Vrije Universiteit Amsterdam

Honours Programme, Report

A Case Study in Scaling Minecraft Using Serverless Computing

Author: Tiberiu Iancu (2659445)

1st supervisor: Alexandru Iosup
daily supervisor: Jesse Donkervliet

A report submitted in fulfillment of the requirements for the Honours Programme, which is an excellence annotation to the VU Bachelor of Science degree in Computer Science/Artificial Intelligence/Information Sciences
version 1.0

May 29, 2022

Abstract

With a player base that is continuously on the rise, Minecraft is one of the most popular games to date. The main feature of Minecraft is a fully Modifiable Virtual Environment (MVE). Similarly to Minecraft, many other games (forming the class of Minecraft-like games) have started featuring MVE and rising in popularity. Despite the high demand, MVE servers don't scale to more than a few hundred concurrent players due to poor parallelization, even under favorable conditions. Serverless computing is a novel type of cloud computing that promises high scalability, delivered on demand and with granular billing. To achieve this promise even for non-expert customers, serverless computing shifts operational logic onto the cloud provider. So far, serverless has shown promising results for a variety of applications, but it has yet to have been applied to the unique demand of online multiplayer gaming. Can serverless computing deliver on-demand scalability for MVEs? Addressing this research question, in this work we redesign a popular Minecraft-like MVE server as a serverless hybrid system, we implement the prototype starting from an open-source code base, and we propose and conduct a series of experiments to evaluate its scalability. Through real-world experiments, we find that serverless computing is suitable to scale individual components of the MVE server.

Contents

1 Introduction

In the past decade, the gaming industry has seen a massive increase in both number of players and revenue, making it the most prolific of the entertainment industries [14]. Minecraft is one of the most popular online games, having sold more than 200 million copies across all platforms, and boasting well over 130 million active players each month [5]. Its main selling point is the fully modifiable virtual environment (MVE), which allows players to build, craft, and explore the sandbox world. This allows Minecraft to be used in many educational projects and even in social activism [3]. However, past research suggests that Minecraft does not scale well: a relatively low number of users (in the low hundreds) can play together in the same world [16].

In this research, we tackle the scalability problem of MVEs using serverless computing: a novel type of cloud computing that allows the user to run code on-demand, without having to address the underlying operational logic. We aim to answer the question "Can serverless computing be used to scale MVEs?", by redesigning the system as a serverless hybrid, implementing a prototype and performing a set of experiments to evaluate the performance of such a system. The goal of this research aligns with the massiving computer systems (MCS) vision [12] and is a step in the direction of redesigning Minecraft-like games as serverless systems [7].

1.1 Problem statement

The complex nature of MVEs, induced by fully modifiable terrain, makes the implementation of a highly parallelizeable server difficult to achieve. Past research [16] has shown that none of the existing server implementations can take full advantage of a multi-core CPU, but rather rely on a few cores to do all the work. This is reflected in the poor scalability of the server, as even datacenter-grade machines can only handle a few hundred players. By scalability we understand the ability of the system to cope with increased demand, without showing signs of higher response times [11]. In this work we focus on on-demand scalability, as opposed to offline scaling.

Since the MVE server is complex and made up of multiple components (discussed more in depth in Chapter 2), in this work we choose to focus on one of them, namely the content generation component. The MVE world is a virtually infinite sandbox that gets generated by an algorithm, usually on the server-side. The server generates the terrain type and shape, as well as structures, buildings, villages, entities, etc. This can be computationally expensive and current MVE server implementations cannot cope well with demand.

In an effort to apply novel techniques to the scalability problem of Minecraft, dynamic consistency units (dyconits) proved to significantly improve the scalability of the server [6]. However, dyconits are a hyperspecialized protocol, and applying it to new applications requires much expertise. Can we obtain the scalability benefits of dyconits (or better) without the specialized approach? Serverless computing has shown promising results in the past, in tasks such as video encoding [9], or graph processing [15], but to our knowledge, it has yet to have been applied to the domain of online multiplayer gaming. Therefore, we have no guidelines on how to best use serverless computing in the latency-critical environment of MVEs. However, serverless' main promise is elasticity, so in theory it is a good fit for the scalability problem of Minecraft.

1.2 Research questions and approach

The main question this research is trying to answer is "How can serverless computing be applied to on-demand scaling of the content generation in Minecraft-like games?" To answer this, we ask the following **R**esearch **Q**uestions:

RQ1 *How to redesign a real-time, monolithic MVE system as a hybrid serverless system?* As there is currently no reference design for an MVE serverless system, we apply the AtLarge system design methodology [13] to find a suitable design.

