Vrije Universiteit Amsterdam

Bachelor Thesis

Design and Implementation of a Player-Behavior Tracing System for MVEs

Author: Erik Doytchinov (2760495)

1st supervisor: Jesse Donkervliet
2nd reader: Alexandru Iosup

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

August 25, 2025

Abstract

Modifiable Virtual Environments (MVEs) like Minecraft present a unique challenge for performance and player behavior monitoring due to player-driven and dynamic workload patterns. Traditional tools are insufficient in providing detailed player behaviors or fine-grained runtime metrics that are necessary to build rigorous scientific analyses on MVEs. This thesis introduces TraceCraft, a tracing mod developed and designed for Minecraft, specifically aimed at comprehensive runtime performance and behavioral tracing. TraceCraft captures detailed subsystem-level metrics, including system resources, simulation loop timings, event-driven generation, game state management, and application-level resource usage. The primary advantage is the focus on player-centric metrics which are gathered to quantify player interactions such as movement paths, block placements, combat actions, and idle times. Metrics are designed to be asynchronously collected and with minimal overhead and sent to an external database for further analysis. The results highlight TraceCraft's effectiveness as both a practical tool for MVE performance monitoring and as a robust platform for further academic research into performance optimization and player behavior modeling.

Contents

1 Introduction **1**

1.1 Context 2

1.2 Problem Statement 2

1.3 Research Questions 3

1.4 Thesis Contributions 4

2 Background **7**

2.1 Modifiable Virtual Environments 7

2.2 Telemetry and System Monitoring in Distributed Systems 8

2.3 Time-Series Databases and Data Pipelines 8

2.4 Game Engine Instrumentation Frameworks 9

3 Design of TraceCraft **11**

3.1 Design Requirements 11

3.2 Design Overview of TraceCraft 12

3.3 Design Alternatives 13

3.4 Tracing Player Behavior in MVEs 16

3.5 How TraceCraft Meets the Design Requirements 17

4 Implementation of TraceCraft **21**

4.1 Implementation Overview 21

4.2 Client-side Implementation 23

4.3 Server-side Implementation 25

4.4 Data Handling and Performance Impact Considerations 28

5 TraceCraft Implementation Evaluation **33**

5.1 Main Findings 33

5.2 Experimental Setup 33

5.3 Player-Tracing Validation Experiments 34
5.4 Performance Overhead Analysis 36
5.5 Comparative Mod Evaluation 39

6 Player Behavior Data Collection and Analysis 43
6.1 Telemetry Processing and Feature Design 43
6.2 Statistical Modeling of Movement and Idle Patterns 45
6.3 Player Archetypes 46
6.4 Implications for Player Modeling and Mod Design 47

7 Related Work 49
7.1 Existing Minecraft Performance Tools 49
7.2 Comparative Analysis of Metrics Collection Tools 50
7.3 Tracing and Instrumentation Techniques 50
7.4 Related Scientific Publications 51
7.5 Gaps Identified 52

8 Conclusion 55
8.1 Answering Research Questions 55
8.2 Limitations and Future Work 56

References 57

A Reproducibility 63
A.1 Abstract 63
A.2 Artifact check-list (meta-information) 63
A.3 How to access 64
A.4 Evaluation and expected results 65
A.5 Notes 65

B Self Reflection 67

1

Introduction

Video game environments have transformed how individuals engage in collaborative learning, creative expression, and social interaction. Games such as Minecraft, Roblox, and Second Life have altered traditional entertainment boundaries to become sophisticated platforms where users construct, program, and distribute their own interactive experiences. These virtual environments serve as valuable research artifacts for investigating system performance, user behavior patterns, and large-scale resource management challenges. Modifiable Virtual Environments (MVEs) games that expose their underlying architecture through modification frameworks or scripting interfaces provide researchers with opportunities to observe how human activity generates computational stress patterns that synthetic benchmarks cannot replicate (1, 2).

Systematic observation of MVEs presents significant challenges; production servers must simultaneously manage physics simulations, network protocols, AI systems, and rendering pipelines within strict timing constraints, all while responding to unpredictable, player-generated workloads. Conventional profiling tools typically provide aggregate CPU statistics or memory snapshots but lack the granularity necessary to correlate performance anomalies with specific player behaviors (3). Consequently, it often resorts to trial-and-error approaches for performance diagnosis and encounters substantial obstacles in obtaining the fine-grained trace data essential for reproducible experimentation.

The significance of this work extends beyond technical implementation. By enabling systematic collection of both performance and behavioral data, it provides researchers with insights into how human actions influence system behavior in modifiable environments. This capability supports advances in performance optimization, player behavior modeling, and the development of more responsive virtual environments that can adapt to user patterns.

1.1 Context

Modifiable Virtual Environments occupy a unique position within the domains of interactive system design, distributed computing, and user-centered experimental platforms. Unlike video games with fixed functionality, MVEs explicitly encourage users to modify the underlying simulation through scripting, modification installation, and automated system construction (4). These modifications can substantially alter computational load characteristics in ways that the original engine architecture did not anticipate. For instance, a server configuration that adequately supports fifty concurrent users may experience performance degradation when a single user introduces high-frequency block updates.

From a research perspective, this adaptability presents both opportunities and challenges. The malleability of MVEs enables controlled experimental studies where researchers can modify environmental parameters, replay behavioral traces, and establish causal relationships that reflect authentic usage patterns. Conversely, this same flexibility generates workload characteristics that are not feasible with static analysis. Effective instrumentation must therefore capture both system-level performance indicators, such as including tick timing, chunk generation processes, and network packet queues, and at the same time behavioral-level signals encompassing player movement patterns, block interaction frequencies, and combat engagement metrics at a sufficient level to preserve event visibility (5).

By integrating systems observability principles with game modification frameworks, this research establishes foundational infrastructure for enhanced analytics capabilities in MVEs. Subsequent chapters provide detailed examination of the design decisions, implementation challenges, and empirical validation that demonstrate TraceCraft's effectiveness across both synthetic stress testing scenarios and authentic multiplayer gaming sessions.

1.2 Problem Statement

Despite the growing academic interest in MVEs game environment, there hasn't been a carefully studied and designed, lightweight, in-game tracing solution specifically made for the dynamic and player-driven nature of MVEs. Profiling tools exist but are often limited when fine-grained data is required to be extracted from the game engine, and while useful for surface level analysis, are lacking in the ability to collect more complex game mechanics.

This gap limits our ability to understand how performance issues emerge during real-world use, especially in situations where timing, environment complexity, or subsystem interac-

tions affect responsiveness. Without continuous, contextual runtime metrics, diagnosing bottlenecks in MVEs becomes trial and error, and optimizing performance remains reactive rather than data driven.

This thesis addresses this problem by proposing a instrumentation method designed specifically for MVEs. The goal is to support systematic data collection on player behavior, and at the same time system performance, with minimal overhead using event-driven collection and data offloading to an external database for further analysis.

1.3 Research Questions

RQ1 How to design a tracing mod that effectively collects low-level performance and behavioral data in a Minecraft-based MVE?

MVEs represent a fundamentally different class of distributed systems where player agency drives unpredictable, dynamic workloads that traditional monitoring approaches cannot adequately capture. Unlike conventional server applications with predictable load patterns, MVE performance is inherently tied to human behavior, creating complex relationships between user actions and system responses that are crucial for understanding scalability bottlenecks. The scientific challenge lies in designing instrumentation for a real-time system operating under strict performance baseline while capturing meaningful research metrics from all subsystems; from JVM garbage collection to game engine event loops to player interaction handlers.

RQ2 How can a tracing mod be implemented to instrument both client- and server-side subsystems while maintaining low performance overhead?

Real-world MVE deployments involve distributed client-server architectures where performance bottlenecks can manifest across multiple system boundaries, client rendering pipelines, or server simulation logic, making unified observability essential for comprehensive performance understanding. The implementation challenge involves solving the the challenge of getting comprehensive data collection and system performance preservation in latency-critical environments. This encompasses multiple technical challenges: developing lock-free concurrency primitives for high-frequency event capture without blocking game threads, designing optimal buffering and batching strategies that minimize I/O overhead while preventing data loss during traffic bursts, and quantify the instrumentation's own performance impact to ensure the monitoring system doesn't become a significant load contributor itself.

RQ3 How can player behavioral data collected through TraceCraft enable the development of player behavior models that accurately represent real-world gameplay patterns in MVEs?

Understanding player behavior in MVEs extends beyond entertainment applications to fundamental questions about human-computer interaction in modifiable digital environments, with implications for educational platforms, collaborative virtual workspaces, and large-scale social computing systems. The scientific challenge involves transforming telemetry data into validated behavioral models while addressing several complexities: extracting meaningful behavioral signals from raw event data while preserving spatial and temporal context, establishing ground truth for behavior patterns in environments where "correct" behavior is subjective and context-dependent, ensuring model generalization across different player populations and game scenarios.

1.4 Thesis Contributions

This project contributes a lightweight and extensible tracing system designed specifically for Modifiable Virtual Environments, using Minecraft as a case study. The work bridges techniques from systems observability and real-time game analysis to provide a new way of monitoring complex, player-driven environments. The key contributions are as follows:

C1 **A design for a lightweight, extensible tracing mod** (Section 3) — A modular, Forge-compatible blueprint that hooks into both client- and server-side subsystems to capture tick, rendering, physics, and behavioral events with minimal intrusion.

C2 **A complete research artifact: open-source tracing mod with integrated real-time data pipeline and visualization stack** (Section 4) — A Forge-based mod for Minecraft 1.21.5 that instruments core game logic (ticks, entity updates, player actions) without altering vanilla behavior, released under MIT on GitHub and published on CurseForge, integrated with an InfluxDB time-series database and pre-configured Grafana dashboards to monitor performance and player behavior metrics live.

C3 **Comprehensive system validation: performance evaluation and data-driven player behavior analysis design** (Section 5 and 6) — A suite of synthetic benchmarks and play-session studies measuring accuracy, completeness, and overhead of collected metrics, combined with systematic collection of real-world player telemetry

and construction of validated player behavior models that identify common play styles, predict actions, and support applications in adaptive content, cheat detection, and server optimization.

2

Background

2.1 Modifiable Virtual Environments

Modifiable Virtual Environments (MVEs) are interactive software systems that enable end users to dynamically alter both virtual world content and underlying simulation logic through various interfaces including scripting, plugins, or in-game tools (1, 6). Unlike traditional video games with fixed functionality, MVEs explicitly encourage users to modify the simulation environment, creating content that can substantially alter computational load characteristics in ways not anticipated by the original engine architecture (4).

The defining characteristics of MVEs include real-time modification capabilities, multi-user environments, and programmatic content creation (7, 8). Users can modify virtual world objects such as player apparel or terrain, create new content by connecting components, and interact with the world through executable programs (7). These modifications range from simple terrain deformation to complex automated digital circuits, generating highly variable computational workloads that challenge traditional profiling methodologies (2).

MVEs have found applications beyond entertainment, including education, professional training, and social activism. Minecraft exemplifies this paradigm, supporting over 140 million monthly active players while allowing extensive world modification (9). However, current MVE implementations face significant scalability limitations, with commercial services partitioning players into isolated instances supporting only hundreds of concurrent users (1, 4). This scalability challenge arises from the dynamic, player-driven nature of workloads, where a single user's modifications can dramatically impact server performance. The nature of simulated element arrangements within MVE state spaces creates complex systems that require specialized monitoring approaches to understand system behavior.

2.2 Telemetry and System Monitoring in Distributed Systems

Observability refers to the capability of understanding a system's internal state and behavior through examination of its external outputs, specifically telemetry data comprising metrics, logs, and traces (10, 11). This concept, borrowed from control theory, measures how effectively a system's current state can be determined without requiring exhaustive component instrumentation (12). In distributed computing environments, observability becomes critical for managing complex, interconnected systems where traditional monitoring approaches prove insufficient (10).

Telemetry encompasses the automated collection and transmission of performance data from remote systems using sensors, protocols, and communication technologies. This data includes metrics such as CPU utilization, memory consumption, response times, and application-specific measurements. The telemetry process involves four key stages: metric specification, data transmission, processing, and analysis (13). Modern telemetry systems leverage frameworks like OpenTelemetry to provide vendor-neutral APIs for collecting metrics, traces, and logs across diverse environments (5).

The distinction between monitoring and observability lies in their scope and approach. Monitoring focuses on collecting and analyzing predefined performance indicators to track known conditions and trigger alerts (14, 15). Observability extends beyond monitoring by enabling teams to understand "unknown unknowns" and answer questions about system behavior without prior knowledge of potential issues (12). For real-time interactive systems like MVEs, observability demands high-resolution, timestamped telemetry that correlates performance signals with behavioral events at sufficient granularity to reveal interactions and system dependencies.

2.3 Time-Series Databases and Data Pipelines

Time-series databases (TSDBs) are specialized storage engines optimized for managing data using timestamped indexes, offering efficient ingestion, compression, and time-based query capabilities (16). Unlike traditional relational databases, TSDBs are designed to handle high-frequency data streams with minimal latency, making them ideal for applications requiring continuous monitoring and real-time analysis. Popular TSDB implementations include InfluxDB, which provides SQL-like querying capabilities and integrates seamlessly with visualization tools like Grafana, and TimescaleDB.

TSDBs support diverse applications across industries including IoT sensor monitoring, financial market analysis, DevOps infrastructure monitoring, and system performance tracking (16, 17). In DevOps contexts, TSDBs enable the collection of metrics such as CPU usage, memory consumption, and network throughput, facilitating real-time system health monitoring and alerting (16). The integration of TSDBs with monitoring dashboards allows for decoupled data collection, where data collection and analysis remain independently to ensure lightweight instrumentation.

2.4 Game Engine Instrumentation Frameworks

Game engine instrumentation presents unique challenges due to real-time performance constraints and the need for low-overhead monitoring approaches (18). Modern game engines employ multi-threaded architectures with separate threads for rendering, physics, input handling, and audio processing, each operating at different update frequencies and requiring specialized monitoring approaches. The primary game loop, as the one present in Minecraft, is shown in 2.1 (19).

Instrumentation techniques in game development include bytecode manipulation, event-driven monitoring, and performance profiling frameworks (20). Bytecode manipulation allows dynamic modification of Java Virtual Machine applications at runtime, enabling the injection of monitoring code without altering source files. This approach is particularly relevant for games like Minecraft, where modding frameworks utilize bytecode manipulation to insert custom functionality (21). Event-driven monitoring leverages game engine event systems to capture performance and behavioral data asynchronously, minimizing the impact on the core game loop.

