Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

An Empirical Evaluation of Performance Variability in Serverless Modifiable Virtual Environments

Author: Junyan Li (2659783)

1st supervisor: prof. dr. ir. Alexandru Iosup
daily supervisor: ir. Jesse Donkervliet
2nd reader: dr. ir. Animesh Trivedi

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 26, 2021

"I am the master of my fate, I am the captain of my soul"
from Invictus*, by William Ernest Henley*

Abstract

Video gaming has been a mainstream and fastest-growing media over the last decade. Minecraft is one of the most popular games with its featured Modifiable Virtual Environment (MVE), which allows players to interact with the environment. However, despite its large number of players, previous research shows that it does not scale well. Serverless technology provides the advantage of automatic scaling, which has the potential to address the scalability challenges of MVEs.

In this thesis, we study how serverless technology benefits MVEs and how performance variability impacts the system. We design and implement a serverless MVE prototype based on Opencraft and Azure Cloud platform. We design and implement a benchmark suite to evaluate the serverless MVE prototype. We conduct both microbenchmarking on services hosted on cloud platforms and macrobenchmarking on the real system with emulated player behaviors. The experiments focus on four aspects: effectiveness of latency hiding policies, scalability, performance variability, and the worst cases perceived by players.

Our finding shows that serverless technology improves the scalability of MVEs. It benefits MVEs in terms of supporting more players by offloading heavy computation tasks to serverless functions. Our first step into using multiple server instances with serverless for one game world further improves the scalability. In the worst cases of specific events where the players have to wait for the results, the gaming experience is not significantly impacted. Furthermore, the performance variability is high on all tested cloud services. However, the high variability of serverless functions does not negatively impact the serverless MVE, when using effective latency hiding policies. Instead, the performance variability of the serverless MVE is mainly caused by the shared CPU resource of virtual machines and the internal process of MVEs.

Contents

List of Figures **v**

List of Tables **vii**

1 Introduction **1**

1.1 Research Questions . . . 3

1.2 Main Contribution . . . 4

2 Background **7**

2.1 Modifiable Virtual Environments . . . 7

2.1.1 MVE Client . . . 8

2.1.2 MVE Server . . . 8

2.2 Cloud Computing . . . 9

2.2.1 Serverless Computing . . . 10

2.2.2 Azure Cloud Platform . . . 11

2.2.3 Performance Variability . . . 12

3 Related Work **15**

4 Serverless MVE Design **17**

4.1 Requirements . . . 17

4.2 Design Overview . . . 17

4.2.1 Gateway . . . 18

4.2.2 MVE Server Pool . . . 18

4.2.3 Function Invokers . . . 19

4.2.4 Storage Client . . . 20

4.2.5 Serverless Functions . . . 20

4.2.6 Monitoring Service . . . 21

5 Serverless MVE Implementation **23**
- 5.1 Persistent Cloud Storage . . . 23
  - 5.1.1 Latency Hiding Policy for Region File . . . 25
- 5.2 Serverless Functions . . . 27
  - 5.2.1 Player Status Operation Function . . . 27
  - 5.2.2 Player Login Handler . . . 28
  - 5.2.3 Environment Simulation Function . . . 29
  - 5.2.4 Terrain Generation Function . . . 30
- 5.3 Gateway . . . 30
  - 5.3.1 Policy for Multiple Server Instances Routing . . . 32

6 Benchmark Suite **35**
- 6.1 Requirements . . . 35
- 6.2 System Overview . . . 36
  - 6.2.1 Benchmark Server . . . 36
  - 6.2.2 Benchmark Clients . . . 38
  - 6.2.3 MVE Thin Client . . . 38
- 6.3 Overall Benchmark Process . . . 39
- 6.4 Workload . . . 39
  - 6.4.1 Straight Walk Behavior . . . 40
  - 6.4.2 Random Behavior . . . 41
- 6.5 Metrics Collection . . . 42

7 Experimental Setup **43**
- 7.1 Experiment Overview . . . 43
- 7.2 Configuration . . . 44
  - 7.2.1 Serverless Function . . . 44
  - 7.2.2 Virtual Machines . . . 44
  - 7.2.3 Storage Account . . . 45
  - 7.2.4 MVE Server and Client . . . 45
- 7.3 Metrics . . . 45
  - 7.3.1 Latency: Using Cloud Services . . . 46
  - 7.3.2 Maximum Players: Scalability of the System . . . 46
  - 7.3.3 Tick Duration: Overload State . . . 47
- 7.4 Microbenchmarking . . . 47
  - 7.4.1 Function Runtime and Life-cycle Variability . . . 47

7.4.2 Blob Storage Latency Variability 48
7.4.3 Azure Storage and SQL End-to-end Latency 49
7.5 Macrobenchmarking 49
7.5.1 Region File Cache Policy 49
7.5.2 Increasing Workload for MVE Systems with Single Instance 50
7.5.3 Increasing Workload for Serverless MVE with Multiple Instances . . 50
7.5.4 Fixed Number of Players for Serverless MVE 51
7.5.5 Worst Cases of Player Perceived Latency 51

8 Evaluation 53
8.1 High Function Runtime Variability 54
8.2 High Latency Variability on Blob Storage 55
8.3 Premium Storage Account Outperforms Serverless SQL 58
8.4 Region File Cache Policy Successfully Hides Remote Reading Latency . . . 59
8.5 Serverless Improves the Scalability of MVE 61
8.6 High Variability of Serverless MVE 63
8.7 Serverless Functions Do Not Negatively Impact Serverless MVE 65
8.8 Worst Cases of Player Perceived Latency 66
8.9 Gateway with Multiple Instances Improves Scalability 68

9 Conclusion and Future Work 71
9.1 Conclusion 71
9.2 Future Work 72

References 75

A Technical Details 79

List of Figures

2.1 MVE System Model. 7

4.1 Serverless MVE Model. 18

5.1 Process of Region File Cache (Read). 25
5.2 Demonstration of Simple Distance Preloading Policy. 26
5.3 Process of Region File Cache (Write). 27
5.4 Flowchart of Gateway for Policy-Based Packet Forwarding. 31
5.5 Demonstration of Splitting Terrain into 2 and 4 MVE Instances. 32

6.1 Benchmark Suite Overview. 36
6.2 Walking Directions of Straight Walk Behavior. 40

7.1 Different Latency Metrics with Timestamp. 46

8.1 Function Performance Metrics (*TerrainGeneration*). 55
8.2 Latency of Events During Function Invocation (*TerrainGeneration*). 55
8.3 Latency of Downloading Player Data (646 B) from Blob Storage. 56
8.4 Latency of Downloading Region File (36.68 MB) from Blob Storage. 56
8.5 Percentage of Time Spent on Different Events when Downloading Files from Blob Storage. 57
8.6 End-to-end Latency to Retrieve the Same Data from Two Storage Back-ends (Premium Storage Account and Serverless SQL). 58
8.7 In-game Graphic When World Becomes Invisible to Players. 60
8.8 Tick Duration with Increasing Straight Walk Workload. 61
8.9 CPU Usage with Increasing Straight Walk Workload. 61
8.10 Memory Usage with Increasing Straight Walk Workload. 62
8.11 Packet Per Seconds with Increasing Straight Walk Workload. 62

8.12 Tick Duration with Increasing Straight Run Workload (8 Blocks / Second). 63
8.13 Number of Players Before the System Under Test is Overloaded with Random Behavior. 63
8.14 Line Plot of Tick Duration of All Iterations. 64
8.15 Line Plot of CPU Usage of All Iterations. 64
8.16 Box Plot of Tick Duration among All Iterations. 65
8.17 Box Plot of Function Duration among All Iterations. 65
8.18 End-to-end Latency of Events Measured at Client. 67
8.19 Maximum Number of Players with Increasing Number of Workers (Straight Walk Workload). 68
8.20 CPU Usage of 4 Servers with 1 Gateway. 69
8.21 CPU Usage of 4 Servers with 2 Gateways. 69

List of Tables

5.1 Details of Persistent Data on OpenCraft. 24

6.1 Possibility of Different Actions on Random Behavior. 41
6.2 An Overview of Benchmark Metrics. 42

7.1 An Overview of Microbenchmarking Experiments. 43
7.2 An Overview of Macrobenchmarking Experiments. 44
7.3 Specification of Azure Functions. 44
7.4 Specification of Virtual Machines. 45

8.1 Different Percentiles of Latency of Reading a Chunk from Region File When Using Different Storage Settings. All Values are Presented in Milliseconds. . 59

A.1 Details of Configuration File on Benchmark Suite. 80
A.2 Details of Exchanged Messages through TCP Socket on Benchmark Suite. . 80

1

Introduction

Video gaming has been a mainstream and fastest-growing media over the last decade [1]. In 2020, the yearly global market revenue of gaming was USD 173.7 billion [2]. Due to the impact of the COVID-19 pandemic, in-person and outdoor entertainment are limited, which leads to further growth in the gaming market, especially in online multiple games. With over 200 million copies sold and more than 131 million monthly active users [3], Minecraft has been one of the most popular games in the world with its Modifiable Virtual Environment (MVE), which allows users to interact and modify the virtual worlds in real-time. However, despite its large number of users, the game does not scale well, limited by its replicated game instances which do not exchange information with each other. Serverless technology is an event-driven programming model, which provides automatic scaling. In this thesis, we design and implement an MVE system using serverless technology. We conduct an empirical evaluation on it. We show that serverless improves the scalability of MVEs, and the high performance variability of cloud services does not negatively impact the MVE with proper latency hiding policies.

Minecraft is an online multiplayer sandbox game with an infinite terrain, which is generated through algorithms as players explore it. Minecraft provides players an MVE that allows them to interact with the objects in a virtual world, construct buildings, or make complex systems such as a computer [4]. Additionally, players can create customized content via modding programs. By providing players a large amount of freedom in creativity, Minecraft goes beyond entertainment and steps in other fields such as computer-aided design [5], education [6], and research [7].

Despite the increasingly large number of users in Minecraft and its awards, Yardstick [8], a benchmark for Minecraft games, shows that Minecraft services are poorly parallelized and do not scale well, by conducting real-world experiments on popular server distributions.

The game performance degrades with the increasing number of players. Under favorable conditions, only up to a few hundred of players can join the same game instance. As the most popular MVE, Minecraft server relies on creating new game instances which do not exchange states with each other for achieving scalability. As a result, the large user-base of Minecraft cannot play together in the same world, though the world is designed to be infinitely large. The players who join different game instances cannot see or interact with each other. The Minecraft-as-a-Service provider, Minecraft Realm, only allows up to ten concurrent players on the same game instance [9]. Similar problem exists on other MVE games. For example, Valheim, a MVE game published in 2021, limits the maximum player to ten due to synchronization and lag issues [10]. Compared to other modern massively multiplayer online games, such as World of Warcraft, Final Fantasy XIV, and EVE Online, which are capable of handling thousands of concurrent players in one game instance, MVE games put forward a motivation for increasing the capability of concurrent players in the same virtual world.

Motivated by the scalability limitation of Minecraft, which also exists in other MVE games, Donkervliet et al. [11] envision a new architecture for large-scale MVE games with serverless technology. They propose a model of services and deployments for MVEs as serverless systems that run as independently scheduled services, so that each part of the system can scale independently with the potential to support millions of users. Serverless is an event-driven programming model whose infrastructure and scalability are handled by cloud providers. It provides high-level abstractions of distributed computing elements, which allows developers to focus on business logic without much server- and resource-management burden [12]. The advantages of serverless technology have the potential to address the scalability challenges of MVEs.

By designing MVEs as serverless systems, new and unique challenges are posed for both providers and developers. In the aspect of development, for example, the additional layers of communication between components and cold start of functions can cause overhead, and may potentially slow down the overall performance when migrating to serverless architecture due to improper system design. The overhead issue can significantly impact applications that require strictly low latency, such as online games [13].

Additionally, the same performance is not always guaranteed for applications hosted in clouds. The reason is that it is difficult to guarantee resource usage in shared infrastructure among tenants, and the resource management and scheduling policies enforced by the providers may not be suitable for all kinds of applications. Previous studies show that performance variability is a widely observed phenomenon in cloud computing [14, 15].

Specifically, in serverless functions, Ginzburg et al. [16] demonstrate that the phenomenon is significant in AWS Lambda, a serverless service provided by Amazon. The performance variability affects the users' Quality of Experience (QoE). It remains an open research question to understand how the performance variability impacts MVEs as serverless systems running on the clouds.

1.1 Research Questions

In this work, our goal is to study the impact of performance variability on serverless MVEs by conducting real-world experiments. To this end, we propose three research questions (RQs).

RQ1 How to design a serverless MVE?

To the best of our knowledge, no serverless MVE currently exists. We first design a serverless MVE. We identify the important components in the game system and propose a high-level design of the system using serverless architecture.

An important challenge is to split the MVE into modules while meeting the game's QoS requirements. The latency requirement for most online games is strictly low, which seems to be incompatible with serverless functions because their latency can be high and variable due to cold starts [12] and communication overhead between different services. We consider latency in our design so that the game can meet player expectations.

RQ2 How to implement the prototype of such a system?

To conduct experiments on serverless MVE, we implement a serverless MVE prototype based on the design. There is no known universal method to translate designs into prototypes. Additionally, there is no publicly available method for migrating monolithic applications to serverless applications. We first explore the implementation of the existing game system, the programming model, and the services offered by the cloud provider. There are many vendor-specific frameworks with a wide selection of built-in components, which may result in significant differences in the performance of the system. Thus, we need to carefully choose services and components, and map our serverless MVE design to them. We modify the logic of the MVE game system and the communication models between different modules to fit the serverless platform.

RQ3 How to evaluate and measure the performance variability of such a system? Does a serverless architecture benefit an MVE?

To understand the performance variability of the system, we perform real-world experiments on it. There are two challenges. First, there is no universal way to evaluate performance variability. Second, currently no benchmark tool exists for serverless MVE systems. Our approach is to implement a benchmark suite that automatically deploys the serverless MVE, runs experiments, and captures metrics. The suite extends YardStick [8], which emulates players and submit workloads to server.

We run experiments to benchmark the monolithic system and the serverless prototype, and capture both system- and application-level metrics. To measure performance variability, we run multiple iterations for each experiment and discuss the deviation. To measure the scalability, we compare the maximum number of players before the server is overloaded, which is defined as consecutive tick duration over 50 ms. Finally, we study the overhead of using serverless technology by comparing the worst cases of player perceived performance.

1.2 Main Contribution

To answer these research questions, we make the following three-fold contribution.

1. We propose a novel design of a serverless MVE system (Section 4), considering the architecture of MVEs and components of serverless platforms to meet QoS requirements.

2. We implement a serverless MVE prototype based on Opencraft and run it on the Azure cloud platform (Section 5). Opencraft is an open-source Minecraft server for research on Massivizing MVE. Azure is a cloud platform from Microsoft and provides serverless functions and storage solutions.

3. We design and implement a benchmark suite to evaluate the serverless MVE (Section 6). We conduct an empirical evaluation (Section 7 and 8), including microbenchmarking on cloud services and macrobenchmarking on the serverless MVE prototype. The microbenchmarking result shows significant performance variability on serverless functions and cloud storage back-ends. The macrobenchmarking result shows that serverless technology improves the scalability of MVEs. Although the performance

variability is high on cloud services, it does not negatively impact the serverless MVE with proper latency hiding policies.

2

Background

In this chapter, we provide comprehensive background information on Modifiable Virtual Environments (MVEs) and its system model (Section 2.1). Next, we introduce the concept of serverless computing and its components (Section 2.2). Finally, we discuss performance variability of cloud services (Section 2.2.3).

2.1 Modifiable Virtual Environments

A modifiable Virtual Environment (MVE) is a real-time, online, multi-user environment that allows players to modify the objects in the virtual world, create new contents by connecting components, and interact with the world through programs [11].

We present in Figure 2.1 a system model of an MVE, with general services. Generally, an MVE system is designed with client-server architecture where players run a client software in their local devices, which periodically exchanges information with a remote server hosted in the cloud, through a network management service with Internet connection.

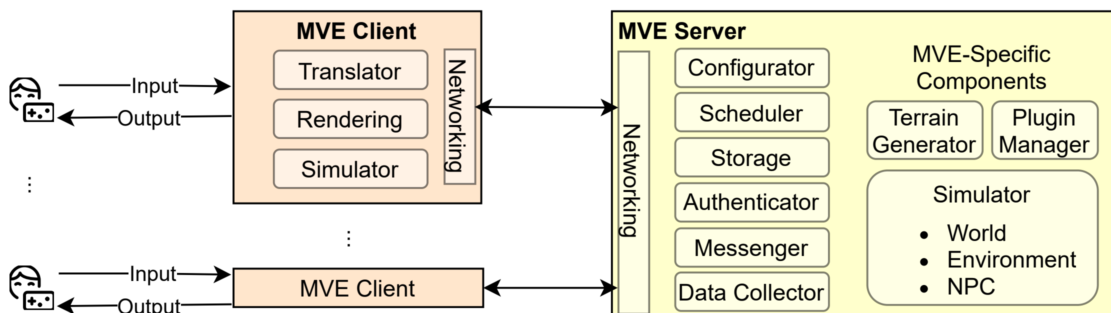

Figure 2.1: MVE System Model.

2.1.1 MVE Client

The MVE client accepts inputs from players, usually from keyboard and mouse, or controller. It translates the inputs into system commands, transmitted to the MVE server, renders graphic content, and provide output to players, usually to monitors. A simulator is maintained in the client, which provides instant feedback to players based on their inputs. The simulator periodically synchronizes with the server to maintain consistency on elements with other clients.

Similar to most games, there are two types of clients in MVEs, namely traditional clients and thin clients. A traditional client does heavy computation tasks in client end and renders graphic contents locally, thus it usually occupies storage for game elements and requires hardware acceleration support such as GPU under many scenarios. Thin client, on the other hand, is more lightweight and similar to video streaming clients. It is proposed with the development of Internet infrastructure and is usually used in remote rendering games, which are also known as cloud gaming, a term introduced as a market stunt by OnLive in 2011. The rendering tasks and most computation tasks are completed in remote server or middleware. The computation tasks in thin client usually only involve handling player's inputs and displaying the streamed contents.