RQ2 *How to implement/realize such a system in practice?* We answer this question by implementing a prototype for the designed system and documenting the main challenges. There are currently no standard methodologies for translating a monolithic systems serverless.

RQ3 *How to evaluate the performance of such a system?* Having the redesigned system on hand, we ask ourselves how we can compare its performance to the monolithic server. We therefore design and perform a series of experiments that can evaluate the scalability and performance of the redesigned system against the monolithic one. There is no standard method to evaluate the scalability of the Minecraft server, or of one of its components. Furthermore, there is no publicly available player behaviour data that could help us devise realistic workloads.

In essence, we ask ourselves how can we design, implement, and evaluate the serverless hybrid system.

1.3 Main Contribution

By answering the Research Questions proposed in the previous section, we contribute with a design of a serverless content generation system for massive virtual worlds, we provide guidelines and document the main challenges involved with the implementation of such a system, and finally, we propose and run a set of real-world experiments to test the performance of the MVE server.

1.4 Report Structure

In this section we summarize the structure of this report. In Chapter 2 we introduce the key concepts and terminology necessary for understanding the design in Chapter 3. In Chapter 3 we define the system design requirements and present an overview of the design; we then explore each component individually. In Chapter 4 we define the requirements, discuss implementation details, and present the main challenges associated with implementing an MVE serverless hybrid. In Chapter 5 we propose and conduct real-world experiments to evaluate the implemented system and from these we derive the main findings of this work. We follow in Chapter 6 with a discussion of the experiment results and limitations of the system. Lastly, we summarize the main contributions and possible directions of future research in Chapter 7.

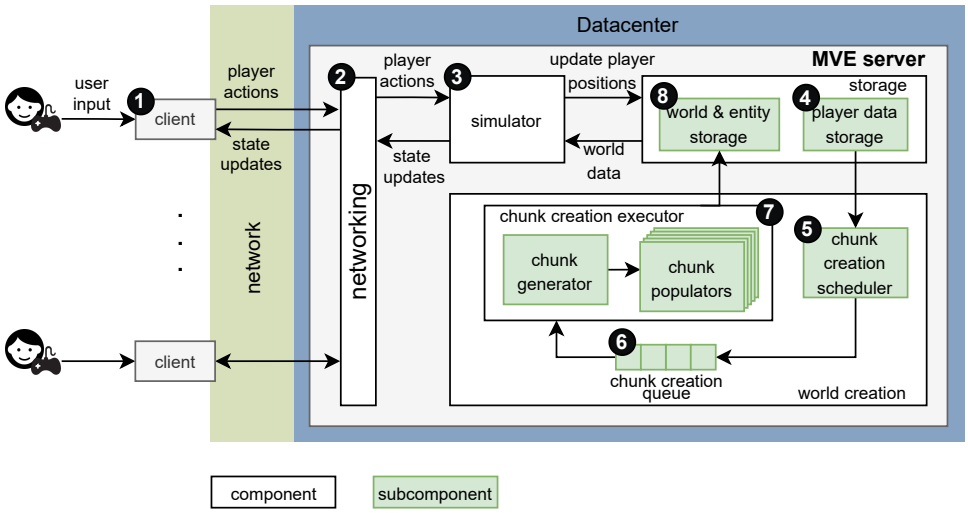


Figure 1: MVE server high level overview.

2 Background

This chapter presents the key concepts involved in this research. Section 2.1 discusses the MVE server and the terminology necessary to understand world creation in Minecraft. Then in Section 2.2 we explain how serverless computing works and its main advantages and disadvantages.

2.1 Modifiable Virtual Environments

A game is considered to have a **M**odifiable **V**irtual **E**nvironment (MVE) if the player can explore a virtually infinite, fully modifiable world. Minecraft is the most popular MVE game. Figure 1 shows an overview of the MVE server. The main components we have identified are: world creation, (player and world) storage and simulation (non-playable characters i.e. NPCs, weather, redstone, etc.). In this work we focus on world creation.

To explain how world creation works in Minecraft, it is necessary to define the main concepts related to the Minecraft world. We explain here common terminology used by the communities of players and developers formed around Minecraft and Minecraft-like clone-projects.

- **Blocks** are the smallest unit in the Minecraft world. They represent 1x1x1 cubes that get stacked together in order to form bigger structures, such as mountains, villages, lakes, forests etc.
- **Chunks** are block regions that span the entire height of the map.
- **Creation** is the process of generating, then populating terrain. This is an important concept for this thesis, because, as described in 1.1, it is central to the process of content generation we aim to scale in this work.