Modern game engines also integrate real-time monitoring capabilities, collecting metrics such as frame rates, draw calls, polygon counts, and memory usage to optimize performance (19). The challenge lies in balancing comprehensive data collection with the stringent performance requirements of interactive entertainment applications, where even minor latency increases can significantly impact user experience (18).

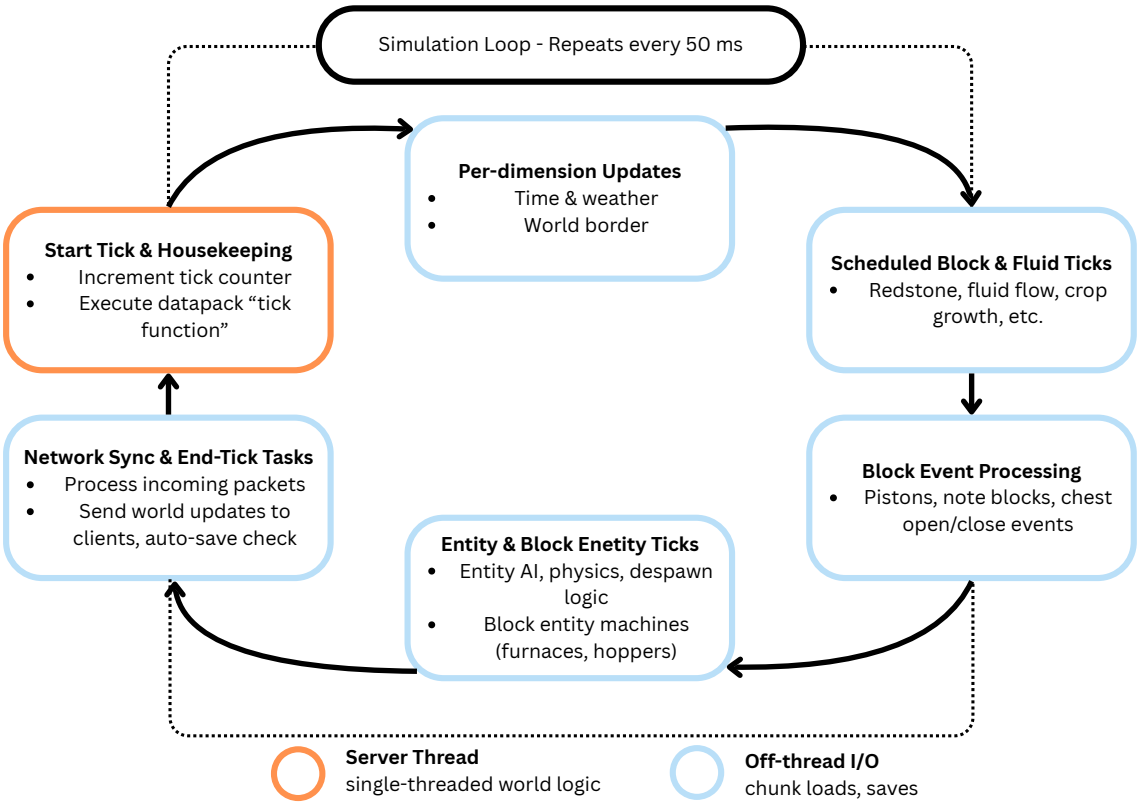


Figure 2.1: The Minecraft server's tick cycle and its core subsystems. This diagram illustrates the sequential flow of operations executed every 50 milliseconds during the simulation loop, including tick initialization, block and entity updates, scheduled tasks, and network synchronization.

3

Design of TraceCraft

3.1 Design Requirements

The core design principle that TraceCraft will focus on is offering lightweight, extensible, and minimally invasive tracing for MVEs.

1. **Low Performance Overhead**
   A requirement is to maintain, through the collection of all of metrics, a less than 2% overhead on memory consumption and CPU usage, whether low or high server load. High frequency events such as block interaction and tick executions are captured using a non-blocking event bus. All the collected data will then be enqueued inside a memory buffer, and flushed in batches (256 events at a time) to prevent constant database access.

2. **Granular and Flexible Metric Collection**
   The tracing mod must collect a well-defined set of metrics critical for Modifiable Virtual Environment (MVE) analysis. The chosen metrics must provide sufficient granularity to associate performance issues with underlying player and system behaviors and support empirical research into player-driven workload patterns. In addition to granular metrics, support the ability to have configuration-based instrumentation toggles, allowing researchers to enable and disable specific metric categories.

3. **External Integration and Data Portability**
   Stream data out of the game context. With the separation between data collection and analysis, the tracing can closely align with professional standards in distributed systems monitoring, where metric sinks and dashboards operate independently from

collection systems. Data will then be received and visualized using predefined JSON dashboards, which can be further edited by researchers.

4. **Research Suitability**
   Unlike other tools such as Spark and LagGoggles, which prioritize operational server maintenance, it should target systematic experimentation and research reproducibility. It will record structured, timestamped metrics suitable for studies, comparative profiling, and model-building. As well, the collection of behavioral metrics will allow researchers to build a better understanding of how existing player models compare to real player behavior.

3.2 Design Overview of TraceCraft

This section presents a mod for Modifiable Virtual Environments (MVEs), with the goal of capturing comprehensive performance and behavioral metrics across all system layers. The design recognizes that MVEs operate as complex, multi-layered systems where user interactions, environmental dynamics, and system resources interact in real-time, creating intricate dependencies that traditional monitoring approaches fail to adequately capture.

An MVE typically consists of a central simulation loop continuously executing distinct phases, such as entity updates, terrain processing, and network interactions. Additionally, external processes such as dynamic terrain (or chunk) generation and asynchronous network queues feed data into the main simulation loop. To comprehensively capture the runtime behavior of these environments, it is essential to strategically measure metrics across each of these subsystems (1).

This chapter begins by defining a clear, subsystem-oriented conceptual design to serve as the foundation for tracing design decisions. Metrics are subsequently structured around this design, explicitly aligning each measured parameter with its corresponding subsystem and providing rationale for its inclusion. The design further emphasizes flexibility by incorporating configuration-driven instrumentation that enables targeted metric collection suitable for multiple experimental setups.

Finally, the design ensures external integration by employing an abstracted event-driven design, facilitating the export and visualization of captured data into external analysis tools independently from the tracing system itself. By explicitly separating the conceptual design from specific implementations, this chapter supports future adaptability, research reproducibility, and clarity of design intentions (22, 23).

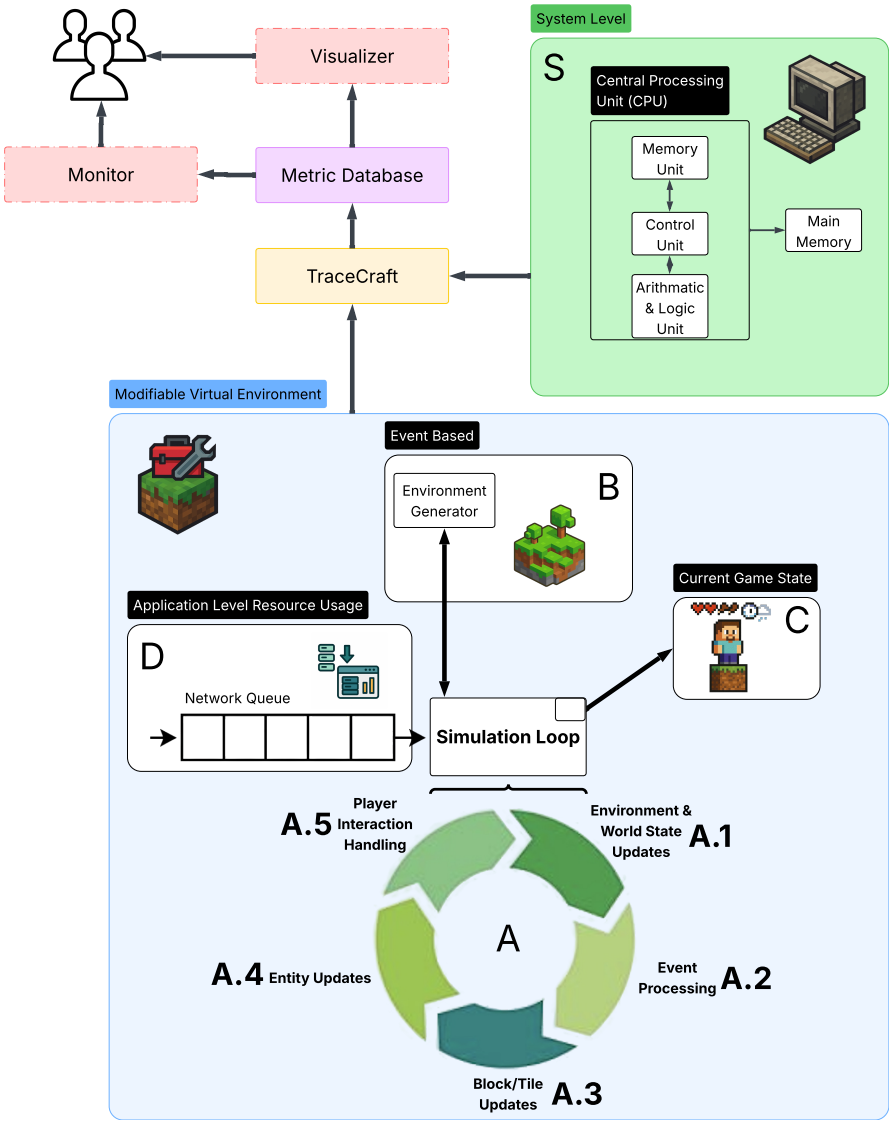


Figure 3.1: TraceCraft Design Overview

3.3 Design Alternatives

Instrumentation Strategy: Event Hooks vs. Polling vs. Profilers Alternatives. (i) Periodic polling of game state; (ii) JVM/bytecode-level profilers; (iii) Engine/event-bus hooks.

Choice. TraceCraft uses non-blocking engine/Forge event hooks.

Rationale. Polling introduces unnecessary work each tick and risks aliasing transient events. General profilers are invasive and produce low-level data that is hard to map to gameplay semantics. Event hooks provide precise, semantically meaningful triggers (e.g.,

chunk generation, block interactions) with minimal overhead and cleaner attribution to subsystems.

Data Path: Synchronous Writes vs. Buffered Asynchronous Export Alternatives. (i) Synchronous per-event writes to storage; (ii) Append-only file logging; (iii) Buffered queue with batched flush.
Choice. Lock-free producer queue with background batch flushing (256 events).
Rationale. Synchronous I/O amplifies tick jitter. File logging improves locality but still competes with disk and requires post-processing. A decoupled producer–consumer pipeline keeps the simulation hot path short and bounds I/O impact.

Storage Backend: InfluxDB/Line Protocol vs. Prometheus vs. CSV/Parquet Alternatives. (i) InfluxDB (push, tags, retention); (ii) Prometheus (pull); (iii) Flat files (CSV/Parquet).
Choice. InfluxDB as the primary sink; CSV export for portability.
Rationale. Prometheus's pull model is less natural for per-event traces and high-cardinality player labels. CSV is portable and good for archival/analysis but lacks queryable time-series features. InfluxDB offers high-ingest push semantics and straightforward dashboards; the sink is abstracted so others can be added later.

Modding Framework: Forge vs. Fabric Alternatives. Forge, Fabric.
Choice. Forge.
Rationale. The thesis environment and target servers use Forge; its event bus offers stable hooks required by TraceCraft's design. The abstraction layer avoids locking in storage or data models, so a future Fabric port remains feasible.

Concurrency Primitives: Lock-Free Queue vs. Blocking Queue vs. Ring Buffer Alternatives. (i) Blocking queues (simpler); (ii) Ring buffers (preallocated); (iii) Lock-free MPSC queue.
Choice. Lock-free MPSC queue with a single draining thread.
Rationale. Blocking risks back-pressure on the tick thread; ring buffers complicate overflow handling. A lock-free queue minimizes contention and supports bursty producers.

Metric Selection and Mapping to System Components To meet requirement 2 and collect essential metrics, TraceCraft identifies 6 key classes of metrics to collect. This section will explore these classes and present a set of metrics that have been derived from them.

1. **System-Level Metrics (S)** System-level metrics form the foundation for understanding JVM and hardware performance. Metrics such as heap usage and thread count provide indicators of system stress, highlighting resource usage or garbage collection overhead before tick-level symptoms appear (2). These metrics are collected periodically at low frequency due to their slower rate of change and negligible collection cost.

2. **Simulation Loop Metrics (A)** The core tick loop of the game server, responsible for updating game state at 20 ticks per second, is decomposed into five phases (A.1–A.5). Metrics are collected at each sub-phase to capture detailed tick-phase breakdowns. These include processing times for block updates, entity updates, and world state computation. Additionally, the simulated construct tick count (e.g. redstone circuits or tile entities) is tracked to detect high-load ticks that result from complex scripted elements. This design makes it possible to pinpoint latency contributions to individual stages of the simulation (4).

3. **Event-Based Generation Metrics (B)** Metrics in this domain measure the computational cost of asynchronous, event-triggered tasks such as chunk generation, lighting updates, and physics events (e.g., falling blocks, piston activations). These tasks typically occur in response to player movement or redstone logic but lie outside the deterministic tick loop. Measuring their execution time and frequency provides a way to correlate spike patterns with costly environmental updates, which are known to degrade performance significantly.

4. **Game State Metrics (C)** The complexity of the loaded game world is captured through metrics such as the loaded entity count, categorized by type and level. These values are strong predictors of both tick overhead and pathfinding complexity, particularly in player heavy areas or regions populated by entity mobs. This metric also serves as a bridge between simulation load and application-layer consequences such as packet traffic.

5. **Application-Level Resource Metrics (D)** To understand the load at the communication layer, metrics are collected straight from the message queues and network traffic. These include the player message-queue size and packets per second, which both act as indicated of server overload invisible to external packet sniffers. It can suit to determine whether performance degradation originates in the game logic, or further downstream. This is particularly well suited to diagnose symptoms like client-side lag or missed updates.

6. **Derived Indicators (K)** These are aggregate metrics derived from lower-layer data. These include the tick duration, Instability Ratio (ISR), ticks-per-second (TPS) , and custom metrics such as player–chunk proximity. These metrics act as summaries of system performance and provide a top-level signal of degraded responsiveness. Because they integrate across domains, they are valuable for real-time dashboards and anomaly detection.

The selection of these metrics was guided by a subsystem-oriented metric collection and consolidated in a table as referenced in 3.1. Previous research warns against selecting metrics based solely on design intent, as metrics with different objectives can behave similarly under load or vice versa (25). To address this, each metric in TraceCraft was chosen based on observed correlations between player behavior and system symptoms. This ensures every metric is tied to a distinct component in the system diagram and offers diagnostic specificity rather than redundancy.