The latency requirements are different for traditional and thin clients. A previous study [13] shows that the latency requirement for traditional game clients is strictly low, and the threshold is different depending on game genres. For example, first-person shooting games are the most sensitive game genre to latency. Third-person role-playing games can tolerant five times higher latency than first-person shooting games before performance degradation is observed by players. Games with thin clients are even more sensitive to latency. Third-person avatar games with thin clients can tolerant up to the same latency threshold as first-person shooting games with traditional clients [17], which are the most sensitive game genres in traditional gaming. Because of the obstacles in thin client, such as meeting stricter latency requirements and scheduling resources of remote GPUs, traditional clients still dominate the online game market.

2.1.2 MVE Server

The server handles most game logic, and performs heavy computation tasks. There are some common components as other game servers. Authenticator verifies each request for security reasons. Configurator loads configuration during server initialization, and updates configuration when necessary. Messenger delivers or broadcasts system and user-defined

messages to players. Storage service saves the system data persistently, such as world data, and player data. Lastly, data collector collects system usage data and saves it for debugging and analytic. The server is responsible for ensuring a consistent game state so that players who join the same game instance can see the same environment. Depending on the specific requirement, both strong consistency and eventual consistency models can be used.

A typical MVE server includes three MVE-specific components. First, Terrain Generator generates the terrain as players explore in the infinite world. As the world is designed to be infinite and can be explored freely, it cannot be fully loaded at once due to resource constraints such as bandwidth. Thus, the server should generate new or load existing terrain data when it is of concern to players. The generation algorithms should be designed in a way that consuming the least resource while not degrading players' experience. Second, the server provides feedback to players' behavior and simulates all elements in a virtual world, including changing environments such as climate, weather, and geography through continuous timers or triggers, and generating the behaviors of non-playable characters. The simulator on the server is periodically synchronized with the MVE client. Third, Plugin Manager loads mods written by players with pre-defined interfaces.

Generally, the server needs to complete most computation tasks within one game tick, which is a constant frequency of the game state updates. Otherwise, the unfinished tasks will delay the thread and slow down overall performance, directly impacting players' experience. A low tick rate results in low granularity for message processing and reduces the precision of the simulation. A high tick rate increases it but requires more resources. Overall, many aspects should be taken into consideration to design a proper MVE, and an empirical evaluation helps to understand the efficiency of the new design.

2.2 Cloud Computing

Cloud computing refers to on-demand computer system resources which provide some levels of abstraction to save the burden of direct management for users. It has been a commonly used solution for deploying online services for over a decade. Several service models exist in cloud computing, providing a different level of abstraction to satisfy some sets of business requirements. Infrastructure-as-a-Service (IaaS) abstracts underlying network infrastructure, physical servers and storage. By providing on-demand computer resources in the forms of virtual machines or containers, IaaS saves users the hassle and cost to manage IT infrastructure from self-hosted hardware, while giving users full control over operating

systems. In addition to infrastructure, Platform-as-a-Service (PaaS) abstracts operating systems and middleware, such as development tools, analytic tools, and database management. By providing a framework for developers to build cloud-based applications, PaaS reduces the coding effort and supports the full life cycle of web applications.

2.2.1 Serverless Computing

With the motivation of making PaaS more accessible, fine-grained, and affordable, serverless computing is born as an emerging paradigm of cloud service model [18]. One of the common ways to archive serverless computing is the Function-as-a-Service (FaaS) model. These two terms are highly overlapped and often mixed-use. We use in this thesis the term serverless.

Serverless is an event-driven programming model whose infrastructure and scalability are handled by cloud providers. It provides high-level abstractions of distributed computing elements, typically including workflow composition, function management, and resource orchestration [19]. By programming the system in serverless architecture with APIs provided by serverless platforms, developers only need to focus on business logic without much server- and resource-management burden [12]. In general, serverless is promising as it further reduces the complexity and cost of distributed applications development by outsourcing operational logic to cloud providers.

In practice, programming in serverless architecture basically follows the principles of microservice, which aims to design a system as a collection of loosely coupled services (serverless functions), and process communication between services via the network, such as HTTP. Existing systems implemented in microservice architectures can be reused in serverless with minor modification. With microservice, developers need to concern about containers creation and termination, and usually need to take care of the scalability of storage. While in serverless, these matters are managed by providers with their full set of cloud services, such as data storage, database, and content delivery network. The serverless functions are run as a more centrally managed service [19].

We summarize the pros and cons of serverless computing below.

Strength of Serverless (S)

S1 Simplification. Serverless saves the burdens of server and resource management for developers. Back-end development is simplified with the API from serverless platforms. Also, the service deployment process is simplified. Developers only need to indicate the function entry, and the rest is handled by providers.

S2 Extended Scalability. Serverless architecture is designed in a way that is highly scalable. The functions are run separately in containers, which will be automatically started up with adaptive request demands. This leads to the potential of a highly scalable MVE if designed properly.

S3 Low Cost. With the pay-as-you-go billing model at a high granularity (usually at the unit of 100ms [20]), only resources that are consumed by running functions are charged. Compared to IaaS which is billed in resource bundle, the cost on serverless is lower.

Weaknesses of Serverless (W)

W1 Vendor Lock-in. Different cloud providers offer slightly different components and APIs. It can be difficult to switch providers as it takes time and effort to change them in codes. Also, not all platforms provide the same set of components, which may result in malfunctioning if migrated.

W2 Overhead. Overall system performance can be impacted by the overhead on serverless architecture, such as function cold start time during initialization and scaling, and function routing time during HTTP requests.

W3 Variable Performance. Performance variability widely exists in cloud computing, so does in serverless. The additional variance may impact the overall system performance, especially for latency-sensitive application such as online games.

2.2.2 Azure Cloud Platform

In this work, we make use of Microsoft's Azure Cloud Platform. As of 2021, Azure has 19% share in the cloud market, second to Amazon's AWS [21]. Thus, it is a representative cloud platform. We introduce some services provided by the platform, including their naming and limitation, which we will take into consideration during implementation.

Azure Functions is the Azure serverless computing. There are three subscription plans. Consumption plan automatically scales and only charges when in use. Premium plan adds support of pre-warmed workers. Dedicated plan assigns dedicated resources and allows full customization. We consider in this work consumption plan only, because it is the most representative one. It provides three types of function triggers, namely HTTP, Timer, and Event. The input and output data format is not restricted and can be fully customized. The resource allocation per function instance is fixed at 1.5 GB memory and 1 CPU.

The function's maximum duration is ten minutes. Users can choose runtime environment between Windows and Linux and region from availability zones.

Azure Blob Storage is the object storage service optimized for storing massive amounts of unstructured data. It consists of three-level of components, namely storage account, container, and blob. Storage account is a namespace for all the data. It provides an endpoint for access and tokens for authentication. A container is similar to a folder and can contain an unlimited number of blobs. A blob presents an object, including the file metadata, including file name, modification time, locks, and binary data. The file name can be made of file path, which works similarly as layered directories. The maximum storage space for block blob is 190.7 TB.

Azure SQL is a database service based on Microsoft SQL (MSSQL). Although Azure provides different database back-ends, including MySQL, and PostgreSQL, only MSSQL service provides a pay-as-you-go (serverless) billing model. The serverless SQL is billed based on actual CPU core usage per second and the database size.

Virtual Machine is the IaaS service provided by Azure. A wide range of specifications is available for selection.

Azure Monitor is a traditional monitor for other services, such as Azure Blob Storage and Virtual Machines. It provides an API for retrieve metrics. However, there are two limitations on the API. First, it is not possible to retrieve raw data points. One of the aggregators, including maximum, minimum, count, and total, must be appended to the request. Second, the minimal time interval for data aggregation is one minute. We need to take these limitations into consideration for our experiments.

Application Insights is the new feature of Azure Monitor, specifically designed for Azure Functions. It collects metrics and logs produced during function invocation. Azure tracks metrics in a non-blocking thread, and sends batched telemetry data, which implicates that the overhead is small. Application Insights provide API to query raw data points. However, it is only available for a limited amount of metrics, such as requests. Other metrics are served by Azure Monitor, which is subject to the same limitation.

2.2.3 Performance Variability

The same performance is not always guaranteed for applications hosted in clouds. The reason is that it is difficult to guarantee resource usage in shared infrastructure among tenants, and the resource management and scheduling policies enforced by the providers may not be suitable for all kinds of applications. Previous studies show that performance variability is a widely observed phenomenon in cloud computing [14, 15]. Specifically in

serverless functions, Ginzburg et al. [16] demonstrate that the phenomenon is significant in AWS Lambda, a serverless service provided by Amazon.

The variance significantly may impact the overall system performance, especially for latency-sensitive applications such as MVE. The performance variability may result in QoS metrics not being met, which leads to degradation of player experience. It remains an open research question to understand how the performance variability impacts different parts of MVE as a serverless system running on the cloud.

3

Related Work

In this chapter, we survey related work on benchmarking MVE and serverless systems.

A few performance analyses exist for Minecraft games and MVE. Yardstick [8] is a benchmark suite to analyze the performance of several Minecraft servers. It emulates players and automatically submits workloads to the server.

Alstad et al. [22] analyze the network performance of Minecraft by considering the impact of player type and number, player activities, and virtualization delays.

Manycraft [23] aims to scale Minecraft to millions of players on a static map through Kiwano infrastructure, which is a scalable distributed infrastructure for virtual worlds.

Donkervliet et al. [11] envision a new architecture for large-scale MVE games with serverless technology. They propose a model of services and deployments for MVEs as serverless systems that run as independently scheduled services so that each part of the system can scale independently with the potential to support millions of users. The vision is the main motivation of our work.

Serverless is the latest paradigm in cloud computing. Some studies exist to migrate monolithic systems to serverless architecture. For example, the performance of a document processing system used in Fintech is significantly improved after moving to serverless architecture with only a marginal increase in cost [24].

The additional latency between different components of serverless architectures poses new challenges. Many studies exist to understand them by benchmarking. FunctionBench [25] provides realistic workloads for evaluating various cloud function services, combing microbenchmark, machine learning models, and real-world data-oriented image and video processing applications.

Towards latency-sensitive applications hosted on serverless platforms, Pelle et al. [26] first conduct a detailed benchmark on Amazon's AWS, adjust a drone control application

to the platform, and study its performance. They observe that the variance and the jitter is very high and conclude that the serverless approach is feasible for applications that can tolerate latency up to 100 ms.

SPEC-RG group envision a comprehensive serverless benchmark that evaluates serverless ecosystems. They compare the performance overhead when feeding workload on both closed- and open-source serverless platform [27].

FaaSdom [28] is a benchmark suite that provides insights on the performance of serverless applications on AWS, Azure, Google Cloud, and IBM. It covers a wide range of workloads and implementation languages. It also compares budget costs for an application hosted on these four platforms.

Ginzburg et al. [16] study the performance variaiblity on AWS Lambda. They observe that the variation is significant but also stable. The lack of performance isolation between tenants can be exploited for performance improvement and cost reduction by using placement gaming algorithms on the platform.

To our best knowledge, currently, no study focuses on benchmarking serverless architecture with online gaming workloads. Thus, we extend Yardstick to benchmark our serverless MVE prototype, with the benchmarking technologies on serverless from related works.

4

Serverless MVE Design

In this chapter, we present the design of a serverless MVE. We first define a set of requirements and present a high-level design that fulfills the requirements.

4.1 Requirements

R1 **Meet QoS requirements, especially in terms of overall system latency.** The serverless MVE should not add noticeable latency compared to the monolithic MVE.

R2 **Improve scalability.** The ultimate goal of the novel design is to support more concurrent players in the same virtual world.

R3 **Compatible with current MVE clients.** The serverless MVE design should be compatible with current MVE clients so that the players can connect seamlessly without the need to modify their current installation.

R4 **Ease of deployment.** The serverless MVE should be easy to set up and deployed on a cloud platform.

R5 **Metrics Collection.** The serverless MVE should provide access to metrics across different levels from each component for performance analysis.

4.2 Design Overview

Considering the design requirements, we make a high-level design of serverless MVE and show it in Figure 4.1. The MVE client is the traditional game client discussed in Section 2.1.1. We keep it unchanged to satisfy R3. Players use existing client software (①) installed

in their local devices to connect to the system gateway. We discuss our main components, namely Gateway, MVE Server Pool, and Serverless Functions.

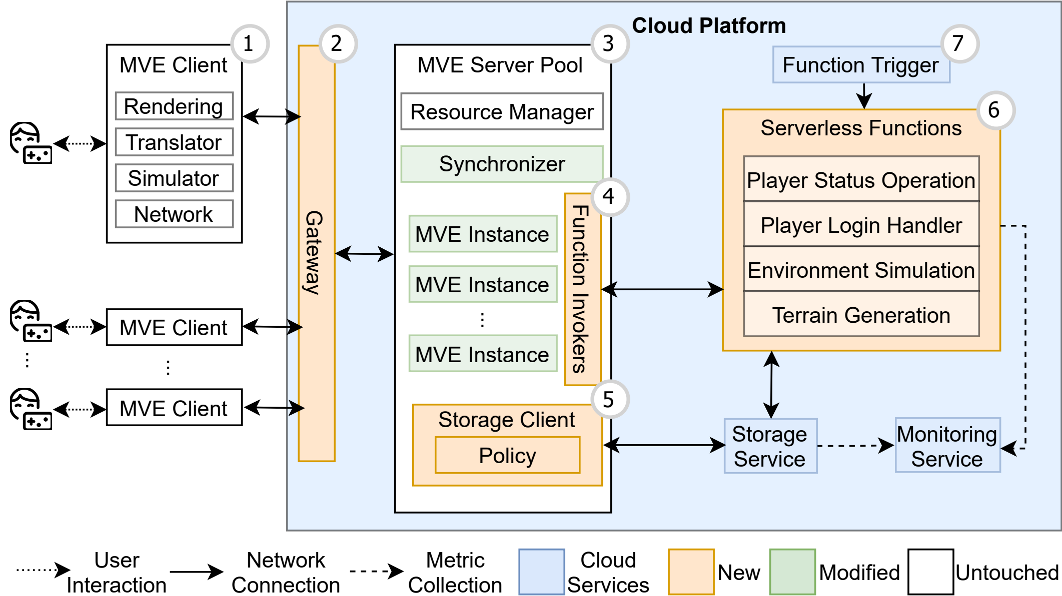

Figure 4.1: Serverless MVE Model.

4.2.1 Gateway

The gateway (②) is a connection entry point for clients. It consists of components such as firewall and load balancer at application layer (layer seven of the Open Systems Interconnection model [29]). The gateway listens to one or more pre-defined network ports and handles incoming connections from MVE clients.

If the connections are from a new client, the gateway looks up an available server in the MVE server pool, routes the connections to it, and records the client-server pair. For connections from an existing client, i.e., a client that matches the record, the gateway routes the traffic directly to the destination server based on the record. Overall, the gateway hides the server details and internal processes from clients, so that existing clients can seamlessly connect to servers (R3).

4.2.2 MVE Server Pool

The MVE server pool (③) contains a resource manager, a synchronizer, and a number of MVE server instances. Initially, a pre-defined number of MVE servers are started. The MVE server is designed to be *thin*, which offloads most computation tasks to serverless

functions, while keeping only lightweight computation tasks and network stacks locally. It keeps the existing network stack for communication with clients (R3). The MVE servers exchange information with serverless functions through a list of Function Invokers, where each invoker can invoke one or more serverless functions.

Additionally, the MVE server connects to storage service provided by the cloud platform with storage client. Persistent data of the system should be stored in storage service instead of local filesystem. There are three benefits by storing data in remote service. First, the scalability of file storage is handled by the infrastructure of the cloud provider (R2). Second, all MVE servers and serverless functions can share the same copy of data, which provide system-wide consistency. Third, it is no longer necessary to consider storage provisioning for MVE server, because the remote storage space is automatically grown on demand.

The synchronizer guarantees the consistency state between different MVE servers. Different consistency model can be applied depending on the system requirement. We here present two use cases. First, the storage service only provides system-wide consistency. After the remote file is loaded, the consistency state is no longer the concern of storage service. To avoid conflict update to a single remote file, strong consistency model must be applied to guarantee the data from different servers are synchronized before writing to storage.

Second, the synchronizer provides a global view of players in different server instances. The players on one server instance should acknowledge the players in another server instance, if they are in the area of interest.

The resource manager keeps track of the system load of each server, and creates new server instance whenever necessary (R2). All server instances use the same copy of server programs. By automatic scaling up the number of servers, the connections from clients can be processed without overloading a single server, which improves scaliability.

4.2.3 Function Invokers

Similar to a traditional function invoker, a serverless function invoker (④) is a caller of a remote serverless function. One function invoker can call one or multiple specific functions.

A serverless function invoker accepts parameters from MVE servers, constructs a request with them, and sends it to a serverless function. When the serverless function is executed, the result is returned to the invoker, which then passes it to the server. Typically, an invocation request is an HTTP request, because HTTP is the standard communication

protocol of current serverless platforms. The function invocation process can be either synchronous or asynchronous depending on the system requirement.

4.2.4 Storage Client

The storage client (⑤) is used to connect to the storage service of cloud platforms for Input/Output (I/O) operation. Usually, cloud platforms provide SDK for development. Because the storage is remote, the latency is higher than the local filesystem, which can lead to QoS degradation. To meet QoS requirements (R1), different cache policies should be applied depending on the needs of latency hiding. The remote I/O process can be either synchronous or asynchronous depending on the system requirement.

Generally, cloud platforms provide different types of storage services for different purposes, such as persistent or ephemeral. The storage client should carefully select storage service based on purposes. For example, player data should be stored in persistent services, while temporally simulation data that will be deleted after retrieved should be stored in ephemeral service.

4.2.5 Serverless Functions

Implemented by developers, serverless functions (⑥) are the business logic of an application. It can utilize other components in cloud platforms. A serverless function is designed to serve a single purpose. In the context of MVE, it can be used to serve one specific feature.

To satisfy R1, two types of tasks are designed to be serverless functions. First, the task that is not directly related to interaction activities with players can be migrated to serverless. Typically such a task can be run independently without blocking the main thread of an MVE server. For example, environment simulator changes the world environment, e.g. weather, time, after a certain time or triggered by command events. The simulation is completely independent regardless of other events from the MVE server, and usually does not need to be complete within one game tick. Another example is to unban a specific player after a certain time. We can safely offload such tasks to serverless functions without the concern of added latency. The functions can be triggered either by MVE servers through invokers, or other trigger events provided by serverless platforms (⑦), such as a timer. The MVE server then retrieves the updated simulation results and changes its game state periodically, e.g. during several game ticks.