  Generation is the process of spawning terrain blocks, such as dirt, rock, water, etc. Generation gives the general shape of the terrain.

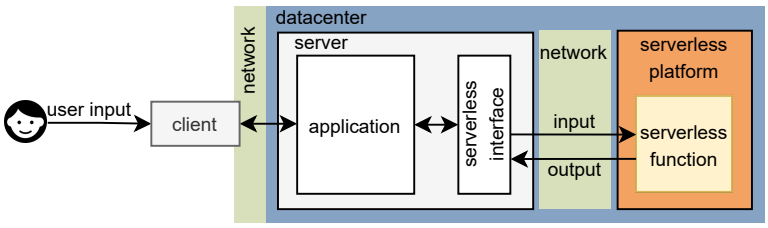


Figure 2: High level FaaS model. The application makes a request to a serverless platform through an interface, providing some input, then receives the output as response.

> **Population** is the process of decorating the generated terrain with structures such as trees, caves or villages and spawning entities e.g., ducks, sheep etc.

The Minecraft world is created procedurally [10], using a pseudo-random number generator. Starting from an integer called **random seed**, the creation algorithm will always produce the same world data. In the Opencraft server, generation and population are performed on a per-chunk basis and each chunk can be created given only the seed and its coordinates (on the x and z axes). Different servers implement different algorithms for creating terrain, but the functioning principle is the same: given the same random seed, the world data will be consistent in-between runs.

Since the world is too large to store in memory, the world data is generated in discrete steps, usually on a per-chunk basis, based on user positions: the client (❶) translates player input into in-game actions and sends them to the server, where the networking layer (❷) relays them to the simulator (❸). The player locations get updated and written to the player storage (❹), which is then used in the process of world creation. Given the player positions, the chunk creation scheduler (❺) decides which chunks have to be created and in which order, then queues them for creation (❻). The queued chunks first get generated, then populated by the chunk creation executor (❼) and finally added to the world storage (❽). From here, the chunk data can be accessed by the simulator. If necessary, the simulator sends state updates back to the client.

2.2 Serverless computing

When building an application that relies on cloud services, it is currently very challenging to predict exactly the load that the service will experience. Especially when the workload depends largely on interactive human input, as is the case for online MVEs in general and for Minecraft-like games in particular, load prediction remains a largely unsolved problem. Guaranteeing good quality of service can currently be achieved through an approach that prioritizes reserving plenty of machines, well beyond the actual need most of the time, which can be costly and inefficient. In contrast, optimizing for cost might mean that the system cannot support all its users. Serverless computing solves this problem by shifting the operational logic from the customer to the cloud provider. In this model, the customer is allocated resources dynamically, as demand increases.

In this work, we focus on Function-as-a-service (FaaS), a type of serverless computing that allows developers to provide code in the form of functions and invoke it remotely from an application (such as a server) to perform cloud computation. Most cloud service

providers offer FaaS platforms (Google Cloud, Microsoft Azure, IBM cloud etc.). Currently, AWS Lambda is by far the most popular [8]. AWS Lambda bills users per 1ms of execution time **times** the amount of memory used, i.e., per gigabyte second. For example, a function that runs for 499.3 ms with 512 MB of memory reserved is billed for $0.5\,s \times 0.5\,GB = 0.25\,GBs$. There is also a fixed fee of \$0.20 per 1 million requests [1].

Internally, serverless platforms run functions inside containers [citation]. When a function is idle for a longer period of time it becomes **cold**, i.e. the container gets gets deinitialized to free up resources. Upon starting to make requests again, another container has to be initialized (or **warmed**), which results in an observable delay. For example, on AWS Lambda we noticed approximately 30 second of latency when using the Java runtime, which is significant for latency-critical applications.

Serverless platforms require the user to upload a **handler**, a function responsible for receiving requests and returning output in the correct format. Internally, the handler can depend on other functions or libraries, which the user also has to provide.

3 System Design

In this Chapter we address the first Research Question: *How to redesign a real-time, monolithic MVE system as a hybrid serverless system?* First, we define the design requirements in Section 3.1, then we present the main components of the system and how they relate to the requirements in Section 3.2. We then walk through the population of a single chunk.

3.1 Requirements

The system design must conform to the following **D**esign **R**equirements:

DR1 *The system should increase the scalability of the chunk creation component, by providing higher content-creation throughput than the monolithic system.* This is the core requirement of the system. Serverless computing grants us increased scalability, so improved MVE system should be designed to efficiently make use of such an addition.