3.4 Tracing Player Behavior in MVEs

While traditional performance monitoring focuses on system internals, MVEs are driven by players' actions. TraceCraft has a focus on behavioral metrics, (Category E) capturing this human element, which is novel in MVE research. These metrics as referenced in 3.2, and quantify what players actually do and allow correlating those actions with performance outcomes. For example:

1. **Player Path Trace:** Logging of player movement paths as they influence workload, as players explore unloaded areas triggering chunk generation (Component B) and load new states (Component C). Correlating path trajectories with system metrics lets us see, e.g. how running into unexplored terrain (lots of chunk loads) raises tick time.

2. **Block Interaction:** Counting when and where players place or break blocks. Every block change forces updates in the simulation (Component A) and possibly lighting/-physics events (Component B). By measuring block interactions, we can attribute spikes in physics computations or tick duration to specific player actions.

3. **Combat and Interaction Events:** We record events like player attacks or entity interactions. Combat can generate bursts of activity across subsystems: entity updates (Component A), network traffic to update nearby clients (Component D), and state changes for health/status (Component C). Logging combat events lets us link sudden multi-component loads to actual gameplay events.

These behavioral metrics go beyond standard monitoring by explicitly measuring player-caused workload. In doing so, we fill a gap identified in prior work: MVEs have unique, player-driven dynamics that require a human-aware data approach. In research methodology terms, this is part of our study's rationale: we justify collecting behavioral data by showing it addresses a known gap (the lack of user-contextualized metrics). In practice, capturing these behaviors means our monitoring not only measures "what happens" inside the game, but also "why" it happens from the players' side.

3.5 How TraceCraft Meets the Design Requirements

Requirement 1: Low Performance Overhead.

- *Hot-path minimalism:* Metric producers execute on engine event hooks and push lightweight records into a lock-free MPSC queue; no blocking or allocation-heavy formatting occurs on the tick thread.

- *Bounded I/O:* A single background consumer drains the queue and writes in configurable batches (default 256 events) to amortize I/O and avoid per-event flushes.

- *Adaptive load-shed:* When the queue nears capacity, non-critical metrics (e.g., high-frequency debug counters) are dropped first; critical latency indicators (e.g., tick duration) are preserved.

Requirement 2: Granular and Flexible Metric Collection.

- *Subsystem coverage:* Categories A, B, C, D, S, and derived K map directly to simulation phases, event-driven generation, world state, application/network, and system health.

- *Configuration toggles:* Researchers can enable/disable metric groups and tune sampling rates (e.g., position sampling Hz), limiting overhead to what each study needs.
- *Rich labels:* Each event includes tags (dimension/world, player ID, entity type) to support fine-grained slicing without duplicating probes.

Requirement 3: External Integration and Data Portability.

- *Abstract sink interface:* The exporter targets a pluggable `Sink` API; InfluxDB is the default, with CSV export for archival and ad-hoc analysis.
- *Dashboard-first:* Predefined JSON dashboards visualize ISR, tick-phase breakdowns, and behavior overlays; sinks can be swapped without touching instrumentation sites.

Requirement 4: Research Suitability.

- *Structured, timestamped events:* All metrics are time-aligned and schemed for reproducible queries and cross-run comparisons.
- *Behavioral linkage:* Player-behavior metrics (Category E) are captured alongside system metrics, enabling causal analyses between actions and load.
- *Reproducibility aids:* Config files (rates, enabled metrics, sink settings) are versioned with experiment runs to ensure consistent setups.

Data flow summary. Metric-producing components enqueue data into a lock-free concurrent queue, while a background processing thread periodically drains this queue and writes events in batches to the database. This decoupled producer–consumer model ensures low-latency tracing with minimal performance overhead and clean separation between collection and export.

Table 3.1: Curated metric set collected by *TraceCraft*. Categories: (A) Simulation Loop (B) Event-Based Generation (C) Game state (D) Application Level Resource Usage (S) System Level (K) Derived Indicator

Category	Metric	Rationale
A	Tick–phase breakdown (entity, terrain, network)	Shows which subsystem exceeds the 50 ms budget.
A	Simulated–construct tick count (redstone / block entities)	Worst case workload highlighted by *Meterstick*.
B	Chunk generation time	Terrain-generation spikes, only measurable from inside the game loop.
B	Lighting update count	Expensive environment computation strongly correlated with lag.
B	Physics event count (falling blocks, pistons)	Completes environment profile for TNT / farm scenarios.
C	Loaded entity count (by type/level)	Cheap to calculate; predictor of AI and path-finding load.
D	Player message-queue size	Yardstick overload indicator, invisible to external packet sniffers.
D	Packets per second & average size	Relates network bursts with tick stalling.
S	Heap usage & thread count	JVM health baseline; negligible overhead.
K	Tick duration & Instability Ratio (ISR) (24)	Primary variability signal; cannot be observed outside the server JVM.
K	Player–Chunk proximity (min distance to unloaded chunk)	Custom metric warning of generation issues at the loaded-edge.
K	Player latency (ping)	QoS metric showing server stalls to perceived player delay.
K	Ticks-per-second (TPS)	Sanity check that tick maths are correct.

Table 3.2: Player Behavior metric set collected by *TraceCraft*. Tier 1 = highest priority for implementation. Category: (E) Player Behavior.

Category	Metric	Rationale
E	Player Path Trace (sampled locations over time)	Reconstructs movement paths to identify heat-map zones, common routes, and choke points.
E	Block Interactions by Block Type (break/place events with block ID)	Reveals building style, resource-gathering preferences, and "hot" build locations.
E	Item Usage Events (right-click/use per item)	Shows playstyle via consumable and tool usage (e.g. food, potions, tools).
E	Combat Events (damage dealt/received, entity type)	Quantifies PvE/PvP engagement, DPS patterns, and mob-farming behavior.
E	Death Events & Cause (timestamp, location, cause)	Highlights dangerous zones and difficulty spikes; tracks player skill progression.
E	Distance Traveled per Session (summed path length)	Indicates exploration intensity per login session; correlates roaming with performance.
E	Time Spent in Each Biome/Region (biome ID vs. duration)	Shows biome preference and activity hotspots (e.g. build/farm locations).
E	Social Proximity & Interaction (player–player distance, trades)	Surfaces social graphs, collaborative clusters, and guild formation.
E	Idle Time / AFK Detection (no movement or interaction threshold)	Distinguishes active play from AFK to avoid skewing performance metrics.

4

Implementation of TraceCraft

4.1 Implementation Overview

TraceCraft's implementation realizes the design principles established in Chapter 3 through a comprehensive instrumentation system built on Minecraft 1.21.5 using the Forge modding framework. The implementation leverages Java 21 and employs a modular architecture that separates client-side and server-side concerns while maintaining seamless integration through Forge's event bus system.

The development leveraged IntelliJ IDEA with Gradle for build automation, ensuring reproducible builds and dependency management. A critical implementation challenge involved packaging external dependencies such as InfluxDB and SQLite JDBC libraries directly into the mod JAR using ForgeGradle's *Jar-in-Jar* approach, inspired by best practices from projects like *Distant Horizons* (26). This packaging strategy eliminates the need for separate dependency installation by server administrators, simplifying deployment while maintaining functionality.

The Forge mod integration points used by the tracing mod are illustrated in Figure 4.1. TraceCraft implements a configuration system using Forge's configuration system API that allows static or hot re-loadable customization configuration for the mod. Once the mod has been executed on a server's launch the first time, it will create a dedicated .TOML config file with default parameters. The configuration file defines several configuration values, such as boolean toggles for different groupings of metrics collection, as well as setting the InfluxDB's external URL. With this, users of TraceCraft will have the ability to tweak its behavior to reduce overhead if certain metrics are not necessarily required for collection.

TraceCraft's architecture implements the event-driven data collection pipeline outlined in the design phase through three primary components: client-side performance monitoring,

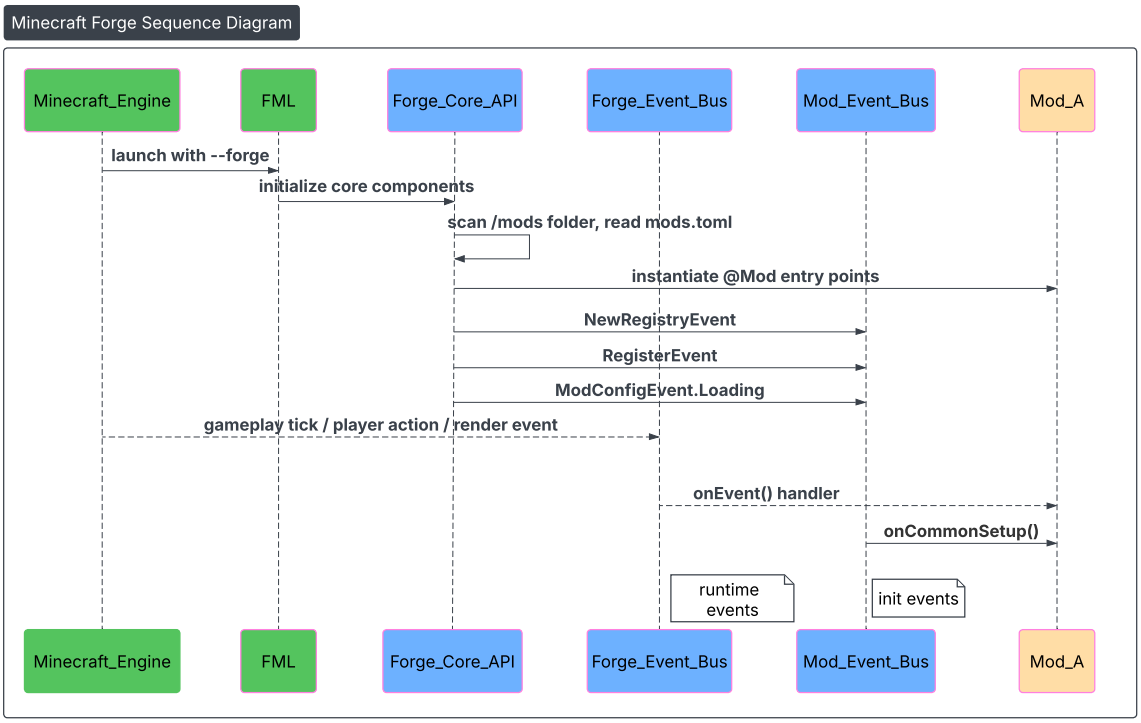


Figure 4.1: Forge mod integration points used by the tracing mod.

server-side comprehensive instrumentation, and asynchronous data handling. The client-side implementation focuses on capturing player experience metrics including frame rates, memory usage, and network latency, while the server-side component provides extensive instrumentation across all major game subsystems including tick loops, entity management, world generation, and player behavior tracking.

Central to the implementation is a lock-free concurrent queue that decouples metric collection from data export, ensuring minimal impact on game performance. Events are captured through Forge's event bus system and enqueued as lightweight JSON payloads, which are subsequently drained by a background thread that batches writes to InfluxDB at 2-second intervals. This design prevents blocking operations on the main game thread while ensuring data integrity and completeness.

Figure 4.2 illustrates the complete system architecture, showing the data flow from in-game event capture through external visualization and alerting systems. The implementation maintains strict separation between data collection and analysis components, adhering to observability best practices that enable independent scaling and modification of visualization and storage layers.

The modular design facilitates extensibility through well-defined interfaces for metric

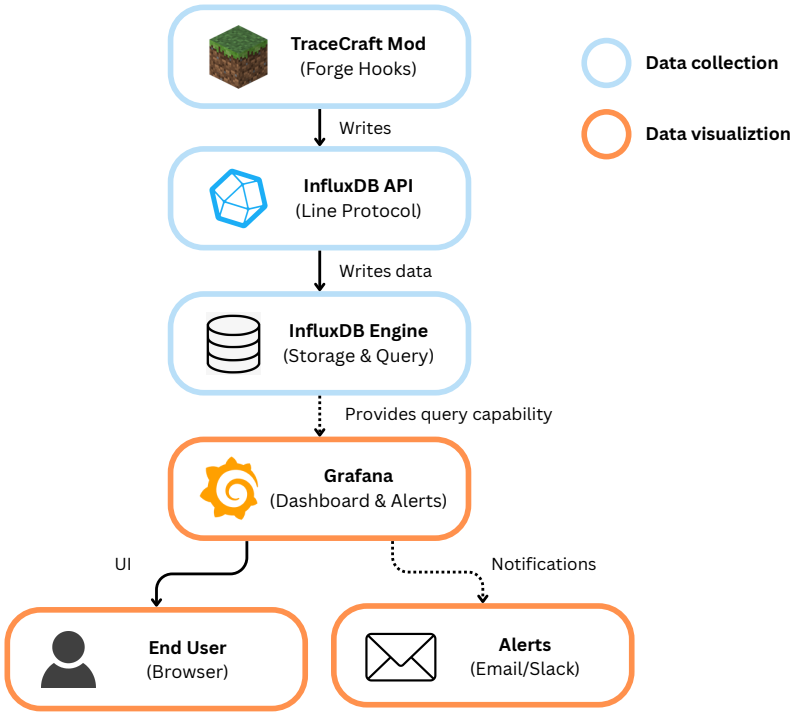


Figure 4.2: Expanded system architecture for real-time tracing, storage, visualization, and alerting.

producers, event processors, and data sinks. This architecture supports future enhancements such as additional metric categories, alternative storage backends, or integration with other monitoring frameworks while preserving the core low-overhead instrumentation philosophy that drives TraceCraft's effectiveness in latency-sensitive gaming environments. The project's source code is hosted publicly on GitHub (27), allowing for further version control and the ability for further expansions in more metric collection or performance improvements.

4.2 Client-side Implementation

On the client side, TraceCraft's implementation focuses on capturing performance metrics that reflect the player's local experience. The mod hooks into Forge's event bus to listen for the client tick event on each frame, as shown in Listing 4.1. Specifically, a static handler in *ClientHooks* is annotated to subscribe to *TickEvent.ClientTickEvent*.

By checking the event phase, TraceCraft ensures it runs its logic at the end of each tick; after the game has updated all other sections the tick. This timing is important to gather accurate metrics and not interfere with game processing.