Second, the task that is computation-intensive can be migrated. This indicates that the task completion time surpasses the introduced latency between serverless function and MVE server. To understand which tasks are computation-intensive, we need to conduct performance analysis. For example, Yardstick [8] shows that the majority of the data sent by MVE server is related to the generated terrain data while players explore the world. Thus, we can offload the terrain generation task to a serverless function, so that the system load of the MVE server is lowered.

4.2.6 Monitoring Service

Monitoring Service logs all events and metrics from the cloud platform. In a serverless MVE, the serverless functions should call the monitoring API, so that the metrics can be recorded and retrieved for future analysis (R5).

5

Serverless MVE Implementation

We implement a serverless MVE prototype based on Opencraft, which is a Minecraft server written in Java, specific for research in massivizing Modifiable Virtual Environments [30]. We choose Microsoft Azure as our cloud platform and implement the prototype with its services and components. In this chapter, we discuss the implementation details of our serverless MVE prototype.

5.1 Persistent Cloud Storage

Currently, Opencraft stores persistent data in local filesystem. As our first step to realize a serverless MVE prototype, we migrate them to cloud storage. Cloud storage provides three advantages through its abstraction. First, the data can be accessed in a distributed way, e.g. by different game instances and serverless functions. Second, cloud storage guarantees high scalability. By migrating persistent data to the cloud, the scalability is improved in terms of Input/Output (I/O) service. Third, cloud storage only charges for the used space and requests. Over-provisioning is no longer necessary. This leads to lower costs.

Azure provides different cloud storage services. In our implementation, we use Azure Blob Storage, which is an object store for text and binary data. As a comparison, we add a Microsoft SQL back-end for structured data so that we can evaluate the Azure Serverless SQL service.

The implementation includes three steps. First, we create a class *BlobClientAzure* to serve cloud IO requests. The class utilizes Azure Blob Storage SDK to create clients for the three components. It provides different upload and download methods to serve different scenarios. This includes execution policies, i.e. synchronous and asynchronous, and data type, i.e. plain text, stream, and file. Additionally, we notice that Azure SDK reads data

Table 5.1: Details of Persistent Data on OpenCraft. Legend: Server Initalization (SI), World Save (WS), Player Login (PI), Player Logout (PO), Player Movement (PM), Access Frequency (Freq), File Size (Size).

			Access Event						
ID	**Data**	**Format**	**SI**	**WS**	**PI**	**PO**	**PM**	**Freq**	**Size**
PS1	Player	NBT		✓	✓			Low	Small
PS2	Player Statistic	JSON			✓	✓		Low	Small
PS3	Player Metadata	JSON	✓		✓	✓		Low	Small
PS4	World Metadata	NBT	✓	✓				Medium	Small
PS5	Scoreboard	NBT	✓	✓				Medium	Small
PS6	Structure	NBT	✓				✓	Medium	Medium
PS7	Region	Anvil	✓	✓			✓	High	Large

from storage service to memory as output stream, and writes input stream to remote. While Java reads data from local storage to memory as input stream, and writes output stream to storage. Thus, the class also provides conversion between input and output streams by using ByteArray as an intermediate. For connecting to Microsoft SQL, we use JDBC driver.

Second, we analyze the existing persistent IO services on Opencraft and classify the data types. Table 5.1 shows the major types of persistent data and their properties. The table does not include events triggered by commands from administrators.

Name Binary Tag (NBT) is a tree data structure consisting of various tags, each of which includes a name and an ID. The data structure can be compressed by GZip. PS1, PS3, PS4, and PS5 employ such a format. Anvil is a file storage format mainly used for storing chunk data. It is a container of NBTs and it is used for storing region data (PS6) where each region anvil file stores a group of 32x32 chunks.

Third, we extend the I/O service classes for each persistent data and modify the storage provider class to create I/O services with our new classes when the configuration is set to Azure. For small and medium files (PS1 - PS6), we load the remote blobs into memory streams using *BlobClientAzure* class, and write the streams directly to remote blobs when save is needed. For large files like PS7, we apply latency hiding policies to hide the latency from players (Discussed in Section 5.1.1).

Forth, we assign different execution policies depending on the requirements of event logic. For example, for player login event, we assign synchronous read for player data, because the data must be ready before logging in the player. We cannot load the data in advance as

we cannot predict which player will login next. The write process is asynchronous because wait is not necessary.

For auto save event, we assign asynchronous write for all data so that the process does not block the main thread. For server stop event, we assign synchronous write to wait for all data to finish uploading.

5.1.1 Latency Hiding Policy for Region File

Region Files (PS7) contains a large number of chunk data, which leads to high latency when reading the files. If the players move to a location that is not previously loaded, an incomplete map will show up while being loaded. Additionally, the data is frequently accessed, which can result in high bandwidth usage if the instance frequently requests the files from cloud storage. As a result, a cache policy is necessary.

Opencraft has a default cache policy based on Guava Cache for efficiently accessing multiple region files simultaneously. The default cache is between local filesystem and memory. We extend the class and add a cache layer between local filesystem and remote storage, making it a three-layer cache. The reason we keep local filesystem layer is that the memory resource is not enough to keep all region file cache, the cache size is small and expiry time is short. This results in frequent save to remote storage, leading to high bandwidth usage. By using a local filesystem layer, we can delay the upload process to save bandwidth.

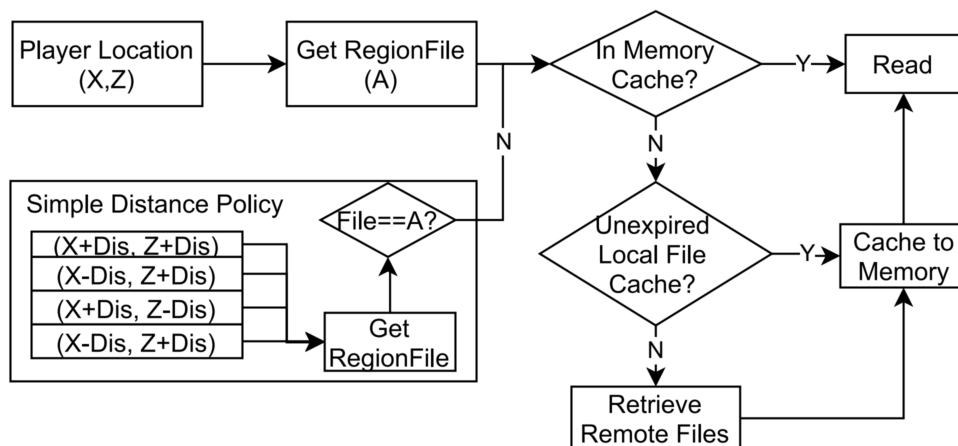

Figure 5.1: Process of Region File Cache (Read).

Figure 5.1 shows the read process of Region File cache, which is triggered by the change of player location event. It checks which region file stores the current chunks, and finds

the data, in the order of memory, local filesystem, and remote storage. If the data is found in a layer that is not memory, the data is then cached to memory.

To improve the hit rate of memory cache, we also utilize a preloading policy. The instance should not only cache the regions that are visible to players, but also those that the players will *likely* visit. We use here a simple distance policy, that is to calculate $newLocations[] = currentLocation \pm distance$, where distance is a pre-defined value. These new locations will likely be visited by the player. Then we check the region file names of the new locations. If any new file name does not match the original one, the new region files are cached in the same way.

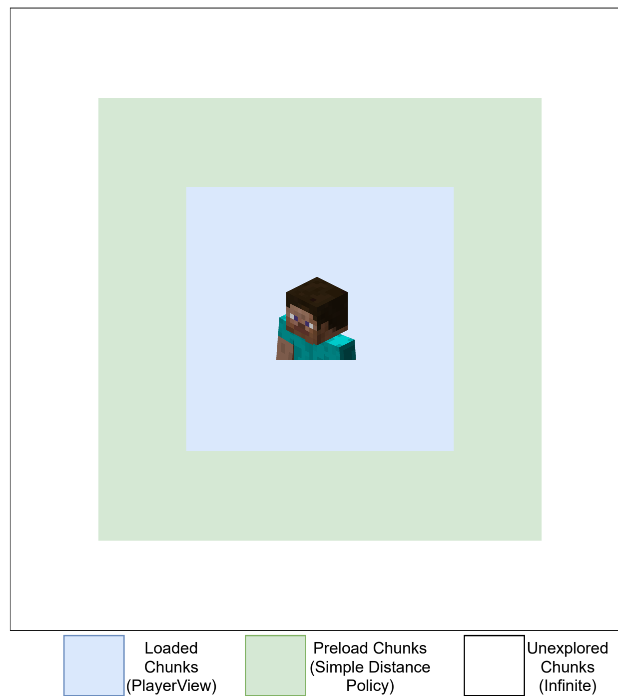

Figure 5.2: Demonstration of Simple Distance Preloading Policy.

Figure 5.2 demonstrates the simple distance policy in player's view. The chunks in blue are loaded by default, which is within the player's view distance. The chunks in green is the additional chunks preloaded by simple distance policy. The chunks in white represents the infinite unexplored chunks.

Figure 5.3 shows the write process of region file cache. The write request is first written into memory buffer in size of 4KB, which is the typical minimum write size of filesystems. When write is complete, the data is stored directly into local file cache, so that memory cache and be unloaded anytime safely. Upload to remote storage is triggered by events,

such as auto save and server stop.

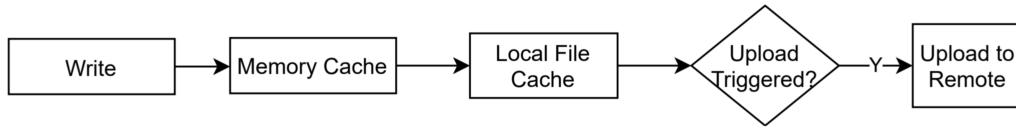

Figure 5.3: Process of Region File Cache (Write).

5.2 Serverless Functions

In this section, we discuss our implementation details of serverless functions based on Azure Functions.

One common way to migrate a monolithic system to serverless functions is to break down the system into small services, which are loosely coupled and communicated via HTTP requests by passing the needed variables [24].

Azure provides core tools for deployment and a Java SDK. Azure Function Core Tools deploy serverless functions based on a set of properties, including the application name, endpoint, datacenter region, operating system, subscription plan, and monitoring service. Java SDK provides a set of definitions for a specific function, including the name, trigger, response, event, and connection strings.

Making Opencraft serverless is a challenge because many services are tightly coupled with references to a large number of classes and some of them are stateful. In this work, we mitigate four game features to serverless functions, and our process for each function include three steps. First, we identify the code segment that handles the game feature, and move it to serverless function with Azure SDK. Second, we create a function invoker that receives objects from the game, constructs requests, receives returns, and parses the returns to objects. Third, we modify Opencraft to interact with the invoker. In this process, we utilize a new scheduling policy to asynchronously interact with the invoker, and checks the asynchronous results during game tick.

5.2.1 Player Status Operation Function

Player status operation service handles the requests for looking up player status, banning and unbanning a player, changing the player role, and managing whitelist. The service mainly concerns I/O operations with the specific player metadata, which is lightweight

with few references to other game classes. Thus, it is easy to migrate to a serverless function.

Our implementation includes the following steps. First, we analyze two classes *GlowBanList* and *UUidList*, which are the main classes for operating player status. We see that the original process loads the metadata from files into Map during server initialization, and dumps the Map to files whenever there is a change.

Second, we create a new Azure Function *PlayerOperation*, which listens to a HTTP trigger, and operates low-level I/O operation with blob storage. The function provides an API interface that accepts JSON as input. For each triggered request, the function parses the input JSON, identifies the resource, operation, and additional information from the request body. It performs I/O operation accordingly with *BlobClientAzure* and returns the result. To reduce request size, besides implementing the original methods (getall, setall, isban, expungeBan), the function provides methods for modifying a single resource (get, ban, and unban one player).

Third, we overwrite the original methods in Opencraft. To read metadata, the method sends a request to serverless function with HTTP client, then processes other logic with the obtained results. When there is an operation on a player, the method sends the single request to serverless function asynchronously. If an exception occurs during the process, the methods fall back to the original one, i.e. dump the whole map, and send it to the function. Also, expungeBan operation is removed from the MVE server.

Last, to automatically expunge bans when expired, we create a new Azure function with 1-minute timer tigger. The function sends multiple expunge requests with different resource types to *PlayerOperation*. Function *PlayerOperation* then performs expungeBan, which unban players if their expiry time is before the current system time, by iterating through the requested resource.

5.2.2 Player Login Handler

Whenever the server receives a login request from client, the server looks up the player data, e.g. username and authentication information. If the player has played before, it also looks up previous game data such as last location, inventory, bed location, player status, and statistics.

We mitigate player status operation to serverless functions as discussed in Section 5.2.1. We further mitigate the other process to serverless functions. The process is feasible because Opencraft provides a data reader implementation. We move the data reader to serverless function handler, and provide an API for looking up data.

The Opencraft default scheduling policy is not capable for handling player login after we move some logic to serverless functions. The reason is that the default policy waits for asynchronously tasks to return results between different schedulers within a tick. In the event of cold start of serverless function instances, which may take up to several seconds, the default schedulers stall the tick loop while waiting for results for player login event.

We implement a new policy for handling player login. When the server receives a login request from client, it starts a new Callable to retrieve data from serverless functions, and stores the Futures in a ConcurrentHashMap. During every tick, the server checks whether the Futures are done. If so, it processes other login logic, e.g. assigning the player into the world, and sending related packets. To avoid the game tick being overloaded, player login events are handled in sequence. Each tick the server only checks Futures once and processes one Future login task. Only after the current login process is complete or cancelled, the server starts to process the next one.

5.2.3 Environment Simulation Function

Environment Simulation includes time and weather simulation, which changes the game time and weather as per time changes. The default implementation maintains several counters that are increased every tick, and changes time and weather parameters based on counters with random factors. If any condition changes, the server calls related events and notifies players.

To make the environment simulated serverlessly, we first move the code segments that handle the simulation to serverless functions, which includes going through the in-game day cycle, changing weather parameters, and processing game master commands.

Second, to keep the data persistent upon function invocation, we store the data in blob storage. The function handler reads history values from blob storage at the start of invocation, and store the new values into blob storage at the end. By doing so, multiple server instances can share the same simulation results.

Third, we modify the tick logic for environment simulation on Opencraft, which is simimlar to player login as discussed in Section 5.2.2. Every thirty seconds, the server requests the latest simulation results from serverless functions asynchronously with Java Callable, and store the Futures with a ConcurrentHashMap. During every tick, the server checks whether there are completed Future tasks. If so, the server gets the results, calls related events, and notifies players.

Last, we keep the local simulation so that the players can observe the effect of commands immediately. Upon receiving environment simulation commands, the server performs local

simulation and broadcasts the results immediately to players, then it sends the local results to serverless functions. The serverless functions take the local simulation results as offsets and perform serverless simulation accordingly, then return and store the results.

5.2.4 Terrain Generation Function

Terrain Generation is one of the heavy computations when players explore the infinite world. There is already one function implemented on AWS Lambda that accepts coordinates, world seed, and other stateful objects as input, and returns the chunk data as output.

We migrate the AWS Lambda function to Azure Function. First, replace the function library to Azure function with HTTP trigger. AWS Lambda requires input and output to be JSON, thus extra (de)serialization processes are necessary. However, for Azure function, we can send any format of data. As a result, we remove the serialization that is not necessary at Azure Functions to improve performance.

Second, we modify the function invoker. AWS Lambda provides a library with Function invoker, while there is no in Azure Functions. Thus, we reuse our implemented invoker and make it called by the function which loads generated chunks for all players during every tick.

5.3 Gateway

We implemented a layer seven gateway prototype with MCProtocolLib [31], a library that wraps network packets for both Minecraft server and client protocols. The gateway is the entry point for policy-based packet forwarding between MVE clients and servers.

Figure 5.4 shows the flowchart of our implemented gateway. The gateway listens to a port and accepts incoming connections with server protocol after start. Two packet handlers, namely server protocol handler and client protocol handlers, are the main components of the gateway. They work in an event-driven mode where incoming packets received from game clients or servers are events that trigger the functions. The gateway maintains a list of player info data in memory, which stores the current player states.

The server protocol handler acts as a server to game clients. There is one server protocol handler per gateway. It identifies and handles mainly three types of packets. First, *LoginStart* Packet is the first packet that the client sends during a login process, which indicates a new connection from a client. If the gateway receives such a packet, it looks up player information from the packet, performs the authentication process. Then, it

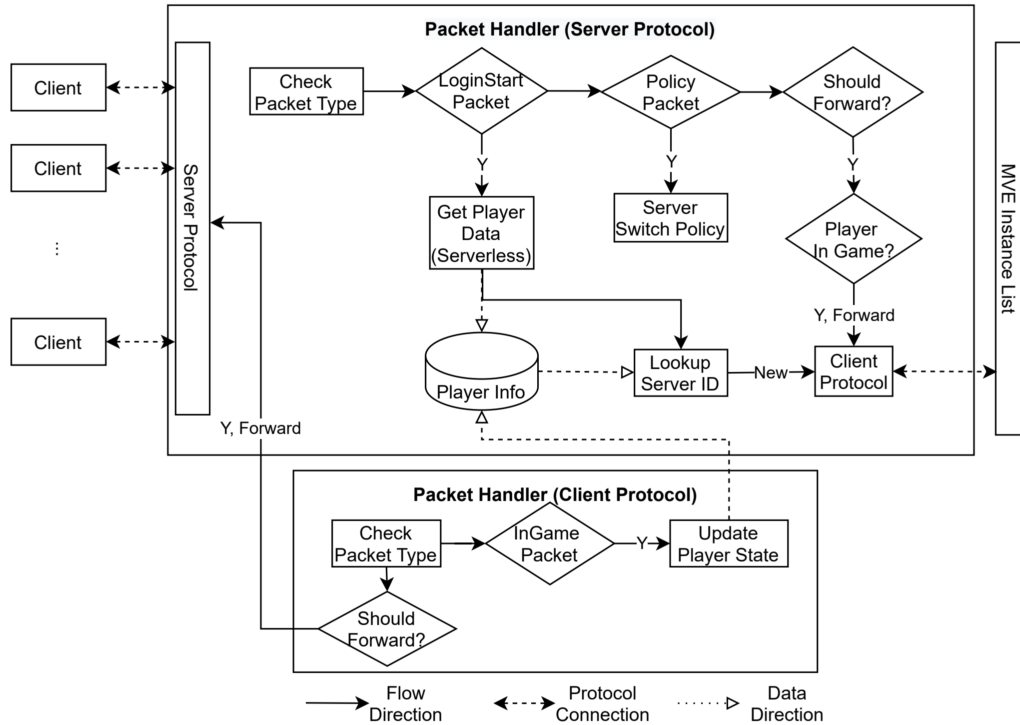

Figure 5.4: Flowchart of Gateway for Policy-Based Packet Forwarding.

looks up the initial server id for the player according to server split policy, launch a client protocol to connect to the server, and saves the session. The client protocol also contains a packet handler, which will be discussed later.