DR2 *The system should operate under typical QoS requirements for MVEs.* Achieving higher content-creation throughput does not necessarily imply reduced latency for the game, end-to-end. The round-trip network time to and from the serverless platform, combined with high-latency cold starts, can result in perceivably lower QoS on the client side. In our design we must account for these drawbacks of serverless computing. Furthermore, we have to balance the increased system complexity induced by the use of serverless, to not bottleneck other components.

DR3 *The system should allow its users to trade-off performance against cost.* Although serverless grants increased scalability, the use of serverless platforms comes at a cost. When the server load is low, we might wish to disable serverless chunk creation or use lower memory functions to save on cost. When the load is high, the system should have mechanisms to increase creation throughput to keep up with the demand.

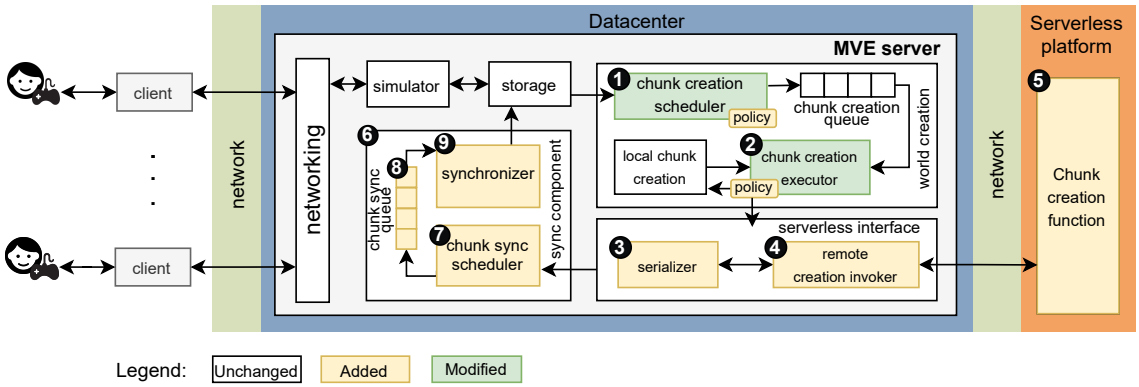


Figure 3: Serverless chunk creation system design.

3.2 Design Overview

In this section we explain how the system design meets the design requirements and discuss the design choices in the serverless system .

Figure 2 illustrates the high level design of the system. We keep the MVE client-server design of the presented in Chapter 2, but make modification to accommodate for the usage of a serverless platform. First, we modify the chunk creation scheduler (❶) and executor (❷) to allow for policy driven decisions. We do this to enable the server to schedule chunks differently for local and serverless creation (as to better take advantage of the serverless elasticity), as well as allow the executor to create chunks locally or serverlessly, according to cost and performance requirements. Second, to enable communication with a serverless platform, we add an interface that consists of two components: a serializer (❸) and a function invoker (❹) that is responsible for making requests to the serverless chunk creation function (❺). Finally, we add a synchronization component (❻) that takes the output of the serverless function, de-serializes it, and writes it to storage.

We expect to satisfy **DR1** by offloading local computation onto the serverless platform, where we can achieve higher levels of parallelization. A detailed look into the scalability comparison between our serverless hybrid and the monolithic system is presented in Chapter 5. However, due to the relatively computationally expensive nature of the creation algorithm, we can expect a performance increase, provided that the overhead of the interface doesn't have a noticeable performance impact. Chapter 4 discusses the challenges associated with creating the AWS interface.

When faced with **DR2** (concerning MVE QoS requirements) and **DR3** (concerning cost-performance trade-off), we decided to keep the world creation component from the reference MVE design (Figure 1), but modify the scheduler and executor as to allow policy driven decisions. This has three main benefits. First, we allow the executor to create chunks locally, according to cost and server load requirements (**DR3**). Second, the scheduler can queue chunks in advance according to serverless platform latency and/or server load (**DR2**). And third, the executor can choose to create chunks on a serverless function configured with a lower or a higher amount of memory to optimize for cost or performance respectively (**DR3**). A visual example of how the scheduler and executor work is presented in Figure 4.