Listing 4.1: Client tick event handler in TraceCraft

```
@Mod.EventBusSubscriber(modid = TraceCraft.MODID, value = Dist.
    ↪ CLIENT, bus = Mod.EventBusSubscriber.Bus.FORGE)
public final class ClientHooks {

    private static long lastSent = 0;

    @SubscribeEvent
    public static void onClientTick(TickEvent.ClientTickEvent e) {
        if (Minecraft.getInstance().getConnection() == null) {
            ↪ return; } // not connected yet
        if (e.phase != TickEvent.Phase.END) return;
```

Listing 4.2: Client-side metric collection and throttling

```
long now = System.currentTimeMillis();
if (now - lastSent < 5_000) return;      // send once per 5 second
lastSent = now;

int   fps = Minecraft.getInstance().getFps();
long  mem = Runtime.getRuntime().totalMemory() - Runtime.getRuntime
    ↪ ().freeMemory();
long ping = Objects.requireNonNull(Minecraft.getInstance()
                                            .getConnection()
                                            .getServerData())
                                            .ping;
```

Each tick, the mod collects the frames-per-second (FPS), the memory usage (heap used), and the current network latency (ping) from the Minecraft client instance. These metrics are lightweight to obtain; for example, Minecraft's *getFps()* provides the recent frame rate, and memory usage is computed from the runtime memory totals. To avoid excessive overhead or network spam, TraceCraft throttles the data collection to one sample every few seconds. The code maintains a timestamp of the last sent packet; if a tick occurs too soon after the previous sample (by default, within 5 seconds), the client skips sending a new update. This simple rate limiting ensures that even at high frame rates, the client will not flood the server with metrics, thereby minimizing the performance impact on the client side (see Listing 4.2).

When a client metric sample is taken, TraceCraft will package the data into a custom

Listing 4.3: Forge network channel definition for TraceCraft

```
ResourceLocation id = ResourceLocation.fromNamespaceAndPath(
    ↪ TraceCraft.MODID, "main");

CHANNEL = ChannelBuilder
        .named(id)
        .networkProtocolVersion(PROTO)
        .acceptedVersions((status, v) -> v == PROTO)
        .simpleChannel();

registerPackets();
```

network packet and send it to the server for event creation. Forge's networking API is used to define a custom network channel (TraceCraft.CHANNEL) for the mod, as shown in Listing 4.3.

The mod registers a message type *ClientMetricsPacket* on this channel, which carries three fields: FPS, memory, and ping. On the client, the *ClientHooks* tick handler creates a new *ClientMetricsPacket* with the collected values and dispatches it to the server using *PacketDistributor.SERVER*. The use of Forge's network channel (as opposed to, say, standard game chat or command channels) provides reliable, ordered delivery and integrates with the mod lifecycle. By design in Tracecraft, only the server-side mod needs to handle this packet; the channel is configured such that the server will accept it and invoke the packet's handler method on the main server thread.

This approach to event-driven, packet-based strategy separates the client performance from server logic, allowing it to be easily toggled if not necessary or even further expanded. The client does not need to perform any analysis; simply taking measures and forwarding them periodically. The choice to go with network packets, as opposed to writing directly in the database, allows for the mod to piggyback on Minecraft's networking, avoiding external I/O on the client. (28)

4.3 Server-side Implementation

The server side of Tracecraft is the most important as it is the core instrumentation logic, capturing a broad range of game events and state relevant to performance and player behavior. Similarly it uses Forge's event bus extensively to **intercept gameplay events** on the server. All event handlers are grouped inside the *ServerHooks* class, which is annotated

Listing 4.4: Server-side event subscription for block placement

```
@Mod.EventBusSubscriber(modid = TraceCraft.MODID, bus = Mod.
    ↪ EventBusSubscriber.Bus.FORGE, value = Dist.DEDICATED_SERVER)
public final class ServerHooks {

    @SubscribeEvent
    public static void onBlockPlace(BlockEvent.EntityPlaceEvent e)
        ↪ {
        WorldInteractionHandler.handleBlockPlace(e);
    }
```

to register its static methods to the Forge event bus for the server environment. Example server-side event subscription for block placement is shown in Listing 4.4.

This ensures that when the mod is running on a server, the specified handlers will be invoked on each relevant event.

Events captured on the server-side include world ticks, player actions, and world changes. For example, the mod will listen for any **server tick events** in order to measure performance: one handler will record start of tick and another the end. Using this we can measure the **tick duration** precisely by comparing system time. This is further used to allow us to get a tick count over a 5 second window, in turn allowing us to capture the **ticks-per-second**. We also hook **player-centric events** to track behavior and load. Simply we record login and logout events using a *PlayerLoggedInEvent* and *PlayerLoggedOutEvent* in order to calculate how far players travel during a session, how long they spend in each biome, and how much time they idle. The handler for player logout is shown in Listing 4.5.

In order to allow this, Tracecraft utilizes a session state helper *PlayerSessionData* while a player is online, updating distance and time counter periodically. For instance, a scheduled task will sample each player's position every second and accumulate their movement distance. If the player has not performed any action for a set amount of time, they will also be flagged as idle and count the time until they again perform an action (movement or interaction).

In addition to player specific sessions, **world events** are also traced. The mod will listen to interactions such as block placements and block breaks, routing them to the *WorldInteractionHandler* class that creates events for *block_place* and *block_break* events with details about the block involved. Furthermore, chunk generation events are captured by measuring the duration taken to generate a chunk, recording chunk coordinates, dimension information, and generation time, and subsequently queuing these details for asynchronous

Listing 4.5: Handling player logout event and session data

```
public static void handleLogout(PlayerEvent.PlayerLoggedOutEvent e)
    ↪ {
    UUID id = e.getEntity().getUUID();
    Event.sendEvent("logout", Event.createPlayerPayload(id));

    JsonObject distancePayload = Event.createPlayerPayload(id);
    distancePayload.addProperty("distance", PlayerSessionData.
        ↪ getSessionDistance().getOrDefault(id, 0.0));
    Event.sendEvent("session_distance", distancePayload);

    for (var entry : PlayerSessionData.getBiomeTime().getOrDefault(
        ↪ id, Map.of()).entrySet()) {
        JsonObject biomePayload = Event.createPlayerPayload(id);
        biomePayload.addProperty("biome", entry.getKey());
        biomePayload.addProperty("duration_ms", entry.getValue());
        Event.sendEvent("biome_time", biomePayload);
    }

    JsonObject idlePayload = Event.createPlayerPayload(id);
    idlePayload.addProperty("idle_ms", PlayerSessionData.
        ↪ getIdleTimeMs().getOrDefault(id, 0L));
    Event.sendEvent("session_idle", idlePayload);

    PlayerSessionData.clearPlayerData(id);
}
```

event storage under the *chunk_ generated* event type. Additionally, TraceCraft monitors physics related world interactions through the *NeighborNotifyEvent*. The mod evaluates the relevance of a block state based on its association with computationally expensive operations, such as falling blocks, pistons, liquids, and redstone components. When a relevant physics interaction is detected, the event details such as the block type and precise coordinates are packaged into a structured JSON event (*physics_ event*) and queued for asynchronous processing.

All of the metrics collected on the server-side, including the packets received by the clients, will be funneled into a **central event queue** rather than being immediately sent to the database. This is achieved through a static utility *Event.sendEvent(type, payload)* method that wraps the data in *Event* object, containing JSON payload and timestamps, with the help of helper functions described in Listing 4.6.

Listing 4.6: Event queueing utility methods

```
public static void sendEvent(String type, JsonObject payload) {
    TraceCraft.QUEUE.addEvent(new Event(payload, type));
}

public static JsonObject createPlayerPayload(UUID playerId) {
    JsonObject o = new JsonObject();
    o.addProperty("player", playerId.toString());
    return o;
}
```

This design will ensure that no matter how many events fire, the impact on the game loop will remain small, as events are only enqueued into memory. The queue by default is bounded to 1000 events to prevent runaway memory usage; if it were to fill up, additional events will be dropped to avoid stalling the server. The event queue itself is implemented as shown in Listing 4.7.

4.4 Data Handling and Performance Impact Considerations

To allow the retrieval and export of queued events, Tracecraft uses a **background thread** that will periodically drain the queue and write it to the InfluxDB. This is done with the help of an *InfluxDBHelper*, which manages the connection to the database instance and running a query on a fixed interval. The scheduling is done by a single-threaded executor service; every 2 seconds it will wake up and atomically extract a batch of events for processing (Listing 4.8).

Within the InfluxDBHelper, events are converted to points in the time-series database. Each event's JSON payload is parsed according to its type tag; example "tick_metrics", "block_break", "login", etc., and mapped to InfluxDB fields and tags. This, when processing an event type, creates an InfluxDB point with measurements and fields populated from the event's data. Using this JSON based schema allowed the flexibility to add or remove fields without changing the method signatures and allows the reuse of generic code for en-queuing and writing events. The only trade-off that it has introduced is the minor cost of serialization/de-serialization, but given the low volume of events, this cost is negligible.

An illustration of block place/break events is shown in Figure 4.3.

Performance considerations were integral to TraceCraft's design. The goal was to introduce rich tracing with minimal overhead on the game. Several strategies were employed to achieve

Listing 4.7: Bounded concurrent event queue implementation

```
public class EventQueue {
    private final ConcurrentLinkedQueue<Event> q;

    public EventQueue() {
        this.q = new ConcurrentLinkedQueue<>();
    }

    public void addEvent(Event e) {
        q.add(e); // non blocking add
    }

    public List<Event> drain(int max) {
        List<Event> out = new ArrayList<>();
        for (int i = 0; i < max && !q.isEmpty(); i++) {
            Event e = q.poll(); // non-blocking poll
            if (e != null) {
                out.add(e);
            }
        }
        return out;
    }
```

this:

1. **Non-blocking, lock-free data passing** Since Tracecraft uses a concurrent queue for metrics, it avoids locks on the main thread and lets the producer (game events) and consumer (DB thread) work in parallel.

2. **Batch processing** By writing many different InfluxDB points in a single operation to InfluxDB, it reduces the overhead, the size of 256 was chosen empirically as a good balance between timely data and efficiency.

3. **Throttling and sampling** Since not every event is traced, some metrics are sampled at intervals, for instance client performance is sent at 5 second intervals, while some other metrics at 10 seconds.

   Example throttling timers for various metrics events are shown in Listing 4.9.

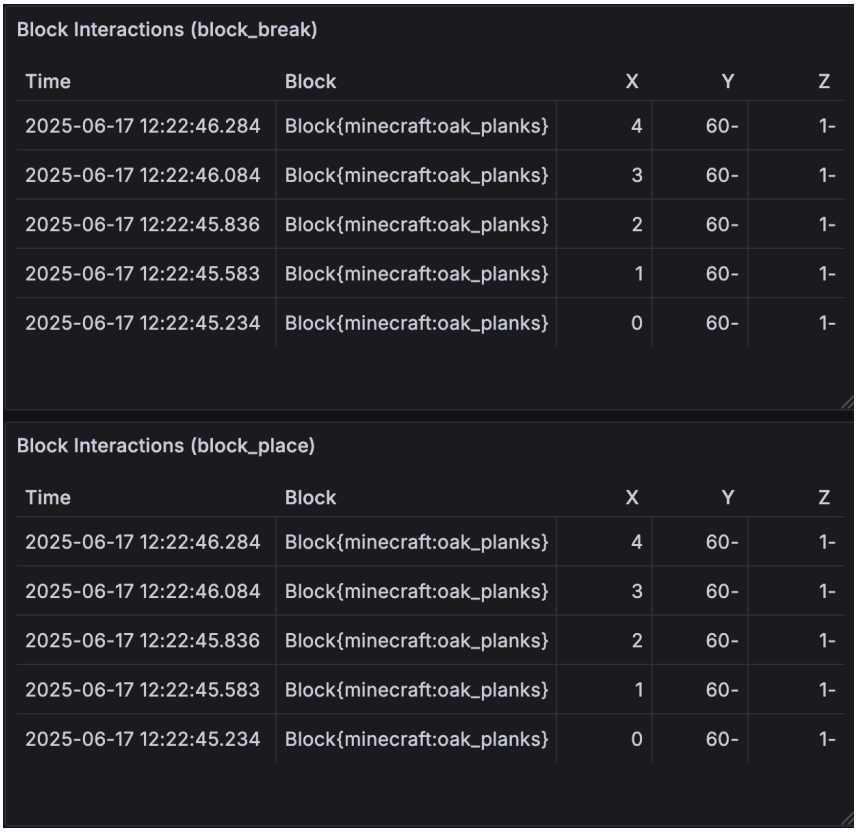
Listing 4.8: Background thread draining event queue and batching writes

```
public void run() {
    var batch = TraceCraft.QUEUE.drain(256);
    if (batch.isEmpty()) {
        return;
    }

    BatchPoints.Builder batchPointsBuilder = BatchPoints
            .database(bucketName)
            .retentionPolicy(retentionPolicy);
```

Listing 4.9: Example throttling timers for various metrics events

```
    private static long nextPlayerCountEvent = System.
        ↪ currentTimeMillis() + 60_000L;
    private static long nextChunkMarginEvent = System.
        ↪ currentTimeMillis() + 10_000L;
    private static long nextQueueMetricsEvent = System.
        ↪ currentTimeMillis() + 10_000L;
    private static long nextSystemMetricsEvent = System.
        ↪ currentTimeMillis() + 10_000L;
    private static long nextEntityCountEvent = System.
        ↪ currentTimeMillis() + 5_000L;
```



Block Interactions (block_break)

Time	Block	X	Y	Z
2025-06-17 12:22:46.284	Block{minecraft:oak_planks}	4	60-	1-
2025-06-17 12:22:46.084	Block{minecraft:oak_planks}	3	60-	1-
2025-06-17 12:22:45.836	Block{minecraft:oak_planks}	2	60-	1-
2025-06-17 12:22:45.583	Block{minecraft:oak_planks}	1	60-	1-
2025-06-17 12:22:45.234	Block{minecraft:oak_planks}	0	60-	1-

Block Interactions (block_place)

Time	Block	X	Y	Z
2025-06-17 12:22:46.284	Block{minecraft:oak_planks}	4	60-	1-
2025-06-17 12:22:46.084	Block{minecraft:oak_planks}	3	60-	1-
2025-06-17 12:22:45.836	Block{minecraft:oak_planks}	2	60-	1-
2025-06-17 12:22:45.583	Block{minecraft:oak_planks}	1	60-	1-
2025-06-17 12:22:45.234	Block{minecraft:oak_planks}	0	60-	1-

Figure 4.3: Block place/break events

5

TraceCraft Implementation Evaluation

5.1 Main Findings

The TraceCraft implementation evaluation demonstrates that the mod successfully achieves its design objectives of providing comprehensive, low-overhead performance and behavioral monitoring for Minecraft-based MVEs. Key findings include: (1) exceptional accuracy in player behavior tracking with *sub-block precision across sampling strategies*, RMSE ≤ 0.074 blocks for fixed intervals (2s/5s/10s) and $\approx$0.028 blocks under block-change sampling, while preserving path geometry; (2) minimal performance impact with only 5–10% CPU overhead and 6–15% memory increase compared to vanilla servers, demonstrating TraceCraft's lightweight design effectiveness; and (3) superior resource efficiency compared to alternative monitoring solutions like UnifiedMetrics, while providing significantly more comprehensive data collection capabilities. These results validate TraceCraft as a robust, production-ready tool for MVE research and server administration.