Second, policy-related packets are to identify whether the player should be transitted to another server. If the gateway receives such packets, it transits the players to a new server according to the policy and notifies the player.

Third, packets that need to be forwarded are related to game contents. The gateway only forwards In-Game packets received from MVE instances to the client. Other packets which are processed internally, such as *Handshake* and *KeepAlive*, are ignored. The gateway looks up whether or not the player is in game state. If not, it saves the packets and replays after the player reaches in-game state.

The client protocol handler works as a client to game servers. There is one client protocol handler per player. It works in a similar way as server protocol handler, which receives packets from MVE instances, identifies the packet types, and performs actions accordingly.

First, for in-game packets, the handler updates the player status to in-game, so that the packet handler of server protocol can start to forward in-game packets.

Second, it checks whether or not the packets should be forwarded. The client protocol handlers only forward in-game packets. All other packets, e.g. *Handshake*, *KeepAlive*, *ServerDisconnect* are handled internally in the gateway. Also, it ignores server initialization packets such as *ServerJoinGame* so that the game client does not reload.

The gateway is highly scalable with its event-driven modes based on incoming packets. The players connect to gateway with hostname, which is updated with DNS load balancer that maps to different gateways based on their IP address.

5.3.1 Policy for Multiple Server Instances Routing

The terrain in Minecraft is infinite and is presented with coordinates x, y, and z. We implement a simple policy based on the x and z coordinates of player locations, where each server handles a part of the terrain. The players are redirected into different servers based on their current position. We do not consider the y axis here because it has a bound between 0 to 255 and it can only be reached with fly action.

Figure 5.5 demonstrates a policy to split terrain into several server instances. (0,0) indicates the origin of coordinates. The arrow indicates the positive directions. For two server instances, we split the infinite world into two halves, with line z at origin as the dividing line. Similarly, for four server instances, four lines split the terrain with coordinates where one of which is zero and the other is from zero to positive or negative infinity. Each part of the terrain is still infinite after being split.

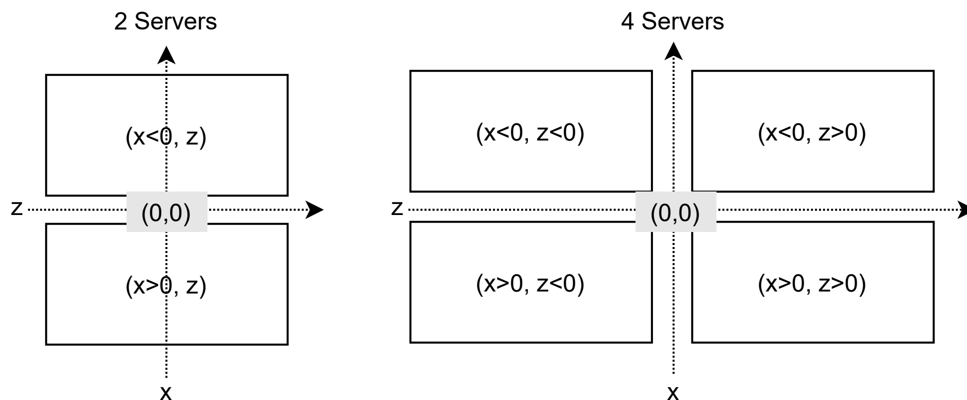

Figure 5.5: Demonstration of Splitting Terrain into 2 and 4 MVE Instances.

The policy-related packet is *ClientPlayerPosition*. When the client protocol packet handler receives such a packet, it checks whether the player reaches the bound of terrain in

the current server. If yes, the gateway notifies the player with messages and starts the transition process.

The transition process first looks up the new server id according to the world bound of each server. Then, it launches a connection to the new server and saves the session. Only after the new session is established, the gateway disconnects the previous session. To replay the player movement during the transitting process, the gateway sends a new *ClientPlayerPosition* packet with offset to new server, and sends a new *ServerPlayerPosition* packet with offset to the client.

The handler caches the packets received before the new session is established and the player reaches in-game state, and replays them After the player reaches in-game state. The related packet handlers then process these packets accordingly.

6

Benchmark Suite

In this chapter, we present a benchmark suite for benchmarking the serverless MVE prototype. We first define a set of requirements, and present the system design overview. Then we discuss the implementation details. Finally, we present an overview of the metrics we collect during the benchmark.

6.1 Requirements

Currently, there is no research in benchmarking a serverless MVE. Defining a set of requirements for benchmarking a serverless MVE is difficult. We follow general guidelines on benchmark [32], and adapt domain-specific benchmarks on MVE [8] and serverless [27].

R1 Relevance The benchmark should generate and submit suitable workloads that are relevant to a serverless MVE, and collect metrics that are relevant to performance variability.

R2 Fairness The benchmark should analyze performance on a class of compatible systems, i.e., serverless MVEs that utilize similar architecture. Also, the benchmark should allow the comparison of a set of metrics based on different configurations. There should not be a bias toward one system or one configuration.

R3 Portability The benchmark should run on general servers, regardless of specification. The benchmark should be configurable to adapt different server specification and operating systems.

R4 Usability The benchmark should be easy to set up and configure. The steps for deployment, benchmarking, and result collections should require few manual involvement.

R5 Low Overhead The benchmark should not influence the system performance. To obtain accurate benchmark results, the benchmark suite should not pose significant overhead to the system performance.

R6 Reproducibility The benchmark results can be reproduced when feeding the same workloads in the same environment. This guarantees the validity of benchmark results.

6.2 System Overview

To satisfy the benchmark requirements, we design a benchmark suite. A high-level design of our benchmark is shown in Figure 6.1, where the arrow indicates the data direction. The benchmark suite includes three main components, benchmark server and benchmark client for deployment and metric collection, and a tailor-made MVE client specifically for benchmark purposes. The components are communicated with each other through client/server architecture with reliable TCP connections, which can be used on systems that utilize similar architecture (R2).

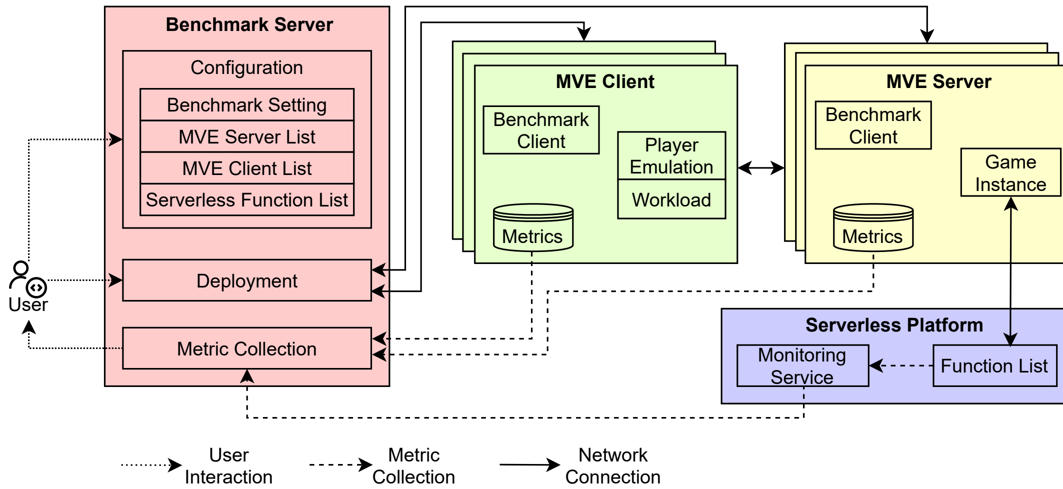

Figure 6.1: Benchmark Suite Overview.

6.2.1 Benchmark Server

The benchmark server consists of three components, namely configuration, deployment, and metric collection.

The configuration file stores global parameters related to benchmark as defined by users based on a template prior to benchmark (R4). We use JavaScript Object Notation (JSON) format, which is a standard for structured data. Table A.1 shows the parameters used in the configuration file. We define here the connection and authentication details for all components of the benchmark suite, including benchmark setting, benchmark server, benchmark clients running on MVE servers and clients, and serverless functions.

The deployment process automatically sets up and controls benchmark environments on destinations, including MVE servers, MVE clients, and serverless functions. It mainly uses SSH protocol, which is a standard service running in modern Linux distributions and can be used for transferring files and running bash commands across platforms (R3). The deployment scripts are written in Python 3. The process is as follows. First, it reads the configuration of all MVE servers and clients, establishes SSH connections with them by utilizing paramiko, which is an SSH module in Python.

Second, it transfers all necessary files to MVE Servers and Clients via SFTP. For MVE servers, the files to be transferred include installation scripts, benchmark configuration, benchmark client for MVE server, Opencraft server, and server configuration. For MVE clients, the files to be transferred include installation scripts, benchmark configuration, benchmark client for MVE client, and MVE client. To shorten file transmission time, we check the MD5 hash for large files that already exist on remote components and match them with local files. Only when the two MD5 hashes do not match, the files will be transmitted.

Third, it executes installation scripts on all MVE servers and clients. The installation script is written in bash, which is a standard shell in most modern Linux distributions. The script is compatible with all operating systems that support bash shell. It installs necessary software, including Python 3 runtime, openjdk for java runtime, and screen for virtual consoles. After the deployment process is complete, users can choose to start benchmark.

The benchmark process first starts a TCP socket, listens to a port, and waits for incoming connections from benchmark clients. Second, it checks whether there are pending iterations. If true, it starts benchmark clients on MVE servers and MVE clients after all MVE servers are started. The benchmark clients start to collect metrics locally and establish TCP connections with the benchmark server. The services are run through detached screen sessions so that they do not block the main thread of the benchmark server. Also, by using screen sessions, users can check the intermediate process by directly interacting with them.

When the benchmark server receives a new connection, it starts a new thread for handling messages and records the connection details. Table A.2 shows the messages exchanged between benchmark server and benchmark clients.

When an MVE client finishes its task, the benchmark server retrieves its metrics via SFTP, removes it from record, and checks whether there are other active MVE clients for the corresponding MVE server. If there is not, it sends shutdown command to the MVE server and removes it from the record as well. The benchmark server then retrieves metrics from the MVE server via SFTP. The reason to retrieve metrics after benchmark ends is that sending real-time metrics influences network metrics, and may impact the network conditions, which leads to performance overhead (R5).

Periodically, the benchmark server checks whether there are active MVE servers in the record. If there is not, it retrieves all function metrics by calling Application Insight API. The current iteration is considered complete. The benchmark server starts the next iteration from the second process. When all iterations are complete, the benchmark server ends.

6.2.2 Benchmark Clients

We implement benchmark clients in Python to run on the MVE server, MVE client, and gateway respectively. We use subprocess.Popen() method to spawn new processes so that we can perform underlying operations, such as recording process id and sending kill signal. For system metric collection, we use psutil, which retrieves system metrics on a running process from system counters. We record the counter data once per second. For application metric collection, we use the internal event logging.

The benchmark clients first establishes a TCP connection with benchmark server. It spawns a new process for related software based on their roles, e.g., MVE server, MVE client, or gateway. After the software is started, it spawns a new process for metric collector. Then, it starts an indefinite loop for receiving messages from the benchmark server.

When the task is finished, the benchmark client sends a message to notify the server. When shutdown command is received, the benchmark client kills the two processes, sends the file paths of metric collections to the benchmark server, and ends itself.

6.2.3 MVE Thin Client

The MVE thin client submits workload to the MVE server during runtime and terminates itself when tasks are complete (R1). Our MVE thin client is based on YardStick [8], which

provides by default some Executors that perform simple tasks, such as walking, breaking, and building a block. Also, it provides pathfinding and collision avoidance.

We extends Yardstick in two ways. First, we add new features such as chatting and teleporting. We do so by constructing new network packets with MCProtocolLib [31], which is a library that wraps network packets for Minecraft protocols. Second, we add experiment controllers to submit our workload, which we discuss on Section 6.4.

6.3 Overall Benchmark Process

The overall process of a benchmark is as follows. First of all, the user modifies the configuration file to define necessary benchmark and authentication parameters. Then the user runs the deployer, which automatically sets up the benchmark environment on MVE servers, MVE clients, and serverless functions. After the environment is set up, the user can start the benchmark.

After the benchmark is started, the MVE clients submit workload to the MVE servers based on configuration. The MVE servers invoke respective serverless functions during runtime depending on the workload. Metrics are collected asynchronously and stored locally at a constant time interval so that the collection does not pose overhead for performance. When the benchmark is complete, the benchmark server is notified and will retrieve metrics from all benchmark clients. The user can perform data analytic on the collected metrics. The experiment is repeated based on the pre-defined number of iterations.

In the event of failure, the benchmark server reports exception information and tries to restart the benchmark.

6.4 Workload

Due to the lack of publicly available actual traces on MVE games, we do not have a real world workload. Considering the implementation of our serverless MVE, we design two player behaviors, namely Straight Walk Behavior, and Random Behavior, and use them as the workloads for our benchmark. The goal of Straight Walk Behavior is to explore as much terrain as possible with a given duration, while Random Behavior is to emulate the random activities of players during actual gameplay.

Non-player workloads are the same as what we see when actually playing the game. In this thesis, we only evaluate the serverless MVE with player behaviors. The non-player

workloads stay constant between iterations to avoid influence on the results and provide conditions for reproducing experiments (R6).

6.4.1 Straight Walk Behavior

The Straight Walk Behavior is to keep walking away from the spawn location towards a fixed direction based on the bot ID, without doing anything else. The walking model emulates bots to explore as many regions as possible in a provided time. Figure 6.2 demonstrates the walking directions, which are star-like. X and Z represent the longitude and latitude coordinates respectively. The number represents the current bot ID, which is increased every time a new bot joins.

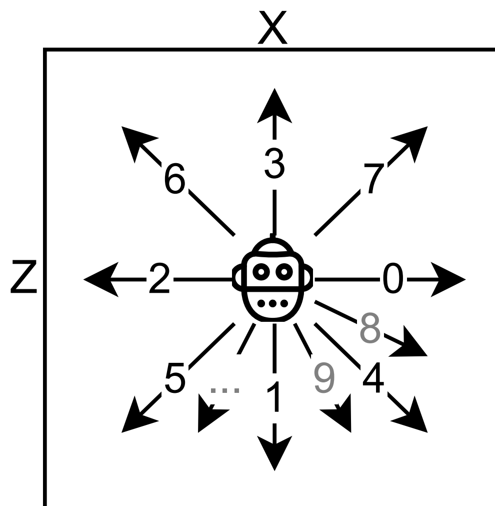

Figure 6.2: Walking Directions of Straight Walk Behavior.

Overall, for bot IDs between 0 and 7, we add a fixed distance to X, Z, or both. The number can be positive or negative depending on the directions. For IDs larger than 7, we add different distances to X and Z, which is a multiplier of the distance. Intuitively, the bot walks towards the intermediate direction of the two nearby bots.

We notice that the pathfinding algorithm may fail because there are too many chunks blocking the way between locations at two ends. The bot stops walking if no available path is found. To prevent the bot from standing still, we set with Java Random a new destination that is close to the source location and set the bot to walk there. After the bot arrives in the new destination, it will attempt to walk towards the fixed direction again.

6.4.2 Random Behavior

The Random Behavior is to perform common gaming actions, e.g., walk, break or build a block, send a chat message, or change inventory in a random fashion. For each idle bot, a random double number between 0 to 1 is assigned, with the current system nanosecond as a random seed. The bot performs different actions based on which range the random number falls into.

Table 6.1: Possibility of Different Actions on Random Behavior.

Possibility	**Action**	**Detail**
40%	Walk	Walk to a random destination with random speed in range [0.1, 0.4]
30%	Break / Build	Break a nearby block, or place a stone on top. Each operation has 50% chance.
20%	Stand	Stand still and do nothing
5%	Send a message	Send a message to all other players
5%	Set inventory	In creative mode, retrieve a random material and hold it on hand

Table 6.1 present the possibility of different actions. Each action takes at least one second. If the action is less than one second, the bot stands still to fill up one second before proceeding next action. In random walk action, there is no boundary for walk destination. Additionally, the walking speed is set to be between 0.1 to 0.4, which covers the average walking speed under different walking scenarios, including slow walk and fast run.

We set different possibilities for two reasons. First, we consider the game property. Previous studies show that walking is an important and major action in games with virtual environments, e.g., Second Life [33] and Pokemon Go [34]. Players walk to their interested areas or walk to a gathering point for events. Then, interaction with the environment, which can be represented by breaking or building a block, is the key element for a game with MVE.

Second, we consider the system resource usage. Different actions result in different system resource usage in the MVE server, for example, walking action requires more resources when the players explore the indefinite terrain, while chatting requires fewer system resources. With a higher possibility of walking and interactions with the environment, we can better compare the resource usage and the variability.

6.5 Metrics Collection

Table 6.2 shows a list of metrics collected during benchmarking. The metrics are grouped based on their types, and the sources where the metrics are collected.

Table 6.2: An Overview of Benchmark Metrics. Legend: MVE Server Instance (SRV), MVE Client (CLT), Serverless Function (FUN), Storage Account (STG).