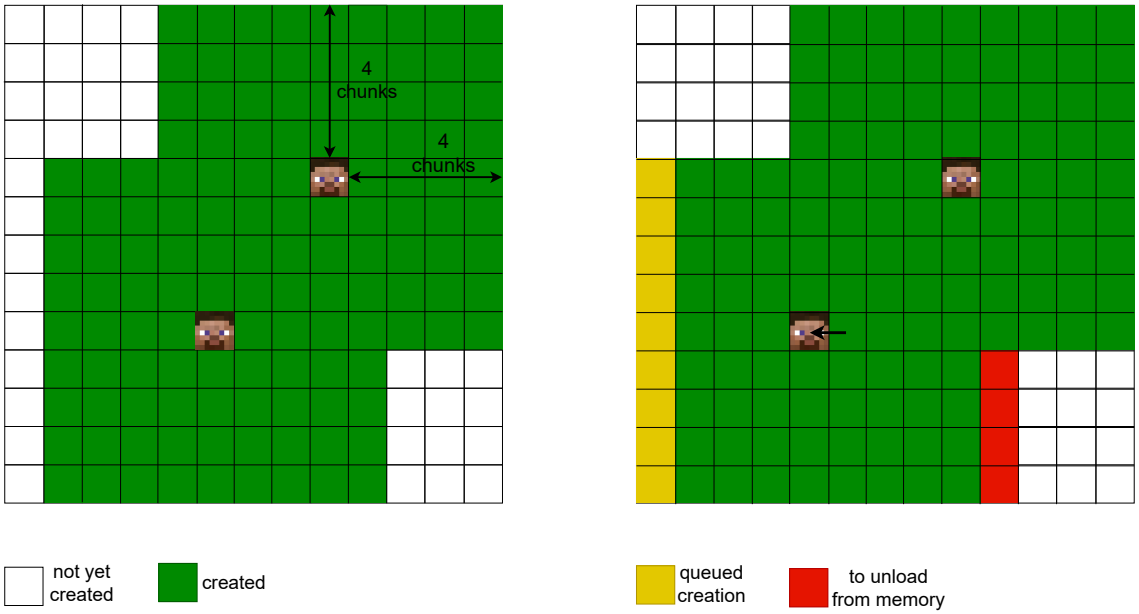


Figure 4: Example of chunk scheduling decision with player field of view of 4 chunks. When one of the players moves, the scheduler queues the light green chunks for creation, while it marks the red chunks (that are now outside of the area of interest of any player) to be unloaded from memory.

To improve user-perceived latency (**DR3**), we introduce a synchronizing component (❻). This component is similar in structure to the world creation component: it has a scheduler (❼), a priority queue (❽), and a synchronizer (❾) (similar to the chunk creation executor). After serverless creation finishes, the resulting serialized chunk data is passed to the synchronizing scheduler, that adds it to the synchronizing queue. The closer this chunk is to a player, the higher priority in the queue it has. This ensures that players first receive chunks that are most relevant. The synchronizer takes chunks from the synchronizing queue, one by one, de-serializes the data, writes it to storage, and sends it to the player that originally triggered the creation of the chunk (only if the chunk is still in the field of view of the player).

Initially, we opted for a design without a synchronizing scheduler (❼) or synchronizing queue (❽). In this version, the synchronizer would work on a 'first come, first served' basis; this had two main drawbacks. First, player positions can change during serverless creation; as such, the chunk that is being created can go out of the field of view of the player. When serverless creation finishes, the synchronizer sends the player data for a chunk they are no longer interested in. Second, although the chunk creation queue ensures that the chunks closest players have priority to be created first, we have no guarantee that serverless creation finishes in the order that we made the requests. Furthermore, player positions also update during serverless creation. As such, most of the time, the synchronizer would send the players data for chunks that are further away. Figure 5 illustrates an example where only the synchronizer is used (5b), and where the whole synchronizing component is used (5c). As noted above, due to the unpredictable nature of serverless creation latency