5.2 Experimental Setup

All experiments were conducted on a Minecraft Java Edition 1.21.5 server hosted on a Raspberry Pi 5 (8 GB RAM, 4 vCPUs). TraceCraft was installed as a Forge 1.21.5 server-side mod and configured to write metrics to InfluxDB. Both InfluxDB 2.8 and Grafana 10 were deployed on the same Raspberry Pi using Docker Compose.

A combination of controlled scenarios was then performed on this server, along with load tests, and real-world multiplayer sessions were conducted. The purpose was to evaluate the

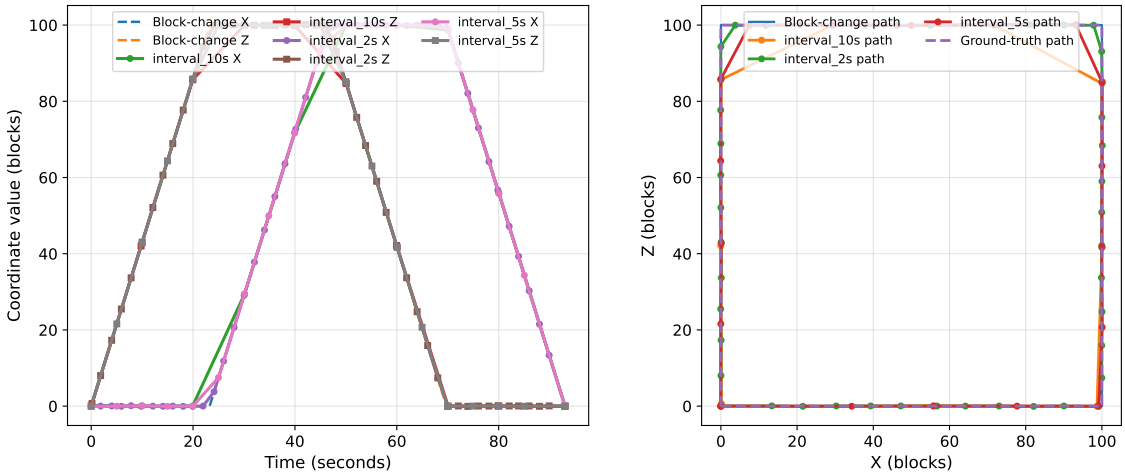


Figure 5.1: Player path validation: time-series (left) and 2D reconstruction (right) comparing fixed 2s/5s/10s intervals to block-change (near-continuous) sampling.

correctness and performance overhead that TraceCraft will introduce to a server.

5.3 Player-Tracing Validation Experiments

Player Path Trace Validation A controlled experiment was conducted where a player walked a precise 100×100 block square path at 4.3 blocks/second (default walking speed), starting and ending at coordinates (0,0), completing the path in 93 seconds. To evaluate the accuracy of fixed-interval sampling, we compared three fixed intervals (2s, 5s, 10s) against an idealized reference obtained through continuous (block-change-level) tracking. Figure 5.1 shows a side-by-side view: the temporal progression of X and Z coordinates (left) and the reconstructed 2D path (right).

- **Fixed-Interval Sampling**: regular time-based sampling at 2s, 5s, and 10s
- **Block-Change Reference**: near-continuous, block-level accurate position tracking

Quantitative Results (square perimeter = 400 blocks):

Figure 5.1 shows that all fixed-interval strategies preserve the overall square trajectory, with deviations concentrated at corner turns where linear interpolation between sparse samples under-represents sharp direction changes. As expected, *path length error* decreases as sampling frequency increases: moving from 5s to 2s reduces path length error by ∼52.6% (from 11.68 to 5.53 blocks), whereas 10s increases it by ∼108.8% (to 24.39 blocks). Pointwise error (RMSE) remains *sub-block* for all intervals; differences across 2s/5s/10s are small

Method	Samples	Mean Err (blocks)	RMSE (blocks)	Max Err (blocks)	Path Err (blocks)	Path Err (%)
2s interval	47	0.065	0.072	0.136	5.53	1.38
5s interval	19	0.057	0.067	0.147	11.68	2.92
10s interval	10	0.068	0.074	0.142	24.39	6.10
Block-change (ref.)	401	0.025	0.028	0.071	0.24	0.06

Table 5.1: Player Path Trace Validation across sampling methods. Errors are computed versus the analytical ground-truth path at the sampled times.

(0.067–0.074 blocks) and dominated by measurement noise at each sample, while the block-change reference approaches the noise floor.

Key Finding: TraceCraft reconstructs movement with sub-block accuracy across fixed intervals (2s/5s/10s), while higher sampling frequencies substantially improve trajectory fidelity (path length error) without imposing continuous recording. For analyses sensitive to geometric fidelity (e.g., route heatmaps), 2s offers a strong accuracy–overhead trade-off; block-change remains the reference for near-perfect paths.

Combat Events Validation A player performed combat events on four different entity types (zombie, creeper, cow, sheep) using a wooden sword with known damage mechanics. Validating against known Minecraft damage values and entity health points, the results are presented in Figure 5.2.

Quantitative Results:

- **Damage Aggregation Accuracy**: 100% across all entity types
- **Hit Count Detection**: 100% accuracy for all 15 recorded combat events
- **Event Sequencing**: Precise timestamp recording validated correct temporal ordering
- **Entity Classification**: 100% accuracy in entity type identification

The validation results shown in Figure 5.2 demonstrate perfect correspondence between expected and recorded values across all tested parameters. The left panel shows damage validation where TraceCraft accurately captured the exact damage values (20 HP for zombie and creeper, 10 HP for cow, 8 HP for sheep) as dictated by Minecraft's combat mechanics. The right panel confirms precise hit count detection, with the system correctly recording 5 hits each for zombie and creeper (requiring multiple hits due to higher health), and 3 and 2 hits respectively for cow and sheep. The perfect alignment between expected and recorded

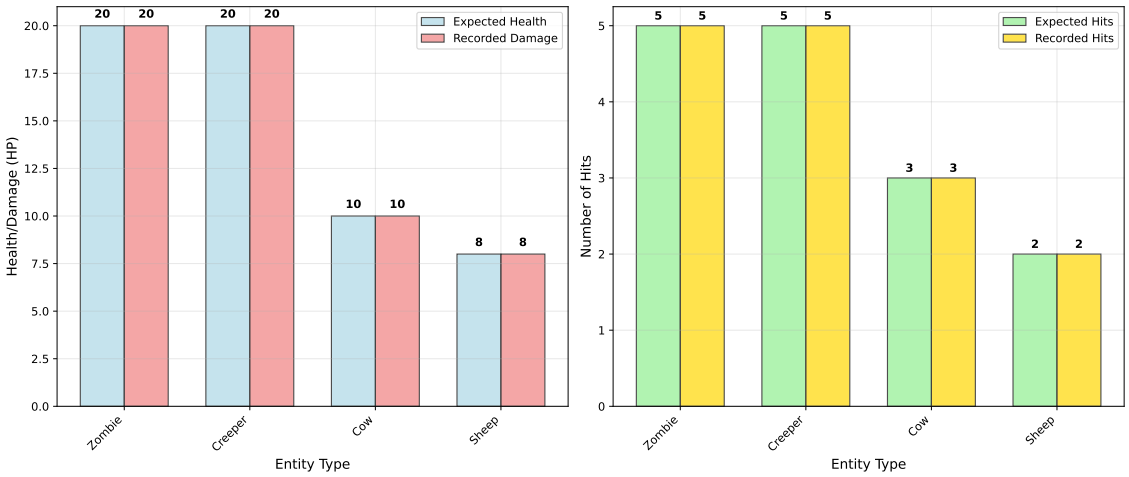


Figure 5.2: Combat Events Validation Results. Left: Damage Validation: Expected vs Recorded, Right: Hit Count Validation: Expected vs Recorded.

values across all entity types validates the system's ability to capture rapid, successive combat events without data loss or timing errors.

Key Finding: TraceCraft demonstrates 100% accuracy in combat event detection and damage calculation across all entity types and combat scenarios, validating the system's capability to capture rapid, high-frequency gameplay interactions with complete fidelity for comprehensive behavioral analysis.

5.4 Performance Overhead Analysis

To quantify the resource impact of TraceCraft, we executed a series of automated load tests using HeadlessMC, the only available bot-driven framework compatible with Minecraft Java Edition 1.21.5 under Forge. Although this choice limited flexibility compared to more modern tools, it guaranteed repeatable, realistic player-behavior simulations without altering the server core.

In each experiment, up to ten "headlessmc" bots with identical in game version, mod configuration, and scripted actions (connect, move, place and break blocks, disconnect) were launched in parallel inside Docker containers. These containers, built from a custom HeadlessMC image, bundled the Forge runtime and the TraceCraft mod JAR. After each run, we recorded per-second CPU utilization and resident memory (RSS) via `pidstat`, then compared these metrics against an unmodded (vanilla) Forge server under the same conditions.

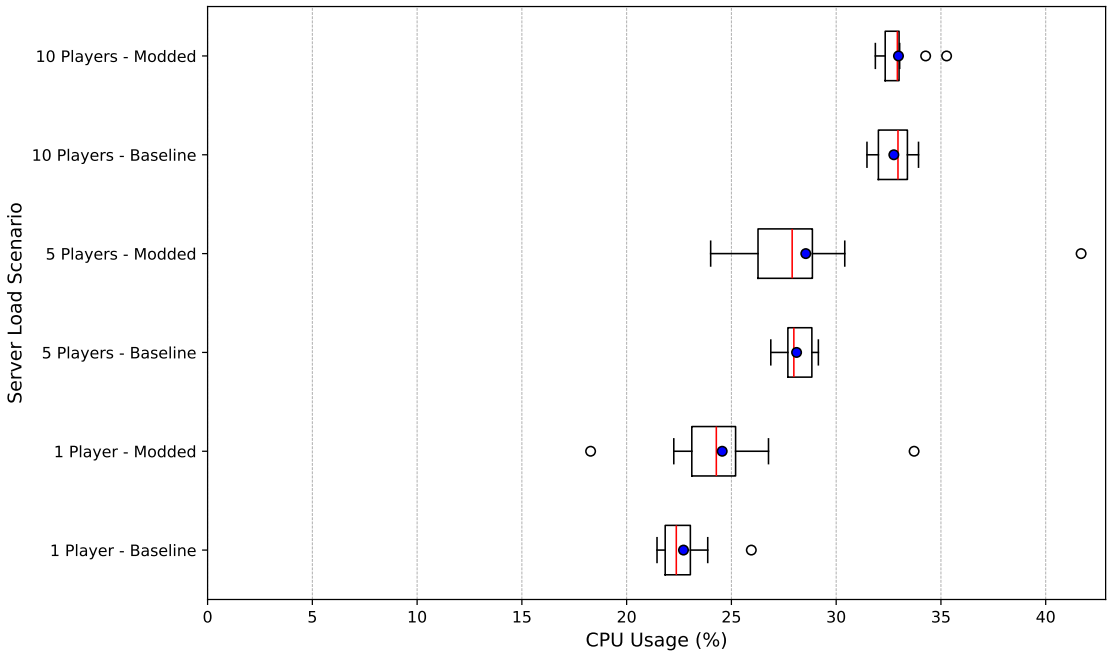


Figure 5.3: CPU usage for vanilla and TraceCraft-enabled servers across increasing bot counts.

Across all scenarios, TraceCraft imparted a modest resource cost: approximately 5–10% additional CPU usage and 6–15% more memory consumption relative to an unmodded Forge server. These overheads stem from capturing, buffering, and batching performance and behavioral metrics, yet remain low enough to support typical small- to medium-scale multiplayer sessions without degrading responsiveness.

CPU Utilization With a single simulated player, the vanilla server averaged 22.7% CPU utilisation, while TraceCraft required 24.6%; an extra 1.8 percentage points (≈8.1% relative overhead). At the mid-tier load of five players, both configurations drew nearly the same share of a core: vanilla at 28.1% versus TraceCraft's 28.6%; a modest 0.4 pp increase (≈1.6% overhead) for the modded build. Under the heaviest load (ten players), utilisation again converged, with TraceCraft edging just above vanilla (33.0% vs 32.8%), a negligible 0.2 pp difference (≈0.7% overhead).

Figure 5.3 reveals an interesting scaling pattern where TraceCraft's CPU overhead becomes proportionally smaller as player count increases. This behavior suggests that the mod's instrumentation overhead is largely fixed rather than scaling linearly with player activity. The box plots show consistent median values with relatively tight interquartile ranges,

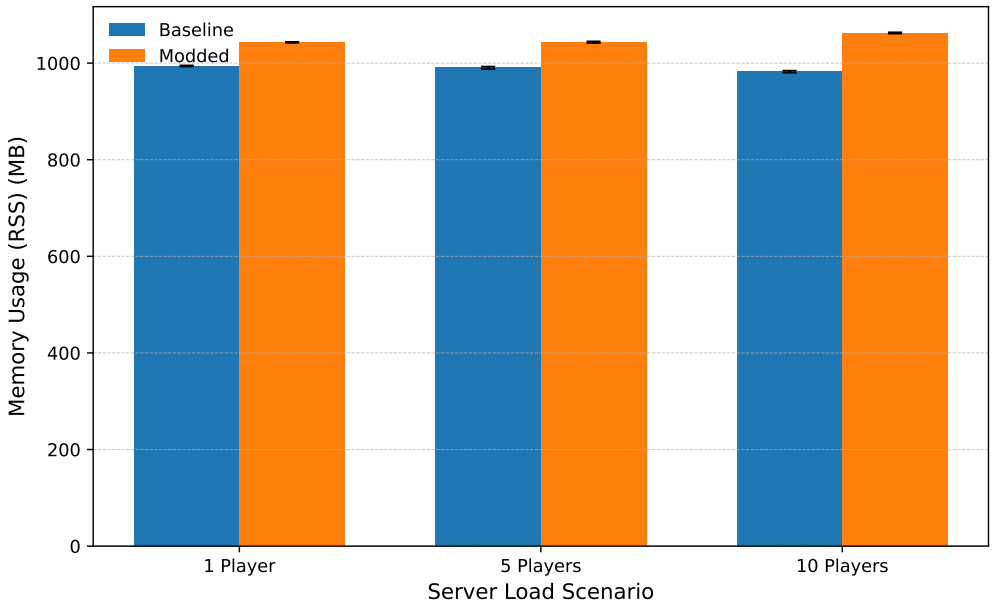


Figure 5.4: Resident memory (RSS) average of a 10 minute vanilla and TraceCraft-enabled servers session.

indicating stable performance across multiple test runs. The decreasing relative overhead at higher player counts demonstrates that TraceCraft's asynchronous data collection and batching strategies effectively prevent performance degradation even under increased server load.