Metric	Type	Component	Details
CPU Usage	System	SRV, FUN	CPU usage of MVE server
Memory Usage	System	SRV, FUN	memory usage of MVE server
Disk I/O	System	SRV	I/O usage of MVE server
No. of Packets	Network	SRV	Number of network packets
Round Trip Time	Network	SRV	Network latency
Tick Duration	Application	SRV	Time taken to process a tick
No. of Players	Application	SRV	Number of concurrent online players
Function Duration	Application	FUN	Time taken to run a serverless function
Server Latency	Application	STG	Internal latency of cloud storage
End-to-end Latency	Application	SRV, FUN STG, CLT	Time taken finish a task. Different definition on different components.

For metrics originating from MVE servers and clients, they are collected with benchmark client as discussed in Section 6.2.2 and retrieved by benchmark server after the benchmark completes. For metrics originating from cloud platforms, e.g., serverless function and storage service, they are collected with the monitoring API.

7

Experimental Setup

In this chapter, we present our experimental setup. We first present an overview of the conducted experiments and the configuration of every components involved in the experiment. We then discuss each experiment in detail.

7.1 Experiment Overview

We conduct experiment in two parts. First, we conduct microbenchmark (Section 7.4) to study the performance variability of different components on Azure platform. This does not require the game system to start. Table 7.1 shows an overview of microbenchmarking experiments. **Ref** refers to the detailed discussion of the experiment and **MF** refers to the main findings from conducting the experiment.

Table 7.1: An Overview of Microbenchmarking Experiments.

Experiment	**Ref**	**Target**	**MF**
Function Runtime and Life-cycle Variability	7.4.1	Serverless Function	MF1
Blob Storage Latency Variability	7.4.2	Blob Storage	MF2
Storage and Serverless SQL Variability	7.4.3	Blob Storage, SQL	MF3

Next, we conduct benchmark with real game systems (Section 7.5). Table 7.2 shows an overview of macrobenchmarking experiments. **MVE Systems** as target means the experiment is conducted on both the monolithic and the serverless MVE.

We define the monolithic MVE as the unmodified Opencraft server without using any serverless function or cloud storage. While for serverless MVE, we define it as our implemented serverless MVE prototype with cloud storage and all implemented serverless functions enabled.

Table 7.2: An Overview of Macrobenchmarking Experiments.

Experiment	Ref	Target	MF
Effectiveness of File Cache Policy	7.5.1	File Cache Policy	MF4
Scalability of Serverless MVE on One Instance	7.5.2	MVE Systems	MF5
Variability of Fixed Workload on One Instance	7.5.4	Serverless MVE	MF6, MF7
Scalability of Gateway with Multiple Instances	7.5.3	Serverless MVE	MF9
Worse Cases of Player Perceived Latency	7.5.5	MVE Systems	MF8

7.2 Configuration

We conduct experiments on the Azure cloud platform. We use three main cloud services, namely serverless function, storage account, and virtual machines. We present the configuration we use in our experiment. Furthermore, we present the configuration of the MVE server and client.

7.2.1 Serverless Function

Azure does not provide options to directly tune the performance for Azure Functions. The specification of Azure Functions is shown in Table 7.3. We choose consumption plan for all our functions because this plan provides the most common on-demand serverless service. The consumption plan provides a fixed amount of resources, i.e. 1.5 GB RAM and 1 CPU.

Table 7.3: Specification of Azure Functions.

Option	Value
Hosting Plan	Consumption
Runtime Version	3.0.15828.0
Operating System	Linux
Runtime Stack	Java 8.0
Location	Germany West Central

7.2.2 Virtual Machines

We choose *Standard_D4s_v4* as the size of all virtual machines used in our experiments. The plan comes with latest generation D CPU and is for general purpose. Table 7.4 shows the details of specification. From reading */proc/cpuinfo*, the CPU model is Intel(R) Xeon(R) Platinum 8272CL CPU @ 2.60GHz.

Table 7.4: Specification of Virtual Machines.

Option	Value
VM Size	Standard_D4s_v4
vCPU	4
Memory	16 GB
Storage Type	Premium SSD LRS
Operating System	Ubuntu 20.04 x64

7.2.3 Storage Account

Unless stated otherwise, the storage account configuration used for our experiments is premium block blob plan (Discussed in Section 7.4.2) with local redundancy. The default access tier is hot. The account version is V2. Also, all storage requests are communicated with SSL, because it is the default option in the Web today.

7.2.4 MVE Server and Client

For reproducibility, all experiments were conducted on the same world, i.e. the same world seed. We assign for all MVE servers a random world seed *6572920767464630924*, which we retrieve during experiment warm-up. The world is the default type of generation that includes various landscapes. With the same world seed, the same world is presented during each experiment. If we feed the same bot behaviors, the results are the same.

The server authentication service is turned off so that our MVE thin client can emulate players to connect to the server during experiments without prior registration.

For showing in-game graphic content, we use the official Minecraft:Java Edition client. The version is 1.12.2. All configurations are default.

7.3 Metrics

Table 6.2 shows a list of metrics collected during benchmarking. Our evaluation focuses on three types of metrics: metrics of cloud services provided by the platform, system and network metrics of the virtual machines used for running MVE services, and application metrics retrieved from MVE servers and clients. In this section, we discuss important metrics and define game states.

7.3.1 Latency: Using Cloud Services

We discuss the important metrics of cloud services we use in our implementation. The metrics are provided by either *Application Insights* or *Azure Monitor*.

Function Duration. Two factors directly impact the performance of a serverless system: the execution time of each invocation, and the latency to invoke the function. The first factor is the internal runtime of a function, which is determined by the underlying hardware of the serverless platform. The second factor includes the latency of network and the latency that the serverless platform routes the requests to specific function handlers. The two factors sum up to end-to-end latency of invoking a serverless function.

Storage Account. Azure Monitor for storage account provides two key metrics: server latency, which is the latency of processing a request internally, and Azure end-to-end latency, which is the latency between a request is received from a client and an acknowledgement is received from the client. However, we notice that Azure does not provide raw data points for these two metrics. One of the aggregators, including maximum, minimum, average, and total, must be applied for a time interval. The minimum interval is one minute.

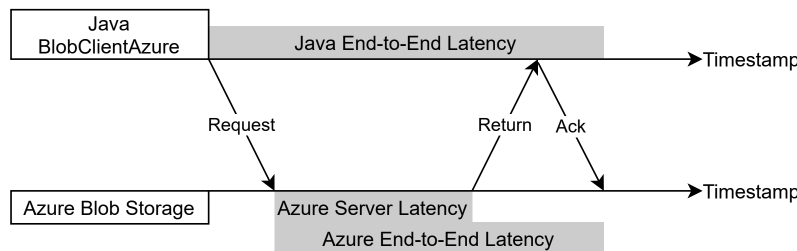

Figure 7.1: Different Latency Metrics with Timestamp.

Additionally, we present Java End-to-End Latency, which is the latency between a request is sent from blob client and the file is received from blob storage. Figure 7.1 shows three latency metrics on timestamp for one request, where the grey blocks indicate the included events.

7.3.2 Maximum Players: Scalability of the System

One common metric to measure the scalability of a system is the number of users. In this work, we measure scalability as the maximum number of concurrent players performing actions. We compare two systems with an increasing number of emulated players performing actions based on the behavior models. If one system supports more players before it

is overloaded than the other one, we say that the scalability is improved on this system. The approach is widely used to measure the scalability of interactive systems, e.g., online games [35] and web applications [36].

In practice, we approximate the maximum number of concurrent players as the number of players at the time before the system is overloaded. The reason is that the system cannot connect more players while providing the guarantee of QoS when it is overloaded. For game instances, we use tick duration to indicate the overload state (Discussed in Section 7.3.3).

7.3.3 Tick Duration: Overload State

We define overload state as the first tick that is over 50 ms with a rolling average of 50 ticks, which are 2.5 seconds if the server is not overloaded. The reason is two-fold. First, the server is run at a tick rate of 20 Hz, which is invoked every 50 ms. If a tick does not finish within 50 ms, the next tick is delayed, which may leads to delayed of overall game update.

Second, instantaneous ticks over 50 ms do not immediately cause the server overloaded, because the next tick duration may be significantly lower, which brings the game loop to normal and is not noticeable to the players. Thus, we apply a rolling average to the data and focus on the values over 50 ms.

7.4 Microbenchmarking

In this section, we discuss the mirobenchmarking experiments in detail. We evaluate the performance variability of serverless function as per invocation intervals, and the variability of different events involved in the life-cycle of invoking a serverless function (Section 7.4.1). We compare the performance between three cloud storage solutions: standard and premium blob storage account (Section 7.4.2), and serverless SQL (Section 7.4.3).

7.4.1 Function Runtime and Life-cycle Variability

In this experiment, we evaluate the performance variability of Azure Functions by sending the same requests to the target functions multiple times in a fixed interval. The requests are sent from Azure VM. To avoid interference, all functions with timer triggers are disabled. Also, we pre-warm the worker so that the result does not include initialization time. As we feed the same request during every invocation, the latency is expected to have low variability in ideal conditions.

We conduct experiment on function *TerrainGeneration*. The request is constructed with the same coordinates x and y, world seed, and other stateful objects. The function generates the requested chunks and returns the same chunk data upon each request. The function only performs internal computation and does not involve other components on the cloud platform.

During the experiment, two performance metrics are logged: the function runtime (duration) of each invocation, which is provided by Application Insights, and the end-to-end latency of a function invocation. The experiment is run with three invocation intervals: five seconds, one second, and 0.1 seconds. The function is invoked 720 times per invocation interval.

Additionally, We look into the events during the life cycle of a function invocation. Although several events occur when a function is called on a pre-warmed worker, Azure Application Insights only provide logs for three events, namely *Executing*, *Invoked*, and *Executed*. However, it does not provide runtime for these events. We manually calculate the runtime of the first two events, as the time difference of one event and its next event.

7.4.2 Blob Storage Latency Variability

In Azure Blob Storage, a basic setting directly impacts performance: standard and premium performance. In this experiment, we study the impact of reading both small and large files on two performance plans. We do not consider the write performance because write operation can be fully asynchronous in our serverless MVE prototype, which does not impact the overall performance. While for some cases, the players must wait for the read operation to complete, which directly impacts the overall performance.

The experiment is to download the same files with fixed intervals from storage accounts with standard and premium settings respectively. We consider use cases with small and large files on the serverless MVE prototype. For small file, we choose player metadata *banned-players.json*, which is 646 Bytes. For large file, we choose region data *r.-1.-2.mca*, which is 36.68 Megabytes.

The experiment is conducted on Azure VM with a customized Java program which calls *BlobClientAzure* class (Discussed in 5.1). Considering the limitation of Azure Monitor, we run experiments with a one-minute interval and retrieve the values with total aggregator every minute. The total value is the raw data because there is only one download activity in this minute, which involves multiple requests, and their sum represents the overall latency for this download activity. For each type of file, we download 120 times.

7.4.3 Azure Storage and SQL End-to-end Latency

We conduct an experiment to compare the performance between Azure premium storage account and serverless SQL plan. We invoke the two storage back-ends with a Java program that invokes storage back-end classes (Discussed in 5.1). The program retrieves the same player data from both blob storage and SQL back-end.

The duration between data request is sent and data is retrieved from storage back-end is the end-to-end latency. We conduct experiments with request intervals: 0.1 second, 1 second, 5 seconds, and 2 hours so that we can study performance per access frequency. We log the end-to-end latency.

Unlike storage account, Azure monitor only provides the percentage of CPU and I/O usage for SQL, which cannot represent the internal performance because we cannot relate the resource percentage to processing latency. Thus, we study the performance by analyzing the end-to-end latency.

7.5 Macrobenchmarking

In this section, we discuss the macrobenchmarking experiments in detail. We evaluate the effectiveness of the file cache policy for reading data from cloud storage (Section 7.5.1). We evaluate the scalability of the serverless MVE with one instance (Section 7.5.2) and gateway with multiple server instances (Section 7.5.3) by feeding an increasing amount of workload. We study the performance variability of the serverless MVE by feeding a fixed workload (Section 7.5.4). Finally, we evaluate the worst cases of end-to-end latency perceived by players during events (Section 7.5.5).

7.5.1 Region File Cache Policy

We implement a three-layer cache policy for I/O operation of the region file (Discussed in Section 5.1.1). We conduct an experiment to study the overhead of using cloud storage and the effectiveness of the implemented cache policy.

The experiment is run with one server instance on Azure VM. We log the time of server event which reads a chunk from the region file. From here our implemented I/O service tries to look it up from cache. In case cache is hit in either memory or local file, the latency is lower than 50 ms, which is an acceptable latency for an MVE game. In case of cache miss, which means that the player has to wait for the file to be retrieved from remote, the latency will be over 100 ms, leading to degradation of player experience.

We assign eight bots to join at the start and walk ten minutes with Straight Walk Model (Discussed in Section 6.4). Prior to the actual experiment, we pre-run the experiment to generate the corresponding region files, which will be accessed with the same bot activities. Each region file is at least 1 Megabyte after preparation. The generated region files are stored in blob storage and are re-used for every iteration.

We conduct actual experiments with three configurations, namely local storage, remote storage without cache, and remote storage with simple distance cache. For experiments on remote storage, all local file cache is deleted before each experiment so that we include the latency of downloading the files in our results. Autosave is disabled so that the files will not be changed. The experiment is repeated four times for each configuration, and we log the latency with *EventLogger* in Opencraft server. The events during server initialization are excluded from the results as they are not perceived by players.

7.5.2 Increasing Workload for MVE Systems with Single Instance

We compare the system resource usage between the monolithic MVE and the serverless MVE prototype, to see if the serverless technology benefits MVE.

We aim to get insight into how the two systems handle increasing workloads and how many players the systems can support before overloaded. We first feed the server with Straight Walk Behavior workload. Fifty bots gradually join the server at a ten-second interval. Each bot walks away from a fixed spawn location at a fixed speed with Straight Walk Model after it joins (Discussed in Section 6.4.1). We use two walking speeds, three blocks per second, which is walking slowly, and eight blocks per second, which is running. Each iteration runs for ten minutes.

Next, we feed the server with sixty bots performing Random Behavior. The bots gradually join the server at a ten-second interval. After a bot joins, it performs random behavior based on the random number during each task (Discussed in Section 6.4.2). Twenty iterations are run for the random behavior workload and average cases are considered.

7.5.3 Increasing Workload for Serverless MVE with Multiple Instances

In this experiment, we evaluate the scalability of our gateway prototype with multiple server instances. We run experiment with five configurations: 1 server without gateway, 1 server with 1 gateway, 2 servers with 1 gateway, 4 servers with 1 gateway, and 4 servers with 2 gateways. All serverless features are enabled for the components.

We feed an increasing number of bots performing Straight Walk behavior, and log the maximum supported players before any one of the components is overloaded. As the gateway is event-driven and does not have a tick loop, we approximate the overload state with CPU resource usage, which is correlated with tick duration with MVE servers as discussed on 8.5.

Each experiment is repeated five times and we consider the mean value of maximum supported players of all iterations.

7.5.4 Fixed Number of Players for Serverless MVE

We study the performance variability of serverless MVE with a single server instance by feeding a fixed number of players. We conduct the experiment on the serverless MVE system, as defined in Section 7.5.2.

During each iteration, twenty bots join the server together and perform Straight Walk and Random behavior for ten minutes. The straight walk behavior produces an almost same routine under the same world with a fixed seed, thus the workload is considered to be fixed. With a high number of repetitions, we expect that the variability can be generalized. While random behavior produces different results on different iterations because of the random nature.

For each behavior, the experiment is run for fifty iterations. We study the variability within iterations and between iterations. The bots do not cause the server to be overloaded under both behavior models as discussed in Section 8.5.

7.5.5 Worst Cases of Player Perceived Latency

Although latency hiding policy is applied for most serverless function invocation and cloud storage access, some parts of requests remain synchronous in the aspect of players, i.e. the player must wait for the executions to finish before they can observe the effect.

For example, when players log in, the server retrieves player data asynchronously, which does not impact the overall performance of the server. However, in the aspect of players, they must wait until the player data is retrieved and verified before they can play. The additional wait time is the overhead of using serverless technology.

Another example is the additional latency of player operation after the command is sent. The server offloads the execution to serverless function asynchronously, however, there is additional time for the system and the players to observe the effect of commands.

We conduct an experiment to study the overhead. We assign twenty bots who join the games and perform Random Behavior. We log the end-to-end latency of login, which is the latency between the time when the bot sends a login request and the time when the player is in game state.

One of the bots is the game master. Every thirty seconds, it bans a random player or check the list of banned players. We log the end-to-end latency between the time when the game operator sends the command and the time when the game operator observes the effect.

The experiment is run on both monolithic and serverless systems so that we can study the overhead of using serverless technology.

8

Evaluation

We conduct experiments shown in Table 7.1. In this chapter, we present and discuss the results we collect from conducting these experiments. We first present an overview of our main findings, then we discuss them in detail.

The main findings (**MF**s) of micro benchmarking are:

MF1 The performance variability of serverless functions is high under all invocation intervals on pre-warmed function instances. The variability is caused by both the internal process of the function and the latency to invoke it. The maximum latency to invoke the function is up to three times the mean value.

MF2 All storage back-ends on Azure platform present high performance variability caused by both the internal process and the network latency. As a result, latency hiding policies are necessary when using cloud storage. There are large differences in performance on different storage back-ends, for example, premium storage account outperforms standard one in terms of lower and more constant latency.

MF3 Azure Storage Account outperforms Azure SQL Serverless Plan as storage back-end, in terms of lower overall latency, lower variability, lower cold start time, and lower cost. We conclude that we should use Premium Storage Account as our storage back-end.

The main findings (**MF**s) of benchmarking the real system are:

MF4 The game world becomes invisible in case of high reading latency from cloud storage and the player has to wait, which significantly impacts the gaming experience. Our three-layer cache policy for reading data from region files successfully hides the latency of accessing remote data and mitigates the overhead.

MF5 Serverless architecture improves the scalability of MVEs. Compared to the monolithic system, the serverless MVE prototype with one server instance supports more players before the server is overloaded, which is mainly constrained by CPU resources. The improvement varies between 51% to 167%, depending on player behavior.