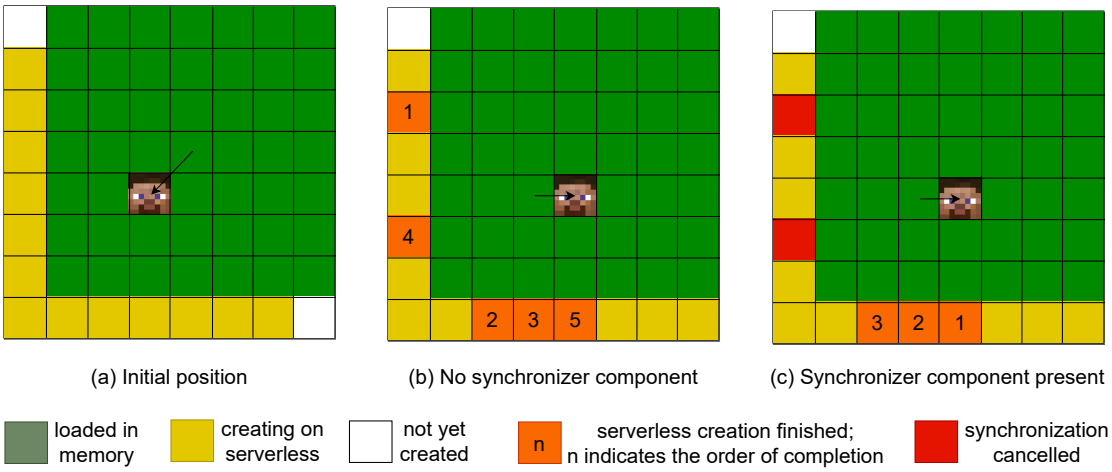


Figure 5: Illustration of the role of the synchronizer. In this example the player has a view distance of 3 chunks. The player moves to the initial position (a), triggering the serverless creation of the chunks marked with yellow. In (b), the player moves one chunk to the right and the serverless creation of the chunks marked with orange is complete. The server will send the player the chunk data in the order that it arrived in. The chunks marked 1 and 4 should not be sent to the player, since they are already outside of the field of view. Furthermore, the player should receive the chunks closest to them first (i.e., in the order 5, 3, 2). (c) displays the same situation when we use the synchronizer: we cancel the chunks too far away from the player, then queue the ones closest to the player to be synchronized first.

and player position updates, we see that when only the synchronizer is used, the player is being sent chunks in an unfavourable order.

In this paragraph we present a walk through the serverless creation of one chunk. The main execution steps in the system are similar to the monolithic server: player actions are relayed to the simulator, that updates the storage accordingly. Similarly, the chunk creation scheduler (❶) queues the chunks according to the selected policy, and the executor (❷) communicates the proper input needed to generate the chunks to the serverless invoker (❹), or performs the computation locally, should cost need to be reduced. Since serverless functions expect input in a specific format (e.g., JSON), we pass the input through a serializer (❸) and make a request to the serverless creation function (❺). The chunk data is then communicated back to the synchronizer (❻), where the scheduler adds it to the priority queue. Once first in the queue, the synchronizer initializes the chunk in memory; from here, the chunk data can be sent back to the players.

4 Implementation

This chapter discusses the process of implementing the designed hybrid system and presents the main challenges associated with it. The code for the prototype is released publicly, through the Opencraft Github repository [2], a Java-based monolithic Minecraft server forked from the community maintained Glowstone server. This way, we make sure our

project contributes to open science and also adheres to the Findable, Accessible, Interoperable, and Reusable (FAIR) principles [17].

In theory, a chunk can be created from only the world seed and its x and z game-world coordinates, so given the stateless nature of serverless functions this looks superficially like an easy task. In reality, the Java code responsible for chunk creation depends on many stateful objects that need to be initialized before the generation and population processes start. Therefore, the main technical challenge in realizing the serverless chunk creation system is synchronizing the relevant game state information with the serverless function.

4.1 Implementation Requirements

We define the following **I**mplementation **R**equirements:

IR1 *The implementation of serverless content generation should increase the generation throughput, by making use of the elasticity provided by serverless platforms.* Analogous to **D**esign **R**equirement 1: the additional cost induced by using a serverless platform should also provide a performance improvement.

IR2 *The network latency to and from the serverless platform, as well as the overhead of the serverless interface should not decrease the QoS of the MVE server.* Analogous to **DR2**: The system should be able to cope with the increased latency of content generation. Furthermore, the interface with the serverless platform should not come at a significant performance penalty.

IR3 *The system should allow for seamless switching between local and serverless content generation, and should provide an easy interface to extend scheduling policies in the future.* Analogous to **DR3**: we want to allow the user to trade-off performance and cost i.e., to balance between local and serverless generation, depending on demand. Furthermore, we want to allow smarter scheduling policies to be easily implemented into the system in the future.

4.2 Serverless Function Implementation

In this work we implement the serverless system using AWS Lambda. The interface we build with AWS Lambda consists of two parts: a function invoker, implemented with the AWS Java SDK, and a serializer, discussed in more depth below (see Figure 3, components ❸ and ❹).

We upload the server code to the AWS Lambda platform and write a simple handler that internally calls the already existing chunk creation function in Opencraft. Here we face the technical challenge of initializing all the necessary objects required by the creation function in the server. The next section discusses how we overcome this. To ensure full functionality we have to take additional steps, as explained below.