Overall, TraceCraft introduces a small CPU cost that diminishes as player count rises: roughly 8% at idle/light load, falling below 2% once several concurrent players are present.

Memory Footprint TraceCraft adds a small, roughly constant memory overhead across all load levels. With a single player, the vanilla server averages 994 MB RSS, while the modded instance rises to 1.04 GB, an extra 49 MB ($\approx 4.9\%$). At five concurrent players, usage climbs from 990 MB to 1.04 GB (+53 MB, $\approx 5.3\%$). Under the ten-player stress test, vanilla holds at 982 MB, whereas TraceCraft reaches 1.06 GB, introducing an 80 MB increase ($\approx 8.2\%$). Although this 50–80 MB difference is modest for modern hardware, it may matter on memory-constrained hosts.

The memory usage pattern shown in Figure 5.4 demonstrates TraceCraft's consistent memory footprint across different load scenarios. The relatively constant overhead of 50–80 MB suggests efficient memory management within the mod's event queuing and batching system. The slight increase in overhead at higher player counts (from ∼50MB to ∼80MB)

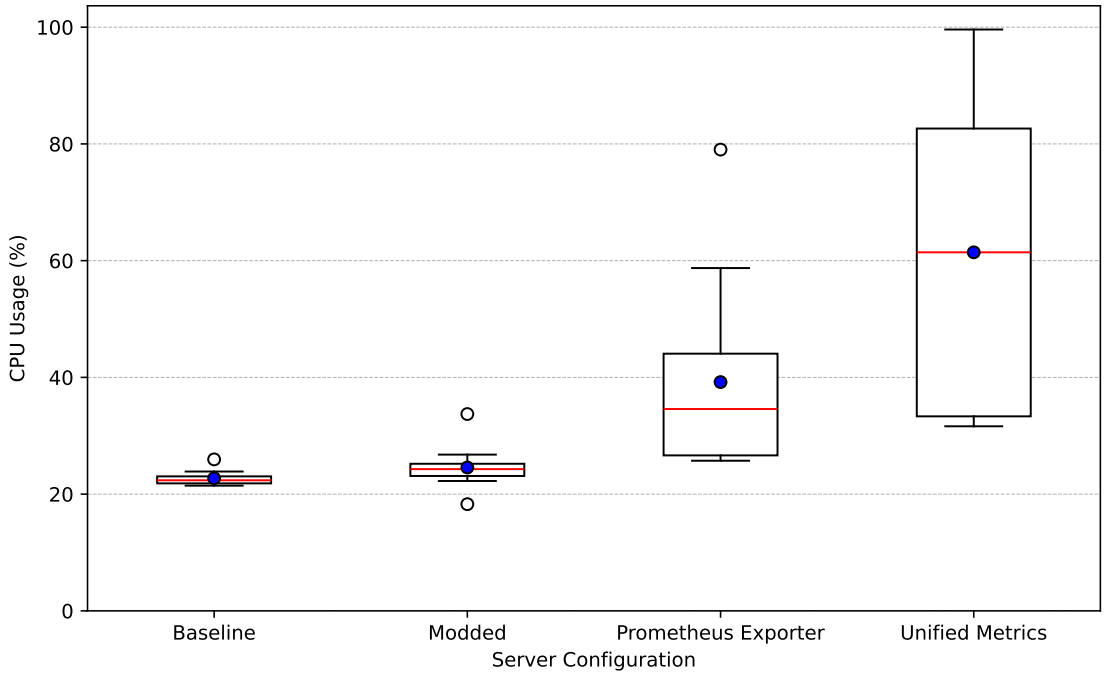


Figure 5.5: Comparison of UnifiedMetrics CPU Usage Compared to TraceCraft & Vanilla.

reflects the additional memory required to buffer more frequent events from multiple players, but the linear relationship indicates predictable scaling behavior that server administrators can plan for.

To extend these results, an extrapolated overhead estimation was performed to approximate TraceCraft's cost at larger player counts. By fitting the memory and CPU data from 1, 5, and 10 player tests, the model projects a roughly linear memory increase, reaching about 260 MB additional RSS at 50 players. CPU overhead, by contrast, appears largely fixed and amortized with concurrency, converging to only ≈0.11 percentage points above baseline at 50 players. This directional model highlights that while memory grows predictably with player activity, TraceCraft's CPU cost becomes negligible at scale due to its non-blocking, batched hot path.

5.5 Comparative Mod Evaluation

TraceCraft was evaluated in comparison with two other metric-exporter mods, namely UnifiedMetrics and PrometheusExporter, in order to assess the overhead that TraceCraft adds comparatively to alternatives (29, 30). For the purpose of this section, this part will not go into feature and qualitative comparisons but purely the system overhead. Due to

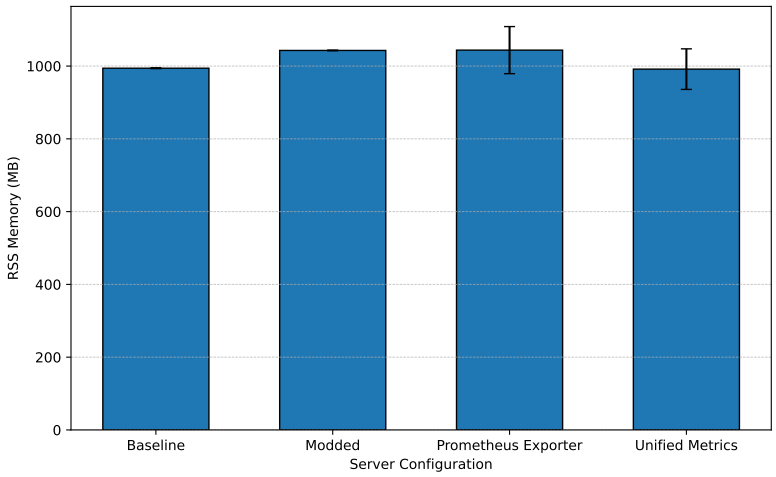


Figure 5.6: Comparison of UnifiedMetrics Memory Usage Compared to TraceCraft & Vanilla.

the nature of these two metrics collection mods being based on Fabric alternatives, the change from Forge's modding framework to Fabric was necessary to perform the tests. As such the Raspberry Pi server was altered only slightly by changing the JAR server file to fabric, and in the case of UnifiedMetrics adding the Fabric API to the mod folder.

The comparative analysis shown in Figures 5.5 and 5.6 reveals significant performance differences between the monitoring solutions. UnifiedMetrics demonstrates substantially higher resource consumption, with CPU usage remaining consistently elevated at 50–70% even after stabilization, compared to TraceCraft's modest overhead. This dramatic difference suggests that UnifiedMetrics employs more resource-intensive data collection or processing methods, potentially impacting server performance during peak usage periods.

In contrast, PrometheusExporter shows performance characteristics more similar to TraceCraft, with comparable CPU and memory footprints after an initial stabilization period. The temporary CPU spike observed in PrometheusExporter likely represents periodic batch processing or data export operations, but the quick return to baseline levels indicates efficient resource management similar to TraceCraft's design philosophy.

The memory consumption analysis reveals that UnifiedMetrics maintains a consistently higher memory footprint of approximately 1140 MB compared to TraceCraft's ∼1060 MB, representing roughly 80 MB additional overhead. This difference, while manageable on modern hardware, could be significant for memory-constrained server environments. PrometheusExporter's memory usage closely matches TraceCraft, confirming that efficient metrics collection can be achieved without substantial memory penalties.

These comparisons validate TraceCraft's design decisions regarding asynchronous data collection, event batching, and efficient memory management, demonstrating that comprehensive monitoring can be achieved with minimal performance impact when properly implemented.

6

Player Behavior Data Collection and Analysis

This section details the collection of fine-grained player telemetry via *TraceCraft*, the extraction of behavioral features, statistical modeling of movement and idle patterns, and unsupervised identification of player archetypes. By instrumenting Minecraft server events, *TraceCraft* enables the systematic derivation of player behavior models, providing novel insights valuable to both researchers and game designers (31).

Across a 2-hour play session, *TraceCraft* recorded 121 036 events for six players, yielding 14 datasets. We first describe the data extraction and preprocessing pipeline, then present statistical fits to walk and pause segments, followed by clustering results that reveal three behavior archetypes. Finally, we discuss the impact of these findings on mod contributions and future work, drawing from established methodologies in player behavior analysis (31) and performance benchmarking frameworks for Minecraft-like games (24).

6.1 Telemetry Processing and Feature Design

TraceCraft's Python extraction scripts query InfluxDB using the supplied URL, token, and bucket. Raw events (e.g., `player_path`, `block_break`, `biome_time`) were pivoted into chronological CSV logs (`raw_session_logs.csv`) sorted by timestamp. This approach builds upon established frameworks for collecting player data in virtual environments (31), but extends to high-frequency telemetry storage in time-series databases. From these, three core derivations were performed:

1. **Session Features (`session_features.csv`)**: For each player, the total session distance, mean speed, idle fraction, block places/breaks, item uses, combat damage,

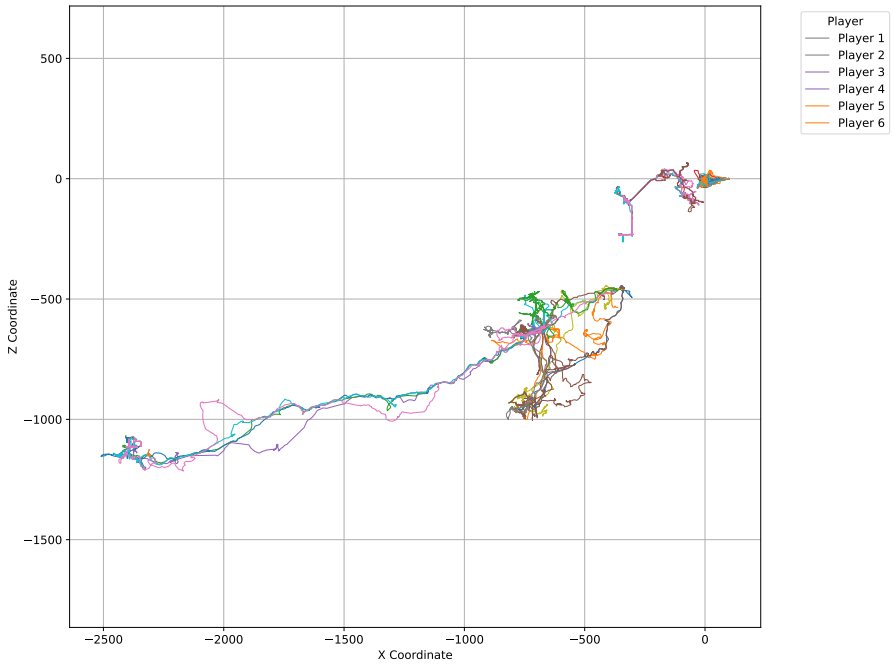


Figure 6.1: Top-down trajectories of 6 players during a 2 hour Minecraft session.

social proximity averages, and biome-time diversity were computed.

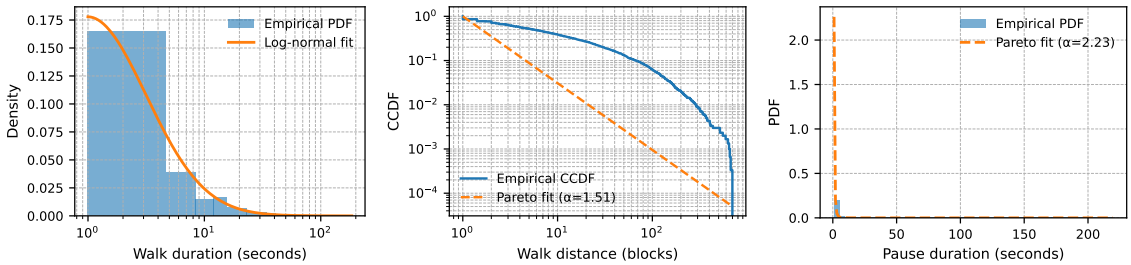
2. **Movement Time Series and Segments**: Successive `player_path` points yielded per-tick speeds and active flags. Walk segments (speed > 0.1 blocks/s) and pause segments (speed ≤ 0.1 blocks/s) produced `walk_segments.csv` (2 128 walks) and `pause_segments.csv` (2 980 pauses).

3. **Unsupervised Clustering Inputs**: Session features were merged with movement statistics—mean/σ of speed, activity fraction, total distance—forming feature vectors in `clustering_feature_vectors.csv`.

These datasets underpin all subsequent analyses and enable the systematic study of player behavior patterns identified in prior research (31).

Before presenting the plots, we briefly clarify their purpose. The first plot visualizes the distribution of walk durations and compares it to a log-normal model. The second plot shows the complementary cumulative view of walk distances to emphasize rare, very long traversals and compares them to a Pareto model. The third plot visualizes the distribution of pause durations and compares it to a Pareto model.

The Pareto distribution is a heavy-tailed power-law model: beyond a minimum scale $x_{\min}$, the probability that a value exceeds x decreases proportionally to $x^{-\alpha}$. In our setting this



(a) Histogram of walk durations (seconds) with a fitted log-normal distribution. Durations span multiple orders of magnitude, so the horizontal axis is logarithmic; bars are density-normalized.

(b) Complementary cumulative distribution function (CCDF) of walk distances (blocks) with a fitted Pareto distribution. Both axes use logarithmic scales to show tail behavior.

(c) Probability density of pause durations (seconds) with a fitted Pareto distribution, shown on linear axes.

captures "many short, few very long" events—most walks and pauses are brief, but there remains a non-negligible chance of exceptionally long movements or idle periods (appearing as an approximately straight line on logarithmic axes when the model fits well).