MF6 Player behavior affects system performance variability. For a fixed straight walk workload, the trending of tick duration is highly similar between iterations, with a different variance on actual values caused by the shared CPU resource. Meanwhile, there is no pattern on the trending of tick duration between iterations with random behavior workload.

MF7 The high variability of latency on serverless functions does not negatively affect the performance of serverless MVE, with effective latency hiding policies including resource preloading and asynchronous invocation. Instead, the performance variability of serverless MVE is mainly caused by the internal process of the server.

MF8 Players observe additional mean latency between 3% to 518% on the serverless MVE depending on event types. Under worst cases, the latency can take up to 6.4 seconds, caused by cold start of serverless function. We conclude that it does not significantly impact player experience based on previous survey on waiting time.

MF9 Connecting players through a gateway improves the scalability of the game. We find that all gateways handle the same amount of workloads, while game instances handle different amount of workload at the same time, which indicates that we need a more efficient policy to split players into different instances.

8.1 MF1. High Function Runtime Variability

Figure 8.1 shows the experiment result of function *TerrainGeneration*. The y axis represents the latency in milliseconds. The x axis represents the invocation interval. The box shows the range between the 25-th and 75-th percentile. The outliers are defined as outside 1.5 times the interquartile range. The boxes are grouped based on the invocation intervals as shown in x axis.

We can see that there are many outliers on both metrics under all invocation intervals. Specifically, the maximum value is up to 3.18 times the median value on five-second interval. This indicates that performance variability is high when the function is executing the same amount of workload.

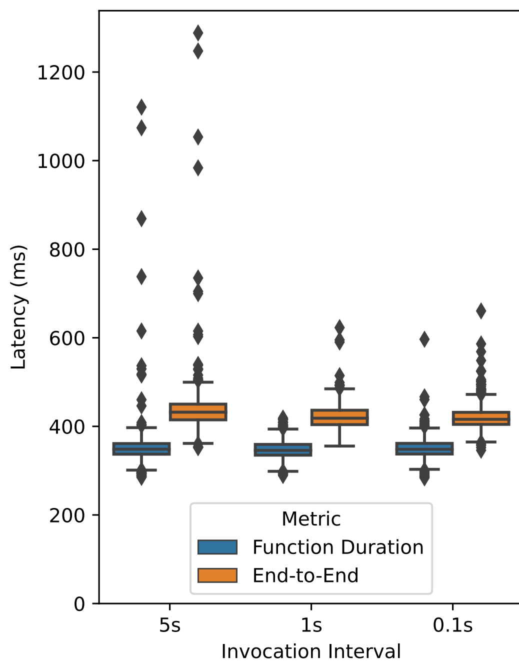

Figure 8.1: Function Performance Metrics (*TerrainGeneration*).

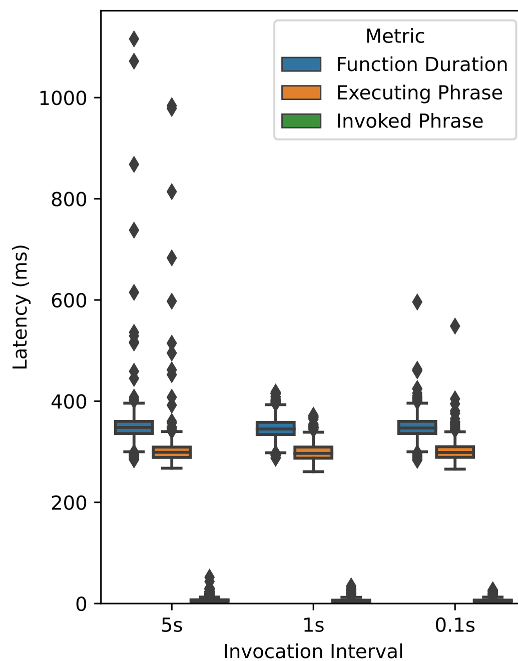

Figure 8.2: Latency of Events During Function Invocation (*TerrainGeneration*).

The performance is better and the variability is lower when the invocation interval is shorter. Figure 8.2 shows the latency of events during the life-cycle of function invocation. When the invocation interval is five seconds, there are many outliers on *Executing* and *Invoked* phrase, with maximum latency over three times the mean value. The outliers are fewer and lower with one second and 0.1 seconds.

The overall function duration is mainly introduced by *Executing* phrase, which schedules and routes incoming HTTP requests to function handlers. Azure does not provide details of how it works. We conjecture that the underlying scheduling system sets a higher priority for functions that are accessed more frequently, which results in requests being routed faster to the function handler.

8.2 MF2. High Latency Variability on Blob Storage

Figure 8.3 shows the box plot of performance metrics of downloading a small file. The outliers are defined as outside 1.5 times the interquartile range. The graph shows that for both service plans, Azure server latency and Azure End-to-End latency metrics are highly similar, which indicates that the latency is mainly introduced by Azure internal process,

instead of network.

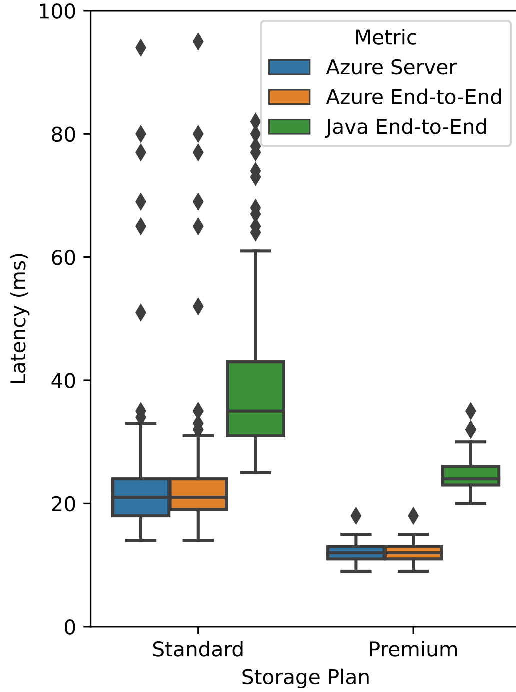

Figure 8.3: Latency of Downloading Player Data (646 B) from Blob Storage.

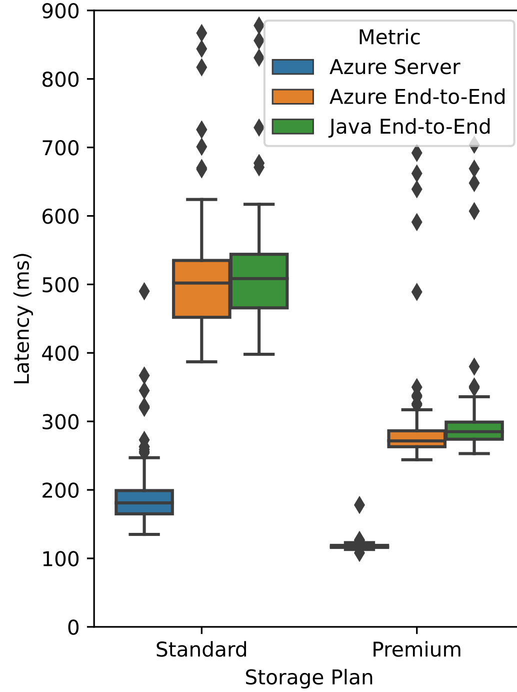

Figure 8.4: Latency of Downloading Region File (36.68 MB) from Blob Storage.

Figure 8.4 shows the performance metrics of downloading a large file. In contrast to the small file, the overall latency is mainly introduced by Azure End-to-End latency, which is highly impacted by network. The result is expected because it takes more time to actually transfer a large file than preparing it. Among the two service plans, premium outperforms standard in all three metrics.

The performance variability is very high for downloading a small file in standard plan. The maximum value is five times higher than the average one. In premium plan, the performance variability is lower, with maximum value doubled as the average one. For downloading a big file, both service plans show high overall performance variability. In standard plan, the variability is caused by both internal processing and network. While in premium plan, the variability is mainly caused by network. The result matches with Azure's service level agreement, which guarantees significantly lower and more consistent internal latency [37] on premium plan.

When downloading a small file, Java End-to-end latency is significantly higher than Azure End-to-end latency in all cases. We look into network packet trace and calculate

the time spent on different events. Figure 8.5 shows the stacked bars of different events for downloading small and big files on two storage plans, where the time spent on sent request is the time between the client sends the packet and the client receives the first TCP ACK from server. The time spent on received messages is the time to receive the actual files from the server. Other packets refer to other packets during the TCP connection.

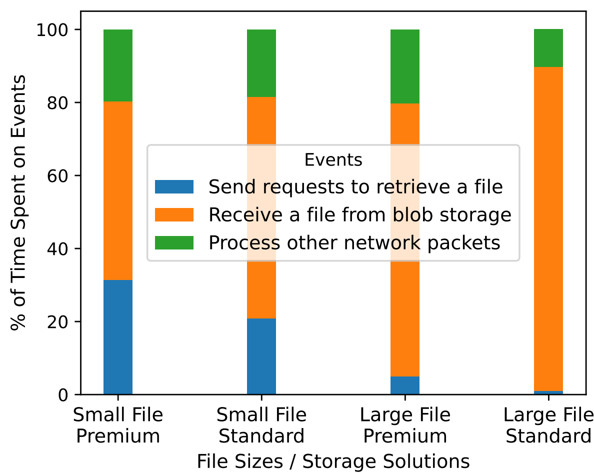

Figure 8.5: Percentage of Time Spent on Different Events when Downloading Files from Blob Storage.

The time spent on *Send* is the extra latency in addition to Azure End-to-End latency. For downloading a big file, the introduced latency is not significant as it only takes a small percentage of the overall latency. The result is expected because the time to send requests for retrieving files from remote storage is basically the same, regardless of file sizes, thus it takes a larger percentage if the file size is smaller, where the time to download the file is smaller.

Azure defines different price policies for standard and premium plans [38]. To summarize, for hot data, premium plan is around 9.9 times more expensive than standard in storage but 41% cheaper in request. In terms of 95th percentile latency, premium plan has 48% reduction in overall latency for downloading a big file. The number is 63% for downloading a small file. Considering the I/O services in a serverless MVE, most files are small and frequently accessed, which benefits from the premium plan pricing and the performance improvement. Thus, we conclude that we should use premium plan of storage account on a serverless MVE for better performance.

Last, due to the performance variability of cloud storage, which does not provide a consistent time for accessing a remote file and the overhead of the network stack, we

should apply latency hiding policies for reading remote files whenever possible.

8.3 MF3. Premium Storage Account Outperforms Serverless SQL

Figure 8.6 shows the box plot of end-to-end latency to retrieve the same small data (player data) from Azure premium storage and serverless SQL with 0.1-second, 1-second, 5-second, and 2-hour request intervals respectively. For better visualization, we split the y-axis into two parts, where the y limit of the upper part is between 140 to 90000, while the lower part is between 0 to 140.

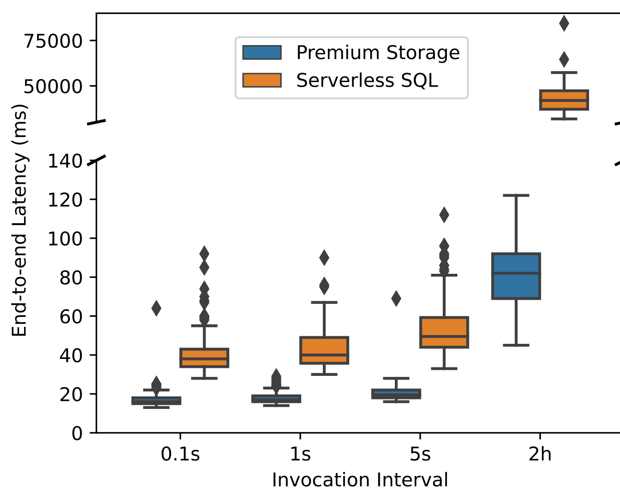

Figure 8.6: End-to-end Latency to Retrieve the Same Data from Two Storage Back-ends (Premium Storage Account and Serverless SQL).

The performance is similar on premium storage across all invocation intervals, and the variability is low, which matches our results on Section 8.2. We look into the mean values of end-to-end latency across all invocation intervals except 2-hour. Surprisingly, the value is better in Premium Storage than in Azure SQL under all cases. The mean value for Premium storage back-end is 18.7 ms, while it is 45.9 ms for Azure SQL. Premium Storage account archives 59.3% lower overall latency than serverless SQL.

The end-to-end latency with two-hour invocation interval indicates the cold start time. The mean value is 82 ms on storage account, which is three times the value without cold start. However, the mean value is 42.4 second on serverless SQL, which is over 900 times

the value without cold start. This indicates that the cold start time is significantly high on serverless SQL, which impacts the server performance.

Although the two back-ends utilize different pricing policies, i.e. storage account charges based on requests, while Serverless SQL charges by CPU seconds, we can compare the approximate cost. For reading data 10,000 times, premium storage account charges for 0.0020 EUR. We approximate the reading frequency as one request per second, which results in a cost of 1.34 EUR on Serverless SQL. In the aspect of data reading, Serverless SQL charges 670 times higher than storage account. Both storage back-ends charge for space based on per-GB pricing, which is 0.16445 EUR for premium storage and 0.15 EUR on Serverless SQL. Overall, Serverless SQL charges slightly lower in space, but much higher in reading data.

Supposed there are one million of players in the system, which are approximately 700GB of data, and there are one hundred thousands of login events for ten hours every day, we calculate the total cost of one month. The price of storage is 115 EUR on premium storage and 105 EUR on serverless SQL. The price of accessing is 0.6 EUR on premium storage and 145.2 EUR on serverless SQL. The total cost is 115.6 EUR on premium storage and 250.2 on serverless SQL. The serverless SQL is 53.8% cheaper.

We conclude that we should use Azure Premium Storage Account as the storage back-end for the Serverless MVE. Compared to Serverless SQL plan, storage account provides better overall performance, lower variability, lower cold start time, and lower cost.

8.4 MF4. Region File Cache Policy Successfully Hides Remote Reading Latency

Table 8.1 shows the maximum, 99.67-th percentile, 99-th percentile, and minimum latency results of reading a chunk from I/O service, with different storage settings, including local storage, remote storage without preloading policy, and remote storage with simple distance policy.

Table 8.1: Different Percentiles of Latency of Reading a Chunk from Region File When Using Different Storage Settings. All Values are Presented in Milliseconds.

Storage Setting	Max	99.67-th	99-th	Min
Local Storage	26.6	9.3	4.6	0.0054
Remote Storage, No Preloading Policy	464.3	114.8	18.1	0.0728
Remote Storage, Simple Distance Policy	41.6	23.9	16.3	0.0715

When using remote storage without preloading policy, around 0.033% of chunk reading events take more than 100 ms. Although the number is small, we argue that the high latency should be mitigated for two reasons.

First, it significantly impacts game experience. When a player moves to a chunk whose region file is not loaded within 100 ms, the world becomes invisible to the player. Figure 8.7 demonstrates the in-game graphic of such an event with Minecraft client. The players cannot perform any actions and have to wait before the world becomes visible again.

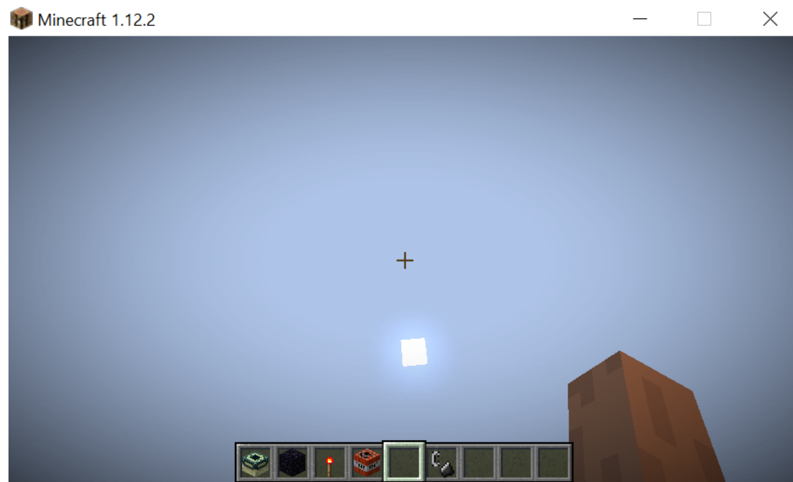

Figure 8.7: In-game Graphic When World Becomes Invisible to Players.

Second, if we consider the worst case in the real world, the occurrence is frequent. Assuming a player sprints in a surface without blocking, he can go across five blocks per second. It takes around seven seconds to traverse a region file with a straight line. If there is no preloading policy, and there is no other player that has previously visited the edge, the region file needs to be retrieved from the remote. If the time to retrieve the remote region file is longer than 100 ms, the player will see an invisible world every seven seconds.

With our simple distance preload policy, the 99.67-th latency is 23.9 ms, and the maximum latency is 41.6 ms, which is an acceptable number. This indicates that our policy successfully hides the remote cloud storage latency from players under normal movement behavior.

Additionally, the local storage performs best with 99th latency at 4.6 ms and maximum latency at 26.6 ms. The latency variability of remote storage with preload policy is higher than local storage. The reason is that more I/O operations are involved when using remote storage, resulting in overhead. For example, when the server is reading a chunk from one region file, another thread may be downloading another region file from remote storage

for preloading purposes. This leads to I/O interference, resulting in higher variability in remote storage with preloading policy than local storage.

8.5 MF5. Serverless Improves the Scalability of MVE

The scalability of MVE is measured as the maximum number of players before the server is overloaded. Figure 8.8 shows the tick duration with increasing straight walk workload where bots walk at a fixed speed of three blocks per second. We apply a fifty-tick rolling average, which is 2.5 seconds if the system is not overloaded, to tick duration figures. The span in red indicates that the system is overloaded and no longer player-able, which is defined as the first tick that is over 50 ms with a rolling average.