Pulse tasks: During the population process, certain structures can be created (such as rivers), and their respective *pulse tasks* are started: each game tick the structures have to be updated (for example, to flow in the right direction). This presents a technical challenge that we solve by collecting pulse task information during serverless chunk creation, then serializing and sending it back to the server where the pulse tasks can be started.

Modifying adjacent chunks: In the Opencraft server, population is a process that modifies each block individually, i.e., once data for a certain block is calculated, it is immediately written to memory, as opposed buffering it and writing the whole chunk data at the end of the creation process. When creating serverlessly, we do not have access to the server memory, so each one of the block modifications is written in the storage on the serverless platform. At the end of serverless creation, we extract the data for the whole chunk and send it to the server, where it can be synchronized. However, in the population process structures often span multiple chunks, for example a tree's leaves can be spawned in adjacent chunks. If we only send the data for the chunk that we are populating, we lose the information about the surrounding blocks (e.g., the tree leaves). To solve this during serverless creation, we collect all block changes and send these as **B**lock **C**hange **M**essages alongside the chunk data.

4.3 Serverless Hybrid Implementation Challenges

AWS Lambda uses serialization for input and output by default. For interfacing with AWS, we use Gson, a JSON library made by Google, due to its ability to auto-serialize most Java fields and to create custom rules for serializing. For simple data structures, Gson can serialize the objects without any additional input from the programmer. In the case of Opencraft, the game state that Lambda needs for chunk creation cannot be automatically serialized by Gson. We faced three main challenges, discussed in detail below.

The first challenge is determining the data dependencies of the serverless function when creating a chunk. Knowing the data dependencies is important, as we need to know which game state objects must be included in the input for the serverless function. Due to the monolithic nature of Opencraft, isolating the creation function and its dependencies is difficult. We do not want to send all data the game holds in memory, rather only what is strictly necessary, to reduce network usage, and save (de)serialization time. We solve this problem through a combination of manually analyzing the code, and trial and error (i.e., excluding certain fields and seeing if chunk creation works without it). Once we know that a field of a certain class is unused during the chunk creation process, we use Java annotations and Gson exclusion strategies to isolate it (e.g., by marking a field with '@ExcludeField' we can tell Gson to exclude it from the serialized JSON string). The data we send to the serverless function consists of information about the different initialized populators and generators in the game server, together with the chunk coordinates and the world seed.

The second challenge is posed by circular dependencies. Java classes can hold references to each other: class A has a field that references class B and class B has a field that references class A. The same approach as above can be employed to solve the problem of circular dependencies: we use Java annotations and Gson exclusion strategies to mark the fields that create the circular dependencies.

The third challenge is serializing not-trivial Java objects: objects that have fields denoting functions or objects included from other Java libraries (the latter cannot be serialized using exclusion strategies, as we cannot modify the code of the third-party libraries). We solve this challenge by writing custom serializers for these types of objects and registering them with Gson. For example, in Opencraft, certain objects use optimized list imple-

mentations from third-party libraries. In this case, we write a serializer that queries the contents of the respective lists and produces a Base64 encoded string. We also write a de-serializer that can instantiate such list objects given the Base64 encoded representation of the data.

4.4 Policy implementation

Having the serverless function and AWS Lambda interface implemented, we make modification to the server to reflect the system design.

First, we modify the chunk creation scheduler (❶) and executor (❷) to allow for different policies to be attached. By default in Opencraft, the scheduler runs on a single thread and is called to refresh the queue once per tick. It prioritizes chunks that are closest to a player to be created first. The executor spawns a thread pool with a number of threads equal to the number of virtual cores of the system, where each thread works on creating queued chunks independently of server ticks. We will refer to this policy as 'local creation'.

Second, we implement a policy for serverless creation, similar to the local policy. The scheduler employs the same policy as described above, but we make modifications to the worker threads of the executor as to use the serverless interface. After serverless creation is finished, we pass the resulted chunk data to the synchronizing component. This component has a similar structure to world creation: a single-threaded scheduler that refreshes the queue with chunks that have finished serverless creation and a synchronizer that runs on a single thread (to avoid memory inconsistencies in a multi-threaded environment).

4.5 Game Performance Improvements

In this section we discuss optimizations we have made to our prototype, as well as inefficiencies we have fixed in the Opencraft codebase.

Due to an inefficiency in the population code, Opencraft initially sent duplicate data to players. More precisely, while populating a chunk the server sends a message (the standard **B**lock**C**hange**M**essage) to all players in the range of a chunk as soon as the data for a single block has changed. After the population finishes, the server also broadcasts the data of the whole chunk. This is highly inefficient, as during the creation process players receive a number of messages in the order of tens of thousands, namely one for each populated block in the chunk. To fix this, we collect BCMs and only broadcast those messages that are not part of the chunk currently being created. This optimization also applies to the serverless function.