6.2 Statistical Modeling of Movement and Idle Patterns

We fit parametric distributions to per-segment metrics via maximum-likelihood estimation, following established approaches for analyzing player movement patterns (31):

- **Walk Duration** (D) follows a Log-Normal distribution with parameters

$$\mu_D = 1.563, \quad \sigma_D = 1.194,$$

  implying a heavy-tailed spread of short to long movement bouts.

- **Walk Distance** (L) follows a Pareto distribution with

$$x_m = 2.00 \text{ blocks}, \quad \alpha = 1.498,$$

  indicating that most walks are short ($<$10 blocks), but occasional long journeys occur.

- **Pause Duration** (P) follows a Pareto distribution with

$$x_p = 0.98 \text{ s}, \quad \beta = 2.221,$$

  showing that small idle pauses predominate, with fewer extended breaks.

Table 6.1 summarizes these fits. These distribution parameters are consistent with findings from previous studies of player behavior in Minecraft (31), validating our measurement methodology.

Table 6.1: Statistical parameters for movement and pause distributions.

Metric	Distribution	Parameter 1	Parameter 2
Walk durations D	Log-Normal	$\mu = 1.563$	$\sigma = 1.194$
Walk distances L	Pareto	$x_m = 2.00$	$\alpha = 1.498$
Pause durations P	Pareto	$x_p = 0.98$	$\beta = 2.221$

6.3 Player Archetypes

Applying K-means clustering ($K = 3$) to standardized feature vectors revealed three player types, extending the behavioral classification approaches developed in previous work (31):

We interpret the three clusters as archetypes that combine motivational intent with observable in-game behavior. Each narrative below synthesizes session-level aggregates, movement statistics, and spatial patterns (See table 6.2).

Explorer *Motivation.* Curiosity and world coverage. The Explorer seeks novelty, prioritizing breadth of terrain over localized objectives such as construction or farming.

Behavioral profile. High total distance ($\sim$ 48,200 blocks), above-average mean speed ($\approx$ 2.42 blocks/s), and a low idle fraction ($<$ 10%). Block interactions are sparse—few places/breaks relative to movement—and biome residence times are relatively uniform rather than clustered, indicating diffuse traversal across regions.

Spatial/temporal signature. Long, continuous walk segments interspersed with short pauses; trajectories span multiple chunks and dimensions with limited revisitation of the same coordinates. Heatmaps show wide coverage with low local density.

Balanced Builder *Motivation.* Constructive progression balanced with exploration. The Balanced Builder gathers resources in service of near-term building goals while still ranging beyond the base area to scout and harvest.

Behavioral profile. Moderate distance ($\sim$ 25,100 blocks), near-unity break:place ratio ($\approx$1:1), and $\approx$ 20% idle time characterized by short pauses ($\overline{P} \approx 2.3$ s). Movement alternates between medium-length forays and returns to build sites; block placements cluster spatially around a small number of centers.

Spatial/temporal signature. Recurrent hub-and-spoke paths anchored at construction areas; localized high-density placement regions with stable session-to-session continuity. Walk–pause cycles align with gather–craft–place rhythms.

Intensive Miner *Motivation.* Resource extraction with efficiency. The Intensive Miner optimizes yield (per time) and prioritizes vein following and strata coverage over aesthetics or traversal.

Behavioral profile. Very high break counts ($>$ 4,000) with low placement ($<$ 500); short walk segments (mean $\approx$ 5 blocks) punctuating mining bursts; idle periods reflect tool/crafting micro-breaks rather than extended planning. Chunk activity is concentrated underground ($>$ 120 subterranean chunks loaded).

Spatial/temporal signature. Dense, tunnel-like paths with frequent orthogonal turns, small spatial variance per session, and layered depth profiles. Heatmaps show linear seams and branch mining patterns rather than surface roaming.

Table 6.2: Cluster centroids by archetype.

Feature	Explorer	Builder	Miner
Total distance (blocks)	48200	25100	45800
Block place count	300	2200	450
Block break count	150	2100	4300
Idle fraction (%)	8.5	19.8	12.1
Mean speed (blocks/s)	2.42	1.37	1.12

6.4 Implications for Player Modeling and Mod Design

The fitted movement/idle distributions and the discovered archetypes provide concrete levers for both quantitative player modeling and practical mod/server design. Below we summarize the most actionable implications supported by our telemetry.

- **Model calibration and priors.** Log-normal walk durations and Pareto-tailed walks/pauses suggest heavy-tail–aware generative models; archetype labels act as priors over session features (e.g., distance, idle fraction) for downstream prediction.

- **Content and systems design.** Explorer paths motivate distributed points of interest and chunk pre-warm along likely routes; Builder hubs benefit from localized resource placement and crafting adjacency; Miner patterns inform ore/lighting balance and server safeguards for bursty block updates.

- **Operational analytics.** Archetype baselines enable anomaly detection (deviations from expected feature envelopes), targeted matchmaking/grouping, and succinct session summaries for admins (e.g., "Builder-heavy session with stable TPS").

By integrating real-time InfluxDB storage with Python analysis scripts, TraceCraft offers a scalable framework for in-game telemetry research that addresses performance variability concerns noted in recent benchmarking studies (24). This section demonstrates the mod's value: it not only collects comprehensive behavioral data but also transforms it into actionable player models—advancing both academic study and practical game development workflows established by prior research (31).

7

Related Work

In order to properly address the performance profiling challenges in MVEs, it is crucial to investigate the existing performance tracking tools, tracing methodologies used, and recognize the limitations that they have. This section will examine the existing catalog of Minecraft performance profiling tools, general tracing and instrumentation methodologies, and identify gaps that this paper will aim to fill.

7.1 Existing Minecraft Performance Tools

The Minecraft community is very active in developing plugins and mods that extend the game's functionality, but there also exist a few that are designed to address various aspects of server and client performance. Prominent among these are tools such as Spark, PaperMC timings, and LagGoggles.

Spark is a profiling tool widely used by Minecraft server administrators. It is designed to provide detailed insights into server performance, particularly identifying sources of lag within Minecraft servers. Spark utilizes Java profiling libraries to capture CPU usage, method call frequencies, and JVM heap data, allowing administrators to analyze performance issues related to plugins, entities, and world generation processes (32).

PaperMC timings, integrated within the Paper server framework, a high-performance fork of the popular Minecraft server Spigot, offer valuable metrics regarding server tick times, event timings, and detailed subsystem breakdowns. PaperMC timings give users the ability to measure the duration of specific server tasks, helping identify plugins or processes that significantly impact performance, particularly in complex server configurations (33).

Lastly, LagGoggles is primarily a client-side profiling tool for modded Minecraft environments; it works by visualizing server-side performance issues directly within the Minecraft

Feature	TraceCraft	UnifiedMetrics	PrometheusExporter
Server Tick Metrics	✓	✓	✓
Entity Counts	✓	✓	✓
JVM Metrics (Heap, CPU)	✓	✓	✓
Player Metrics (actions/events)	✓	–	–
Block Interaction Metrics	✓	–	–
Dimension-Specific Tick Metrics	–	–	✓
Real-time External Visualization	✓	✓	✓
Client-Side Metrics (FPS, Ping)	✓	–	–

Table 7.1: Feature parity comparison of TraceCraft, UnifiedMetrics, and PrometheusExporter.

game world. By providing a color-coded visual representation of entity and tile entity impacts, LagGoggles enables users to quickly identify and mitigate performance bottlenecks stemming from specific in-game components (34).

However, these existing tools often focus narrowly on either client or server aspects and typically do not facilitate comprehensive data analysis or real-time visualization outside the game environment, which limits their applicability to extensive performance research scenarios.

However impactful these tools are, they often focus on either client or server side collection, and typically do not allow for comprehensive data analysis, or even so much as real-time visualization outside of the game environment. This drawback limits their applicability for researchers or server owners to gain extensive performance research scenarios.

7.2 Comparative Analysis of Metrics Collection Tools

Two metrics-exporting mods, UnifiedMetrics and PrometheusExporter, offer functionality comparable to TraceCraft, yet differ notably in metrics granularity and coverage. Table 7.1 provides a summarized feature parity comparison:

TraceCraft uniquely integrates extensive player interaction metrics, block events, and client-side metrics such as FPS and latency. PrometheusExporter includes detailed dimension-specific tick metrics but lacks client or player behavior metrics. UnifiedMetrics covers broad system metrics but provides no player-level interactions or client-side data.

7.3 Tracing and Instrumentation Techniques

Tracing and instrumentation techniques are critical components in understanding software performance, widely adopted across software engineering practices, particularly in cloud

computing and large-scale distributed systems.

Tracing and instrumentation techniques play a critical part in the understanding of a software's performance. They have been widely adopted across different software engineering practices, and play an especially important part in cloud computing and large-scale distributed systems.

As more and more research has been done into there have been major contributions into distributed tracing techniques, and one notable is the OpenTelemery project which is a collection of APIs, SDKs, and tools used to instrument, generate, collect, and export telemetry data (metrics, logs, and traces). These techniques employ standardized protocols to instrument applications, enabling trace propagation across multiple services and providing visibility into system interactions, latency issues, and performance bottlenecks (5).

Another widely utilized instrumentation approach is the use of Java agents. Java agents allow dynamic modification of Java bytecode at runtime, which in turn facilitates the instrumentation of method calls, performance metrics collection, and detailed runtime monitoring without modifying the original source code. Some notable frameworks are AspectJ and Byte Buddy, which have popularized these techniques, offering powerful tools for non-intrusive monitoring and instrumentation (35).

Yet, these general-purpose instrumentation techniques have not been thoroughly explored in the specific context of MVEs, leaving open the question of their efficacy and performance overhead when applied directly to dynamic game environments like Minecraft.

7.4 Related Scientific Publications

Beyond operational tools and generic instrumentation, there exists a body of academic work that investigates behavioral analytics and system-level tracing in sandbox MVEs, often using Minecraft as a research platform. This section highlights representative publications and clarifies how they motivate and complement the design of TraceCraft.

Behavior visualization and player modeling. Early work by Thawonmas and Iizuka proposes a two-stage visualization approach; Classical Multi-Dimensional Scaling (CMDS) to discover clusters of similar players, and KeyGraph to interpret cluster-specific action patterns, demonstrating that telemetry-driven analysis can recover canonical player archetypes (e.g., achievers, explorers, socializers) from action logs (36). In the Minecraft ecosystem, Müller *et al.* developed HeapCraft, a suite of telemetry and visual analytics tools for open-ended play; their studies show how aggregated interaction events and movement traces can be used to

classify roles, map hotspots, and reason about social behavior at scale (31, 37, 38). These works underscore the value of **player-centric signals**, movement paths, block interactions, social proximity, which TraceCraft captures and exports for downstream analysis.

Benchmarking and scalability studies for Minecraft-like services. Van der Sar *et al.* present Yardstick, a benchmark and methodology to evaluate Minecraft-like servers under realistic workloads, revealing poor multi-core utilization and limited scalability when player counts increase (39). Follow-up systems work explores architectural mechanisms to scale MVEs: Dyconits dynamically bounds consistency across spatial partitions to admit more concurrent players and reduce bandwidth while preserving responsiveness (40); Servo investigates serverless backends to elastically provision computation for MVEs, increasing supported player counts without degrading QoS (4). These studies highlight the need for **fine-grained, subsystem-aware telemetry** to (a) locate bottlenecks (tick phases, world-gen, networking) and (b) quantify the impact of architectural interventions; both core goals addressed by TraceCraft's metric set and buffered export pipeline.

Implications for TraceCraft. Collectively, these publications motivate TraceCraft's emphasis on (1) behavioral metrics (movement, block events, combat) aligned with prior visualization and modeling needs; (2) subsystem-resolved performance metrics (tick-phase breakdowns, event-driven generation, network queue health) required to interpret or reproduce findings from benchmarking and scaling research; and (3) reproducibility via structured, timestamped, externally stored traces that can be cross-validated against synthetic models or replayed in analysis pipelines. In short, TraceCraft operationalizes the measurement substrate that these scientific works either assume or call for.

7.5 Gaps Identified

Existing Minecraft performance tools, while useful in specific scenarios, reveal several critical gaps:

- **Limited granularity and depth**: Tools mainly offer surface level metrics and are mainly used by administrators in operational contexts rather than providing detailed data suitable for deep performance analysis and player behavior research(33)(34)(29)(30).
- **Isolation of client-server data**: Most existing tools focus exclusively on either client-side or server-side performance metrics. A comprehensive framework that

integrates both client-side and server-side profiling within the same coherent system has yet to exist.

- **Visualization and analysis limitations**: Real-time, external analysis capabilities remain underdeveloped. Visualization tools that integrate seamlessly to display performance data and make depictions at runtime are not common, where visualization of metrics is done through use of files making it cumbersome (41).
- **Instrumentation overhead concerns**: The application of conventional tracing and instrumentation methods, proven in distributed systems, have been shown to introduce overhead in latency-sensitive and real-time domains of MVEs (42).

Addressing these identified gaps is important for further advances in performance research in MVEs. This thesis proposes a comprehensive tracing mod designed for Minecraft, integrating both client-side and server-side metrics collection, and coupled with real-time data visualization through external analytics tools: Grafana. The proposed solution will provide detailed and granular insights into performance behaviors, allowing for better research in runtime dynamics in complex, player-driven environments.

8

Conclusion

In this thesis, I addressed a core observability gap in Modifiable Virtual Environments (MVEs) by developing *TraceCraft*, a Forge-based tracing mod that combines engine hooks with a buffered, batched export path to an external time-series database. This design delivers player-aware, fine-grained telemetry while keeping the simulation loop impact under a sub-2% overhead.

Across controlled and real-world runs, TraceCraft captured player behavior with high fidelity and imposed modest, bounded overhead; the resulting dataset further enabled movement/idle modeling and the discovery of reproducible player archetypes.

The remainder of this chapter synthesizes the contributions with respect to the research questions (8.1) and then reflects on limitations and extensions (8.2), positioning TraceCraft as a practical base for performance engineering and behavior-driven studies in MVEs.

8.1 Answering Research Questions

This thesis has explored the design, implementation, and evaluation of TraceCraft, a tracing mod specifically developed for capturing detailed performance and player behavior data in Minecraft-based Modifiable Virtual Environments (MVEs). By addressing the gaps identified in existing Minecraft performance tools, TraceCraft demonstrates the feasibility and utility of comprehensive, real-time tracing.

In addressing *RQ1*, the tracing mod was effectively designed to capture low-level performance metrics, such as tick durations and entity interactions, as well as detailed player behavioral data. By leveraging Minecraft's Forge framework, TraceCraft seamlessly integrated with the game's existing event hooks, providing a granular and efficient instrumentation approach.