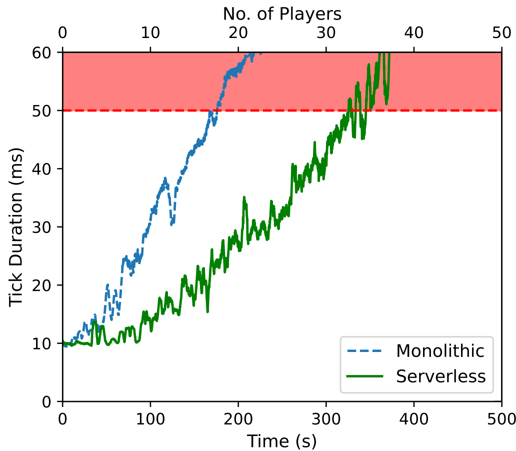

Figure 8.8: Tick Duration with Increasing Straight Walk Workload.

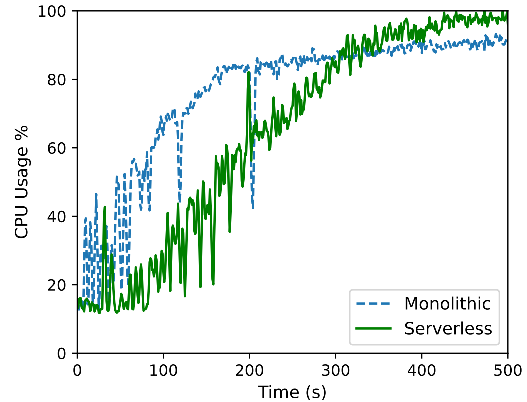

Figure 8.9: CPU Usage with Increasing Straight Walk Workload.

The monolithic system starts to overload at 170 seconds, which has seventeen bots. The serverless system starts to overload at around 330 seconds, which has thirty-three bots. This indicates that the serverless system support 94% more players than the monolithic system before the server is overloaded.

Figure 8.9 shows the normalized CPU usage of the monolithic system and serverless system with increasing straight walk workload. Every 25% represents a fully utilized core. The CPU usage is increasing for both systems, which is expected as system resource is increasing with more workloads.

While matching the tick duration with CPU usage, we can see that for the monolithic system, the CPU usage reached 82% when the system is overloaded, and continue to slowly increase overtime. For the serverless system, the CPU usage reaches 90% when the

system is overloaded, and continue to increase to 100%. This indicates that the maximum supported player is limited by CPU resource, and our serverless system successfully offloads a part of heavy computation task to serverless functions.

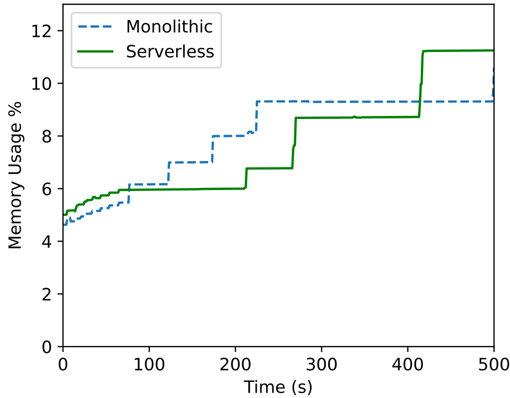

Figure 8.10: Memory Usage with Increasing Straight Walk Workload.

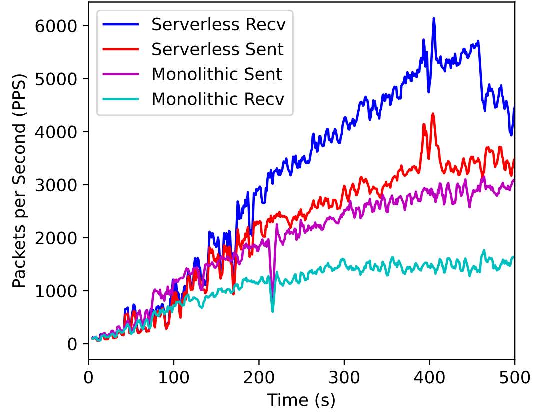

Figure 8.11: Packet Per Seconds with Increasing Straight Walk Workload.

We look into other system resource usage. Figure 8.10 shows the memory usage in percentage for two systems. The memory usage is increased when more bots join the game and more chunks are explored, with the maximum at 12%. This indicates that memory usage is not the factor that impacts the overall system performance. Also, there is no significant difference between the two systems. We conclude that the serverless system does not impact memory usage.

Figure 8.11 shows the packet per second for both sent and received directions of the two systems. The serverless system has higher PPS in both directions, especially in receive direction. This is expected because the serverless system receives data from remote services, e.g. cloud storage and serverless function.

Next, we look into the maximum number of players with increasing run workload where bots run with a fixed speed of eight blocks per second. Figure 8.12 shows the tick duration of the experiment. The maximum number of players is twelve on the monolithic system, which is lower than the slow walk workload. While the number is thirty-two on the serverless system, which is similar to the slow walk workload. The serverless system supports 167% more players.

The change in the maximum supported players is reasonable. When the bots walk at a faster speed, more terrain is explored within the same duration, resulting in heavier computation on the monolithic system caused by terrain generation. This part is offloaded

to serverless function on the serverless system, thus the maximum number of supported players is similar.

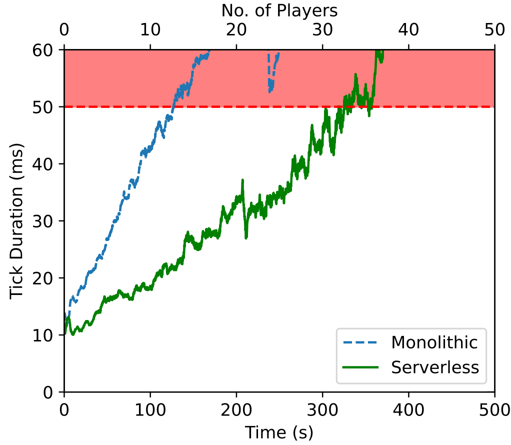

Figure 8.12: Tick Duration with Increasing Straight Run Workload (8 Blocks / Second).

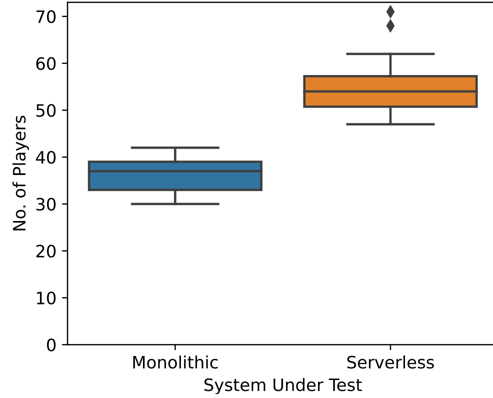

Figure 8.13: Number of Players Before the System Under Test is Overloaded with Random Behavior.

Finally, we look into the maximum number of players on two systems with Random Behavior workload. Figure 8.13 shows the number of players before the system under test gets overloaded among twenty iterations. The y axis represents the maximum number of players before the server is overloaded. The x axis represents the system under test. The box shows the range between the 25-th and 75-th percentile. The whiskers represent the lower 25% and upper 25%. The outliers are defined as outside 1.5 times the interquartile range.

There exists high variability on maximum supported players, which is expected because of the random nature. Overall, the serverless system performs better. If we consider the medium value, which is 37 for monolithic system and 56 for serverless system, the serverless system can support 51% more players than the monolithic system. The improvement is less than Straight Walk Behavior, where terrain generation consumes most system resource.

8.6 MF6. High Variability of Serverless MVE

Figure 8.14 shows the tick duration with a rolling average of 50 ticks of all iterations of Straight Walk and Random behaviors. Figure 8.15 shows the normalized CPU usage of all iterations on two behaviors. We highlight three iterations, 1, 23, and 47 to get more insights.

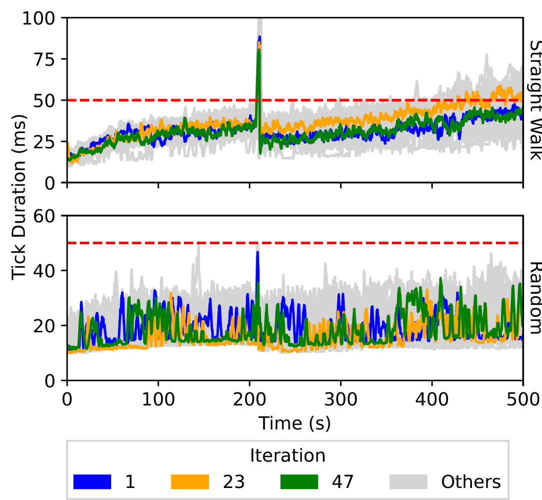

Figure 8.14: Line Plot of Tick Duration of All Iterations.

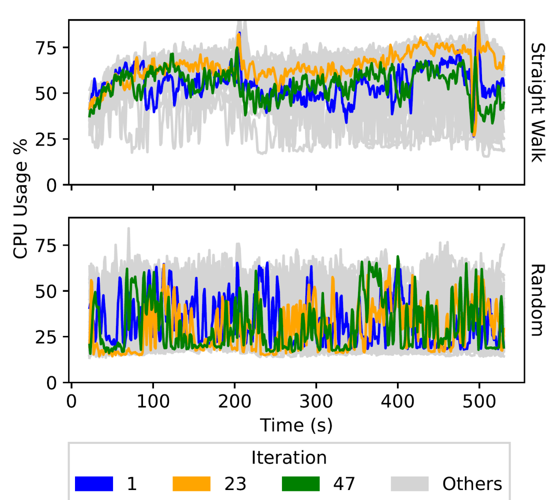

Figure 8.15: Line Plot of CPU Usage of All Iterations.

Player behaviors impact the variability over time within one iteration. With Straight Walk behavior, the trending of tick duration among iterations is similar, with different variance on the actual tick duration values. At the start, the tick duration goes up as players join the server. Then it becomes stable. At around 210 seconds, there is a spike among all iterations. After that, the tick duration becomes stable with a different degree of variability.

Except for ticks at 210 seconds, most ticks stay under 50 ms, which indicates that the server is not overloaded most times. Some iterations, for example 23, have ticks over 50 ms duration. As shown in Figure 8.15, the value of tick duration is highly correlated with CPU usage. We conjecture that this is due to the shared CPU resource, which provides different performance over time.

While with Random behavior, the tick duration goes up and down without pattern among all iterations. The result matches the random nature. Also, not all iterations present the spike at 210 seconds, and the spike values are lower than Straight Walk Behavior.

The spike at 210 seconds is caused by *unloadOldChunks* event, which unloads chunks that are no longer within view distances of players from memory. The event is called every five minutes on the server, which matches the time after adding 90 seconds which were excluded due to server initialization. If there is a large number of chunks to be unloaded, the system is slowed down. At straight walk behavior, there are many chunks to be unloaded as the players are walking towards a fixed direction. While at random behavior, it is uncertain

whether the players walk away from the spawn location, thus the spike does not always exist. This indicates that a more efficient scheduler is needed to unload old chunks.

8.7 MF7. Performance Variability of Serverless Functions Does Not Negatively Impact Serverless MVE

Figure 8.16 shows the box plots of tick duration for all iterations under two behavior models. The x-axis is the behavior model, and the y-axis is the raw data point of tick duration in milliseconds. The y-axis is broken into two parts, where the upper part indicates values between 250 to 3100 ms, and the lower part indicates values between 0 to 250 ms. The box shows the range between the 25-th and 75-th percentile. The outliers are defined as outside 1.5 times the interquartile range.

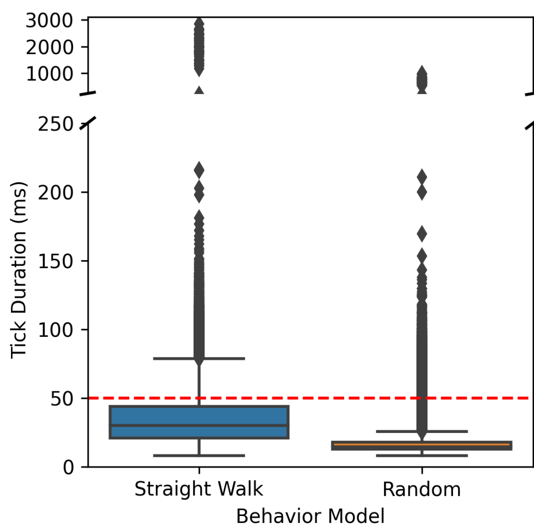

Figure 8.16: Box Plot of Tick Duration among All Iterations.

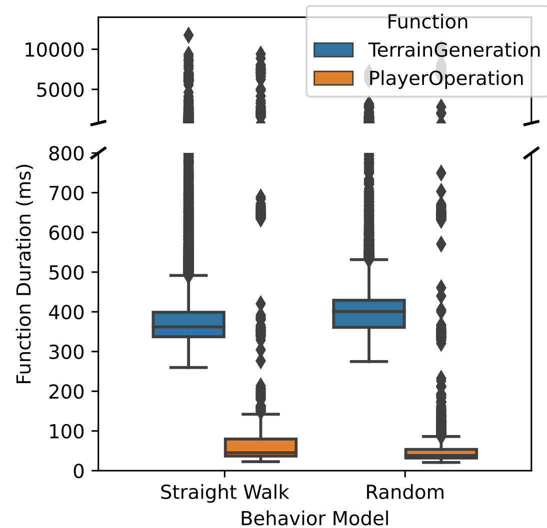

Figure 8.17: Box Plot of Function Duration among All Iterations.

There are fifty values on each behavior among all iterations on the upper part. We look into each iteration and find that there is exactly one such value on each iteration, which is caused by *unloadOldChunks* as discussed in Section 8.6.

Except for the extreme values on the upper part, the variability of the values on the lower part is high if we consider the outliers. There are more outliers on random behavior, which indicates that it presents more performance variability.

Figure 8.17 shows the function duration of two functions among all iterations under two behavior models. The x-axis is the behavior model, and the y-axis is the raw data point

of function duration in milliseconds. The y-axis is broken into two parts, where the upper part indicates values between 800 to 14000 ms, and the lower part indicates values between 0 to 800 ms. The configuration of boxes are same as Figure 8.16.

We can seet that the variability is high, which matches the result in Section 8.3. There is no siginificantly different between two behavior models in function duration. Extreme values exist on the upper part, with maximum values 11.7 seconds with Terrain Generation function and 10.1 seconds with Player Operation function. These extreme values are caused by cold start of function instance, which is started automatically by Azure Functions to support more requests.

Despite the high extreme values and high variance in the performance of functions, it does not directly impact the tick duration of the game (lower part of 8.16), which does not correlate with function duration. The reason is that we apply proper latency hiding policies when invoking the serverless functions, i.e. preloading and asynchronous invocation. We conclude that the performance variaibility of functions does not negatively impact the MVE system.

8.8 MF8. Worst Cases of Player Perceived Latency Does Not Significantly Impact Gaming Experience

Figure 8.18 shows the end-to-end latency of six events, namely login, show ban list, ban, and unban an player, set weather to clear, and set weather to thunder under the monolithic and serverless MVE systems. For better visualization, we break the y axis and set different scales for upper and lower parts.

Compared to the monolithic system, the latency under the serverless system is higher and the variability is higher for all events, which is caused by the additional latency between the server and remote services, and the variability of the serverless function and cloud storage (Discussed in Section 8.1 and 8.2).

Considering the mean values, the serverless system adds 134%, 456%, 88%, 518%, 3%, and 24% for login, list ban, ban, unban, set clear weather, and set thunder weather events respectively. The additional latency for set weather events is lower than others. The reason is that local environment simulation is enabled to return simulation results immediately, while retrieving results simulated by serverless functions with local simulation offset (Discussed in 5.2.3). For other events, there is no local simulation because the data is only stored and verified remotely for consistency.

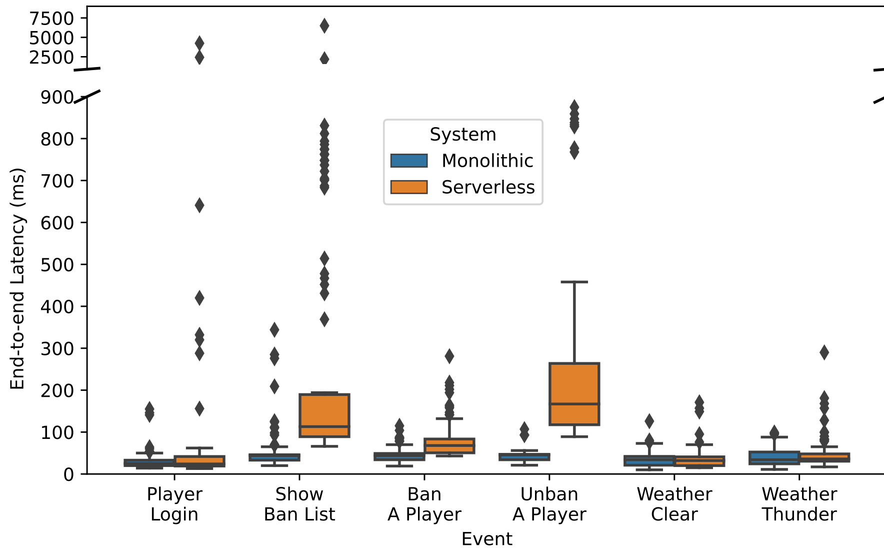

Figure 8.18: End-to-end Latency of Events Measured at Client.

We also see various box sizes for different events, which indicates different degree of performance variability. The reason is that different event types involve in different number of invocations of cloud services. For *ban a player* event, there is one invocation of remote service: the game instance submits the ban request to serverless function. The serverless function returns immediately while writing the ban data in blob storage. For *unban a player* event, there are three invocations: the game instance checks whether the player is banned through the serverless function, which waits for data from blob storage, and instruct actual unban if the ban data exists. With more cloud services involved, the performance variability is higher.

For *show ban list* event, the execution time is different during each invocation, because the number of banned players are different. Thus it also has high variability. The performance variaibility is caused by the internal process.

If we consider the worse case, it is 6499 ms on banlist event and 2425 ms on login event, which are caused by cold start of function instances.

Although the additional latency looks high, we conclude that the latency is acceptable for these events and that serverless technology benefits MVE for its performance gain for two reasons. First, because the events and function invocations are handled asynchronously, the additional latency is only observed by players. It does not slow down the server. Thus, the server benefits from the performance gain as discussed in Section 8.5 without being impacted by these events.

Second, except for login, the events do not block gameplay, i.e. the players can perform other in-game activities while waiting for the results of the commands. The player experience is not degraded while waiting for the results.

