. Dynamic Resource Provisioning in Massively Multiplayer Online Games. *TPDS* 22, 3 (2011), 380–395.
[36] Newzoo. 2021. Newzoo Global Games Market Report 2021 | Free Version | Newzoo. https://newzoo.com/insights/trend-reports/newzoo-global-games-

market-report-2021-free-version [accessed Oct. 2021].
[37] Aline Normoyle, Gina Guerrero, and Sophie Jörg. 2014. Player perception of delays and jitter in character responsiveness. In *Proceedings of the ACM Symposium on Applied Perception*. 117–124.
[38] Felix Richter. 2021. Infographic: Amazon Leads $150-Billion Cloud Market. https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers [accessed Oct. 2021].
[39] Felix Richter. 2021. Infographic: Gaming: The Most Lucrative Entertainment Industry By Far. https://www.statista.com/chart/22392/global-revenue-of-selected-entertainment-industry-sectors [accessed Sep. 2021].
[40] Michal Ries, Philipp Svoboda, and Markus Rupp. 2008. Empirical study of subjective quality for massive multiplayer games. In *2008 15th International Conference on Systems, Signals and Image Processing*. IEEE, 181–184.
[41] William J Riley. 2008. Handbook of frequency stability analysis. (2008).
[42] Henning Schulzrinne, Steven Casner, R Frederick, and Van Jacobson. 2003. RFC3550: RTP: A transport protocol for real-time applications.
[43] Javid Taheri, Albert Y. Zomaya, and Andreas Kassler. 2017. vmBBProfiler: a black-box profiling approach to quantify sensitivity of virtual machines to shared cloud resources. *Computing* 99, 12 (2017), 1149–1177.
[44] timeam. 2020. The Most Efficient Minecraft Lag Machine. https://www.youtube.com/watch?v=QI0zdI4mDcA [accessed Oct. 2021].
[45] Unity. 2021. Unity - Manual: Dynamic resolution. https://docs.unity3d.com/Manual/DynamicResolution.html [accessed Feb. 2023].
[46] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan S. Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. 2020. Is Big Data Performance Reproducible in Modern Cloud Networks?. In *NSDI*. 513–527.
[47] Jerom van der Sar, Jesse Donkervliet, and Alexandru Iosup. 2019. Yardstick: A Benchmark for Minecraft-like Services. In *ICPE*. ACM, 243–253.
[48] Reinhold Weicker. 2002. Benchmarking. In *Performance Evaluation of Complex Systems: Techniques and Tools, Performance 2002, Tutorial Lectures*, Vol. 2459. 179–207.
[49] Wilkinson et al. 2016. The FAIR Guiding Principles for scientific data management and stewardship. *Nature SciData* 3 (2016).