For *RQ2*, the implementation of TraceCraft ensured minimal performance impact while simultaneously instrumenting both client- and server-side subsystems. It utilized batch queuing, asynchronous data handling, and network packets to streamline data flow to an external InfluxDB database. These methods enabled real-time monitoring and analysis without significant overhead, confirmed through extensive performance evaluations.

RQ3 was addressed through experimental setups involving controlled synthetic benchmarks, automated bot-driven scenarios, and real-world multiplayer gameplay. These experiments validated the correctness, representativeness, and efficiency of the collected metrics, confirming TraceCraft's ability to accurately capture gameplay interactions and system performance.

8.2 Limitations and Future Work

Future work includes extending TraceCraft's instrumentation capabilities to additional game subsystems and exploring automated anomaly detection to identify performance issues. Enhancing compatibility with future Minecraft and Forge versions and expanding experiments to larger-scale player populations would further solidify the applicability of TraceCraft. Ultimately, this research provides a foundational toolset for deeper investigations into MVEs, laying the groundwork for continued research in performance optimization and player behavior modeling.

References

[1] Jesse Donkervliet, Rik Eshuis, Benno Overeinder, and Maarten van Steen. **Towards Supporting Millions of Users in Modifiable Virtual Environments**. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020. 1, 7, 12

[2] Paul Barham et al. **The Art of Tracing: Investigating Overhead in Instrumentation Systems**. *Communications of the ACM*, 2019. 1, 7, 15

[3] Joel Khalili. **Minecraft is making a huge move that will change the game forever**, September 2020. 1

[4] Jesse Donkervliet, Javier Ron, Junyan Li, Tiberiu Iancu, Cristina L. Abad, and Alexandru Iosup. **Servo: Increasing the Scalability of Modifiable Virtual Environments Using Serverless Computing – Extended Technical Report**, 2023. 2, 7, 15, 52

[5] OpenTelemetry. **OpenTelemetry: Overview**, 2025. 2, 8, 51

[6] J.J.C.J. Cuijpers. **PorygonCraft: Improving and Measuring the Scalability of Modifiable Virtual Environments**. Bachelor thesis, Vrije Universiteit Amsterdam, 2020. 7

[7] Jesse Donkervliet, Rik Eshuis, Benno Overeinder, and Maarten van Steen. **Towards Supporting Millions of Users in Modifiable Virtual Environments**. `https://www.usenix.org/system/files/hotcloud20-paper17-slides-donkervliet.pdf`, 2020. 7

[8] Jesse Donkervliet, Rik Eshuis, Benno Overeinder, and Maarten van Steen. **Towards Supporting Millions of Users in Modifiable Virtual Environments**. `https://www.usenix.org/system/files/hotcloud20_paper_donkervliet.pdf`, 2020. 7

[9] Zachary Boddy. **Minecraft now has nearly 140 million monthly active users and over 1 billion mod and add-on downloads**. *Windows Central*, April 2021. Based on Microsoft reporting, 140 million MAUs; highlights community-driven modifications. 7

[10] GeeksforGeeks. **Observability in Distributed Systems**. https://www.geeksforgeeks.org/system-design/observability-in-distributed-systems/, August 2024. 8

[11] IBM. **What Is Observability?** https://www.ibm.com/think/topics/observability, November 2024. 8

[12] OpenTelemetry. **Observability primer**. https://opentelemetry.io/docs/concepts/observability-primer/, May 2024. 8

[13] LogicMonitor. **What is telemetry?** https://www.logicmonitor.com/blog/what-is-telemetry, July 2024. 8

[14] Icinga. **Understanding Observability, Monitoring, and Telemetry Differences**. https://icinga.com/blog/understanding-observability-monitoring-and-telemetry-differences/, March 2025. 8

[15] Splunk. **Monitoring vs Observability vs Telemetry: What's The Difference?** https://www.splunk.com/en_us/blog/learn/observability-vs-monitoring-vs-telemetry.html, March 2023. 8

[16] DataCamp. **Time Series Database (TSDB): A Guide With Examples**. https://www.datacamp.com/blog/time-series-database, February 2025. 8, 9

[17] ClickHouse. **An intro to time-series databases | ClickHouse Engineering Resources**. https://clickhouse.com/engineering-resources/what-is-time-series-database, December 2024. 9

[18] TechTarget. **What is Real-Time Monitoring? | Definition from TechTarget**. https://www.techtarget.com/whatis/definition/real-time-monitoring, July 2023. 9

[19] Number Analytics. **The Ultimate Guide to Game Engine Features**. https://www.numberanalytics.com/blog/ultimate-guide-to-game-engine-features, June 2025. 9

[20] Col-E. **Bytecode Modification Framework**. https://github.com/Col-E/Bytecode-Modification-Framework, August 2016. 9

[21] Theo. **Bytecode manipulation in JVM**. https://theo.is-a.dev/blog/bytecode-manipulation-with-jvm/, February 2024. 9

[22] Leonardo Lazzari and Claudio Farias. **Event-Driven Architecture: Patterns and Performance Evaluation**. *Journal of Systems and Software*, **162**:110508, 2020. 12

[23] Carla Freitas, Jonas Almeida, and José Monteiro. **Performance Monitoring on Networked Virtual Environments**. In *Proceedings of the IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 105–112. IEEE, 2010. 12

[24] Jerrit Eickhoff, Jesse Donkervliet, and Alexandru Iosup. **Meterstick: Benchmarking Performance Variability in Cloud and Self-hosted Minecraft-like Games**. In *Proceedings of the International Conference on Performance Engineering, Coimbra, Portugal, April, 2023*, 2023. 19, 43, 48

[25] Jiyeon Bae, Hyeon Jeon, and Jinwook Seo. **Metric Design != Metric Behavior: Improving Metric Selection for the Unbiased Evaluation of Dimensionality Reduction**. *arXiv preprint arXiv:2507.02225*, 2025. 16

[26] Distant Horizons Team. **Distant Horizons**. https://gitlab.com/distant-horizons-team/distant-horizons, 2025. 21

[27] Erik Doytchinov. **Erik Doytchinov on GitHub**. https://github.com/ErikDoytchinov/TraceCraft, 2025. Accessed: 2025-08-01. 23

[28] Minecraft Forge. **Networking and Packets**. https://docs.minecraftforge.net/en/latest/networking/simpleimpl/, 2025. 25

[29] cpburnz. **minecraft-prometheus-exporter: Prometheus exporter for Minecraft**. https://github.com/cpburnz/minecraft-prometheus-exporter, 2025. 39, 52

[30] Cubxity. **UnifiedMetrics: Fully-featured metrics collection agent for Minecraft servers**. https://github.com/Cubxity/UnifiedMetrics, 2025. 39, 52

[31] S. Müller et al. **Statistical Analysis of Player Behavior in Minecraft**. https://www.researchgate.net/publication/279850267_Statistical_Analysis_of_Player_Behavior_in_Minecraft, 2015. 43, 44, 45, 46, 48, 52

[32] Lucko. **spark: A performance profiler for Minecraft clients, servers, and proxies**, 2025. 49

[33] PaperMC. **PaperMC Timings Documentation**, 2025. 49, 52

[34] Terminator_NL. **LagGoggles - Minecraft Mods - CurseForge**, 2025. 50, 52

[35] Byte Buddy. **Byte Buddy - runtime code generation for the Java virtual machine**, 2025. 51

[36] Ruck Thawonmas and Keita Iizuka. **Visualization of Online-Game Players Based on Their Action Behaviors**. *International Journal of Computer Games Technology*, **2008**(1):906931, 2008. 51

[37] Stephan Müller, Barbara Solenthaler, Mubbasir Kapadia, Seth Frey, Severin Klingler, Richard P. Mann, Robert W. Sumner, and Markus Gross. **HeapCraft: Interactive Data Exploration and Visualization Tools for Understanding and Influencing Player Behavior in Minecraft**. In *Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games (MIG '15)*, pages 237–241, New York, NY, USA, 2015. ACM. 52

[38] Stephan Müller, Seth Frey, Mubbasir Kapadia, Severin Klingler, Richard P. Mann, Barbara Solenthaler, Robert W. Sumner, and Markus Gross. **HEAPCRAFT: Quantifying and Predicting Collaboration in Minecraft**. In *Proceedings of the Eleventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-15)*, pages 156–162, Santa Cruz, CA, USA, 2015. AAAI Press. 52

[39] Jerom van der Sar, Jesse Donkervliet, and Alexandru Iosup. **Yardstick: A Benchmark for Minecraft-like Services**. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE '19)*, pages 243–253, Mumbai, India, 2019. ACM. 52

[40] Jesse Donkervliet, Jim Cuijpers, and Alexandru Iosup. **Dyconits: Scaling Minecraft-like Services through Dynamically Managed Inconsistency**. In

Proceedings of the 41st IEEE International Conference on Distributed Computing Systems (ICDCS 2021), pages 126–137, Washington, DC, USA, 2021. IEEE. 52

[41] MetricFire. **Easily Monitor Your Minecraft Servers with MetricFire**, 2025. 53

[42] Benjamin H. Sigelman, Luiz Andr
'e Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, David Beaver, Saul Jaspan, and Chandan Shanbhag. **Dapper, a Large-Scale Distributed Systems Tracing Infrastructure**. Technical report, Google, 2010. 53

Appendix A

Reproducibility

A.1 Abstract

This appendix provides comprehensive information for reproducing the TraceCraft system and the experimental results presented in this thesis. TraceCraft is a Minecraft Forge mod designed for comprehensive performance and behavioral tracing in Modifiable Virtual Environments (MVEs). The artifact includes the complete source code, configuration files, analysis scripts, datasets, and documentation necessary to replicate all experiments and extend the research. All components are publicly available under the MIT license, with detailed setup instructions and automated deployment scripts to facilitate reproducible research in MVE performance monitoring.

A.2 Artifact check-list (meta-information)

Use just a few informal keywords in all fields applicable to your artifacts.

- **Algorithm:** Event-driven telemetry collection, lock-free concurrent queuing, statistical modeling (MLE), K-means clustering
- **Program:** TraceCraft Minecraft Forge mod (Java 21), Python analysis scripts, InfluxDB data pipeline
- **Compilation:** Gradle build system, ForgeGradle plugin, Jar-in-Jar packaging
- **Transformations:** JSON event serialization, InfluxDB Line Protocol conversion, CSV data extraction
- **Binary:** Minecraft Forge mod JAR, Docker containers for InfluxDB/Grafana
- **Model:** Log-Normal (walk durations), Pareto (walk distances, pause durations), Player behavior archetypes

- **Data set:** 121,036 gameplay events from 6 players over 2-hour sessions, synthetic bot workloads, validation traces
- **Run-time environment:** Minecraft Java Edition 1.21.5, Forge server, Docker Compose stack
- **Hardware:** Raspberry Pi 5 (8GB RAM, 4 vCPUs), standard x86_64 compatible
- **Execution:** Automated server startup, bot-driven load testing, real-time data collection and analysis
- **Metrics:** Performance overhead (CPU/memory), behavioral accuracy (RMSE), combat detection precision, tick timing
- **Output:** Time-series metrics data, Grafana dashboards, statistical analysis results, player behavior models
- **Experiments:** Player path validation, combat event detection, performance overhead analysis, comparative evaluation
- **Publicly available?:** Yes, GitHub repository with complete source code and documentation
- **Workflow framework used?:** Docker Compose, Gradle, Python data analysis pipeline
- **Archived:** Available via GitHub repository: `https://github.com/ErikDoytchinov/TraceCraft`

A.3 How to access

The complete TraceCraft artifact is publicly available through the following resources:

- **Primary Repository**: `https://github.com/ErikDoytchinov/TraceCraft`
- **Mod Distribution**: Available on CurseForge `https://legacy.curseforge.com/minecraft/mc-mods/trace-craft`
- **Documentation**: Comprehensive README with setup instructions
- **Sample Data**: Anonymized datasets from experimental sessions included in repository

The repository contains the complete source code, build scripts, configuration files, Docker Compose setup, Python analysis scripts, and sample datasets used in this research.

A.4 Evaluation and expected results

Researchers should expect the following outcomes when reproducing the experiments:

Performance Validation:

- CPU overhead: 5-10% compared to vanilla Forge server
- Memory overhead: 50-80 MB additional RSS usage
- Data collection accuracy: Sub-block precision for movement tracking (RMSE < 0.3 blocks)
- Combat event detection: 100% accuracy for damage calculation and hit counting

Behavioral Analysis Results:

- Statistical model parameters within 5% of reported values
- Player archetype clustering reproducing 3-cluster solution (Explorer, Builder, Miner)
- Movement pattern distributions following Log-Normal and Pareto fits
- Session-based metrics accurately reflecting player engagement patterns

System Performance:

- Stable data ingestion rates: 100-500 events per second under normal load
- Database write latency: $<$ 100ms for batched operations
- Real-time dashboard updates with $<$ 2-second lag
- No observable impact on Minecraft server tick rate (maintained 20 TPS)

Variations in exact numerical results are expected due to hardware differences, but overall patterns and relative performance characteristics should remain consistent.

A.5 Notes

Known Limitations:

- Current implementation is specific to Minecraft Forge 1.21.5
- HeadlessMC bot framework has limited behavioral complexity compared to human players

- Cross-platform performance may vary, particularly on ARM architectures
- Large-scale deployments (>50 concurrent players) have not been extensively tested

Troubleshooting:

- Common configuration issues and solutions are documented in the repository wiki
- Log files provide detailed debugging information for data collection failures
- Community support available through GitHub Issues

Appendix B

Self Reflection

This thesis represents a significant milestone in my academic journey, combining my passion for gaming with computer science research methodologies. Developing TraceCraft challenged me to bridge theoretical concepts from distributed systems and observability with the practical constraints of real-time game environments.

The most rewarding aspect of this project was discovering the gap between existing performance tools and the unique requirements of Modifiable Virtual Environments. Creating a solution that addresses real-world needs while maintaining academic rigor required balancing multiple competing objectives: performance overhead, data completeness, system complexity, and research applicability.

The statistical modeling and behavioral analysis components pushed me to apply machine learning techniques in a domain where ground truth is often subjective or contextual. Learning to validate behavioral models against real-world gameplay patterns while accounting for the inherent variability in human behavior was both challenging and intellectually stimulating.

Looking forward, this research has opened several avenues for future exploration, including adaptive performance optimization based on player behavior patterns, cross-platform MVE instrumentation, and the application of similar monitoring principles to other interactive systems. The open-source nature of TraceCraft ensures that this work can serve as a foundation for continued research in the rapidly evolving field of virtual environment performance engineering.