For login events, players must wait until the event is complete before they can start gameplay. During the process, the player is presented with a loading screen. According to a discussion on forum [39], players can tolerant between thirty seconds to a few minutes on the loading screen. In our case, even with function cold start, the latency stays under the range. Thus, we conclude that the overhead does not significantly impact player experience.

8.9 MF9. Gateway with Multiple Instances Improves Scalability

We measure the scalability of gateway with multiple instances as the maximum number of players before any worker is overloaded. Figure 8.19 shows the maximum number of supported players performing Straight Walk behaviors with an increasing number of workers. We present the mean values among five iterations for each worker configurations. The x-axis indicates the number of workers and their roles.

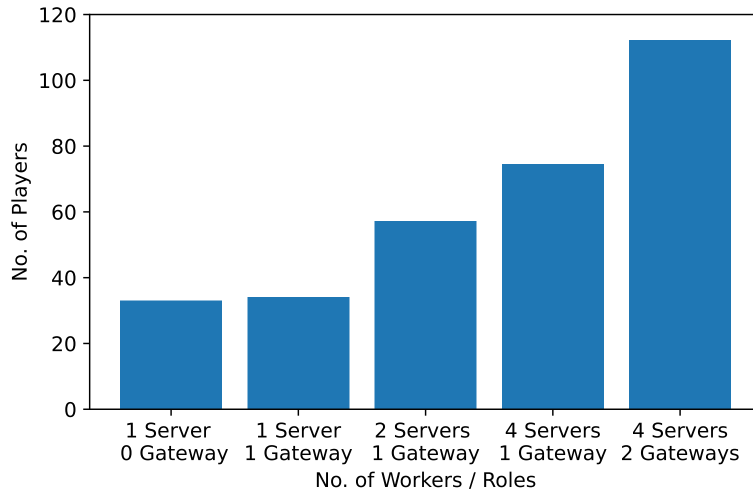

Figure 8.19: Maximum Number of Players with Increasing Number of Workers (Straight Walk Workload).

There are two types of workers, namely gateway and server instances. For one server with one gateway, the result is the same as one server without gateway (33). The reason is that the sole server is overloaded while handling the same number of players.

The maximum supported players are 57 for two servers with one gateway, which is an increase of 72%, and 75 for four servers with one gateway, which is an increase of 127%. Both results are lower than the sum of maximum supported players of independent server instances, with 13.6% lower on two servers and 43.1% lower on four servers. After we add one additional gateway to the four server configuration, the maximum supported players are increased to 117, which is an increase of 254% compared to one server with one gateway.

Figure 8.20 shows the normalized CPU usage of four servers with one gateway of one iteration, where each 25% represents a fully utilized core. The gateway is first overloaded with over 97% CPU utilization. This indicates that the gateway is the bottleneck. Also, we can see that different CPU usage on different server instances over time, with instance 1 utilizing the highest CPU resource. This indicates that the servers are not handling the same amount of work at the same time. The result is reasonable because our workloads only assign bots to explore the terrain. Different chunks on the terrain contain different amounts of data, depending on the chunk types. Instance 1 handles the chunks which consume most resources.

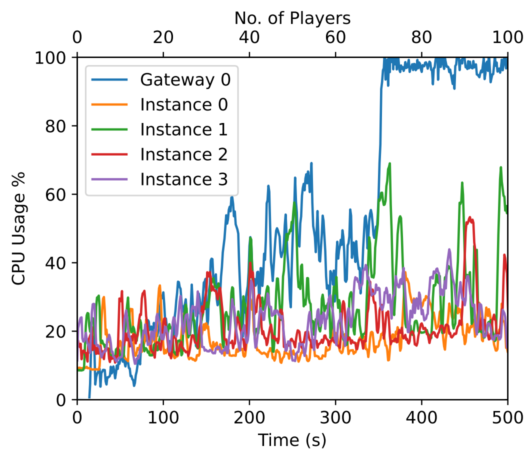

Figure 8.20: CPU Usage of 4 Servers with 1 Gateway.

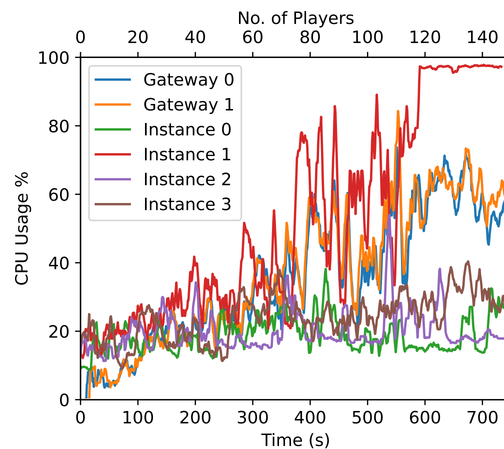

Figure 8.21: CPU Usage of 4 Servers with 2 Gateways.

Figure 8.21 shows the normalized CPU usage of one iteration with four servers with two gateways, where each 25% represents a fully utilized core. The gateway is no longer the bottleneck. Instead, instance 1 becomes the bottleneck while the other instances have relatively low CPU usage. This is the same situation as Figure 8.20 where the CPU utilization on instance 1 is the highest. Also, we can see that the CPU usage on gateway 0 and 1 are highly similar, which indicates that the work is evenly split.

We conclude that the layer seven gateway improves the scalability of MVEs. The gateway is highly scalable, with each one processing the same amount of workload. However, different server instances handle different amounts of work at a time, which indicates that we need a more efficient policy to assign players to different servers so that different game instances have similar amount of workloads at the same time, which further increases the scalability.

9

Conclusion and Future Work

9.1 Conclusion

Video gaming has been a mainstream and fastest-growing media over the last decade. Minecraft is one of the most popular games with its featured Modifiable Virtual Environment (MVE), which allows players to interact and modify the virtual worlds in real-time. However, despite its large number of players, previous research shows that it does not scale well. Serverless technology provides the advantage of automatic scaling, which has the potential to address the scalability challenges of MVEs.

In this thesis, we study how serverless technology benefits MVEs and how performance variability impacts the system. We design and implement a serverless MVE prototype based on Opencraft and Azure Cloud platform. We first migrate the file I/O services to remote cloud storage and apply latency hiding policies. Then we migrate some computation tasks to serverless functions. Finally, we implement a layer seven gateway to redirect players to different server instances and split the infinite terrain to different servers based on coordinates.

We design and implement a benchmark suite to evaluate the serverless MVE prototype. We conduct both microbenchmarking on services hosted on cloud platforms, with a focus on the performance variability of cloud services, and macrobenchmarking on the real system, focusing on four aspects: effectiveness of latency hiding policies, scalability, performance variability, and the worst cases perceived by players. We design two emulated player behaviors as workloads, namely Straight Walk, which stresses the server while exploring the infinite terrain, and Random, which mimics the random behaviors of real players.

The microbenchmarking result shows that the performance variability is high on all cloud services, including storage back-ends and serverless functions, caused by the underlying

scheduling policy of the platform and the network fluctuation to invoke them.

The macrobenchmarking result shows that serverless technology improves the scalability of MVEs. It benefits MVEs in terms of supporting more players by offloading heavy computation tasks to serverless functions for automatic scaling. The improvement varies per player behaviors. Additionally, the performance variability of the serverless MVE is mainly caused by the CPU resource of virtual machines and the internal process of MVEs. With proper latency hiding policies and asynchronous invocations, the high variability of serverless functions and blob storage does not negatively impact the serverless MVE. In the worst cases of specific events involved in the cold start of function instance where the players have to wait for the results, the player experience is not significantly impacted.

Furthermore, we show that redirecting players to multiple game instances which handle the same world through our implemented layer seven gateway further improves the scalability. Our first step of serverless MVE prototype shows that it has the potential for MVEs to support millions of players with serverless technology and effective latency hiding policies.

9.2 Future Work

In this thesis, we show that serverless technology improves the scalability of MVEs and does not negatively impact the performance and game experience. Here we present three future work directions.

First, our current implementation only migrates some parts of the computation tasks to serverless functions from the monolithic system. The computation tasks that remain on the monolithic still consume server resources. As shown in the evaluation results, the scalability is mainly limited by the CPU resource. In the future, we can offload more tasks to serverless functions, so that the CPU resource usage of the instance is lower. This further improves the scalability of MVEs.

Second, we implement the serverless technology on one MVE system Opencraft and we evaluate it on the Azure cloud platform. The vendor lock of serverless development limits the deployment to different platforms. In the future, we can implement it on different MVE systems and different cloud platforms, so that we can study and compare the performance variability.

Third, the main component of the MVE instance is the tick loop. The default scheduler that ticks the instance does not consider serverless technology and cloud storage. It assumes that the task duration is short and waits for the asynchronous result between schedulers.

This results in a long tick duration. In this work, we mitigate it by utilizing a new scheduling policy that postpones the task to future ticks when the results are available. In the future, we can implement a new scheduler that considers all the cloud components. The new scheduler also benefits the layer seven gateway for game content synchronization and player redirection.

References

[1] **'Digital media trends survey: Video gaming goes mainstream**. 1

[2] Mordor Intelligence. **'GAMING MARKET - GROWTH, TRENDS, COVID-19 IMPACT, AND FORECASTS (2021 - 2026)**. 1

[3] Brittany Vincent. **'Minecraft' Tops 131 Million Monthly Active Users**. 1

[4] Minecraft Wiki. **Tutorials - Redstone computers**. 1

[5] David French, Brett Stone, Tom Nysetvold, Ammon Hepworth, and W. Red. **Collaborative Design Principles From Minecraft With Applications to Multi-User CAD**. **16**, 08 2014. 1

[6] **Minecraft Education Edition**. 1

[7] Steve Nebel, Sascha Schneider, and Günter Daniel Rey. **Mining Learning and Crafting Scientific Experiments: A Literature Review on the Use of Minecraft in Education and Research**. *Journal of Educational Technology & Society*, **19**(2):355–366, 2016. 1

[8] Jerom van der Sar, Jesse Donkervliet, and Alexandru Iosup. **Yardstick: A Benchmark for Minecraft-like Services**. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19, page 243–253, New York, NY, USA, 2019. Association for Computing Machinery. 1, 4, 15, 21, 35, 38

[9] Minecraft Help. **Minecraft Realms Plus and Minecraft Realms: Java Edition FAQs**. 2

[10] **Can dedicated servers exceed 10 players? - Valheim**. 2

[11] Jesse Donkervliet, Animesh Trivedi, and Alexandru Iosup. **Towards Supporting Millions of Users in Modifiable Virtual Environments by Redesigning Minecraft-Like Games as Serverless Systems**. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association, July 2020. 2, 7, 15

[12] Erwin van Eyk, Alexandru Iosup, Simon Seif, and Markus Thömmes. **The SPEC Cloud Group's Research Vision on FaaS and Serverless Architectures**. In *Proceedings of the 2nd International Workshop on Serverless Computing*, WoSC '17, page 1–4, New York, NY, USA, 2017. Association for Computing Machinery. 2, 3, 10

[13] Mark Claypool and Kajal Claypool. **Latency and Player Actions in Online Games**. *Commun. ACM*, **49**(11):40–45, November 2006. 2, 8

[14] A. Iosup, N. Yigitbasi, and D. Epema. **On the Performance Variability of Production Cloud Services**. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 104–113, 2011. 2, 12

[15] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. **Is Big Data Performance Reproducible in Modern Cloud Networks?** In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 513–527, Santa Clara, CA, February 2020. USENIX Association. 2, 12

[16] Samuel Ginzburg and Michael J. Freedman. **Serverless Isn't Server-Less: Measuring and Exploiting Resource Variability on Cloud FaaS Platforms**. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, WoSC'20, page 43–48, New York, NY, USA, 2020. Association for Computing Machinery. 3, 13, 16

[17] M. Claypool and D. Finkel. **The effects of latency on player performance in cloud-based games**. In *2014 13th Annual Workshop on Network and Systems Support for Games*, pages 1–6, 2014. 8

[18] E. van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uță, and A. Iosup. **Serverless is More: From PaaS to Present Cloud Computing**. *IEEE Internet Computing*, **22**(5):8–17, 2018. 10

[19] E. van Eyk, J. Grohmann, S. Eismann, A. Bauer, L. Versluis, L. Toader, N. Schmitt, N. Herbst, C. L. Abad, and A. Iosup. **The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms**. *IEEE Internet Computing*, **23**(6):7–18, 2019. 10

[20] David Jackson and Gary Clynch. **An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions**. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 154–160, 2018. 11

[21] Canalys. **'Global cloud services market Q1 2021**. 11

[22] Trevor Alstad, Riley Dunkin, Rob Bartlett, Alex Needham, Gaétan Hains, and Youry Khmelevsky. **Minecraft computer game simulation and network performance analysis.** 11 2014. 15

[23] Raluca Diaconu, Joaquín Keller, and Mathieu Valero. **Manycraft: Scaling minecraft to millions**. In *2013 12th Annual Workshop on Network and Systems Support for Games (NetGames)*, pages 1–6, 2013. 15

[24] Alireza Goli, Omid Hajihassani, Hamzeh Khazaei, Omid Ardakanian, Moe Rashidi, and Tyler Dauphinee. **Migrating from Monolithic to Serverless: A FinTech Case Study**. 04 2020. 15, 27

[25] Jeongchul Kim and Kyungyong Lee. **FunctionBench: A Suite of Workloads for Serverless Cloud Function Service**. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504, 2019. 15

[26] István Pelle, János Czentye, János Dóka, and Balázs Sonkoly. **Towards Latency Sensitive Cloud Native Applications: A Performance Study on AWS**. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 272–280, 2019. 15

[27] Erwin van Eyk, Joel Scheuner, Simon Eismann, Cristina L. Abad, and Alexandru Iosup. **Beyond Microbenchmarks: The SPEC-RG Vision for a Comprehensive Serverless Benchmark**. In *Companion of the ACM/SPEC International Conference on Performance Engineering*, ICPE '20, page 26–31, New York, NY, USA, 2020. Association for Computing Machinery. 16, 35

[28] Pascal Maissen, Pascal Felber, Peter Kropf, and Valerio Schiavoni. **FaaSdom: A Benchmark Suite for Serverless Computing**. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, DEBS '20, page 73–84, New York, NY, USA, 2020. Association for Computing Machinery. 16

[29] **'ISO 35.100 OPEN SYSTEMS INTERCONNECTION (OSI)**. 18

[30] **Opencraft @Large Research**. 23

[31] Github. **Steveice10/MCProtocolLib**. 30, 39

[32] Jóakim v. Kistowski, Jeremy A. Arnold, Karl Huppler, Klaus-Dieter Lange, John L. Henning, and Paul Cao. **How to Build a Benchmark**. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, page 333–336, New York, NY, USA, 2015. Association for Computing Machinery. 35

[33] Huiguang Liang, Ian Tay, Ming Feng Neo, Wei Tsang Ooi, and Mehul Motani. **Avatar Mobility in Networked Virtual Environments: Measurements, Analysis, and Implications**, 2008. 41

[34] Wei Liu and Arika Ligmann-Zielinska. **A Pilot Study of Pokémon Go and Players' Physical Activity**. *Games for Health Journal*, **6**(6):343–350, 2017. PMID: 28853912. 41

[35] G. Turchini, S. Monnet, and O. Marin. **Scalability and availability for massively multiplayer online games**. In *EDCC 2015*, 2015. 47

[36] Trieu C. Chieu, Ajay Mohindra, and Alexei A. Karve. **Scalability and Performance of Web Applications in a Compute Cloud**. In *2011 IEEE 8th International Conference on e-Business Engineering*, pages 317–323, 2011. 47

[37] Microsoft Docs. **Latency in Blob Storage**. 56

[38] Microsoft Docs. **Azure Storage Blobs Pricing**. 57

[39] **How much of an issue are loading times for you?** 68

Appendix A

Technical Details

Technical Details of Benchmark Suite

We discuss benchmark suite in Chapter 6. We attach the technical details of the benchmark suite in Appendix. Table A.1 shows the details of configuration file used on our implemented benchmark suite. Table A.2 shows the exchanged messages between benchmark server and clients via TCP socket. The benchmark server controls clients and the clients notify the server through messages.

Source Code

The code of this project is open source and is available in the following Github repositories.

- Serverless MVE: https://github.com/atlarge-research/opencraft-dev/tree/feature/serverless-azure
- Layer Seven Gateway: https://github.com/JyQuery/opencraft-gateway
- Extended Yardstick: https://github.com/atlarge-research/yardstick/tree/feature/serverless-azure-exp
- Benchmark Suite: https://github.com/JyQuery/mve-bench

Table A.1: Details of Configuration File on Benchmark Suite.

Component	Parameter	Details
Benchmark	iteration	The number of benchmark iterations
Benchmark	mvepath	The relative path of MVE files
Benchmark	resultpath	The relative path of result files
Benchmark	exp. num.	The experiment number of the benchmark
Benchmark	players count	The number of emulated players
Benchmark	delay	Delay between the start of different components
Benchmark Server	IP	IP address of the benchmark server
Benchmark Server	port	Network port of benchmark server
Benchmark Server	timeout	TCP sockets connection timeout (default: 30)
MVE Server	IP, port	IP address, SSH port of MVE servers
MVE Server	game port	Game port of MVE servers (default: 25565)
MVE Server	login	System login and password of MVE servers
MVE Client	IP, port	IP address, SSH port of MVE clients
MVE Client	login	system login and password of MVE clients
MVE Client	mveserver	The MVE server that the client connects to
MVE Gateway	IP, port	IP address, SSH port of MVE Gateways
MVE Gateway	game port	Game port of MVE gateways
MVE Gateway	mveservers	The servers which the gateways concern
Functions	name	The function name as defined
Functions	applicationId	The application ID of the function
Functions	apiKey	The API key for accessing application insights

Table A.2: Details of Exchanged Messages through TCP Socket on Benchmark Suite.

Message	Parameter	Event	Details
Hello	mveclient/server	Connection established	Benchmark server identifies type of the other end of connection based on this message
Result	metric locations	Task completed	Send metric locations to benchmark server so that they can be retrieved
Keepalive	None	Benchmark executing	Keep TCP socket alive, useful for networks behind NAT
Bye	None	Benchmark completed	Notify the other end before closing the connection