Since chunk data makes up the majority of the output from the serverless function, we use a custom chunk (de)serializer (as opposed to letting Gson auto-serialize to JSON) to speed up the deserialization process on the server. Opencraft already has a mechanism in place to write chunk data to files in byte format, which we can reuse to convert chunks to Base64 and include them in the final JSON string, thereby improving the deserialization process latency by a factor of 10.

Most of the information that the serverless function requires to create chunks has to be re-serialized and re-sent each time. In fact, other than the chunk coordinates and the world seed, all other input data can be cached and reused, instead of serializing it on each request. This is a small optimization that drastically improved the input serialization time (by a factor of 10).

Section	Focus	Lambda Memory [MB]	Duration [min]	Acceleration $[blocks/s^2]$
5.2	Loaded world overhead	512	20	0
5.3	Cost-performance trade-off	320-10,240	5	0
5.4	Performance variability	512	5	0
5.5	Stress test	512	20	$\frac{1}{200}$*

Table 1: Experiment overview. (*The acceleration is not constant, rather the bot speed increases every 200 s.)

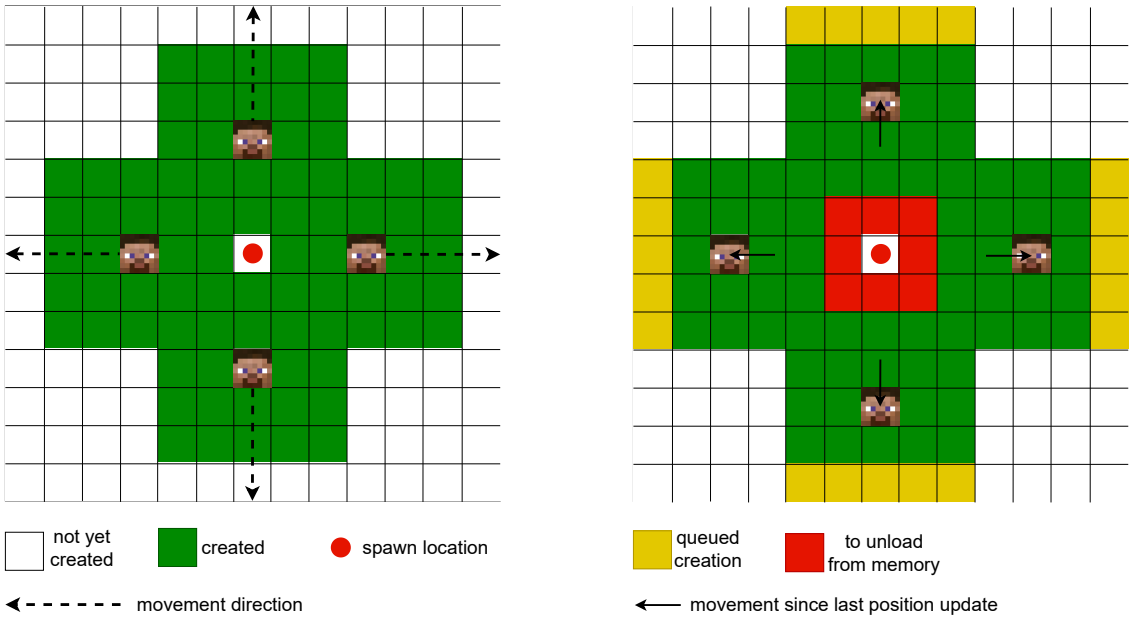


Figure 6: Bot movement model visualized. In this example there are 4 players, each with a field of view of 2 chunks.

5 Real-World Experiments

In this chapter we propose a set of experiments to evaluate the performance of the serverless and monolithic systems. From the results we derive the following **M**ain **F**indings:

MF1 The AWS Lambda serverless function memory configuration can be used to trade-off cost against performance.

MF2 Using serverless computing for content generation introduces high performance variability.

MF3 Serverless computing improves the scalability of world generation in the MVE server.

Component	Value
CPU model	2 × Intel Haswell E5-2630-v3
CPU frequency	2.4 GHz
CPU cores	2 × 8 cores, 2 × 16 threads
Memory	64 GiB

Table 2: DAS-5 node specification. Each machine is equipped with 2 CPUs.

5.1 Experiment setup

This section presents an overview of the experiments. The experiments in this paper have been conducted using Yardstick, a benchmark for the Minecraft server [16]. Yardstick allows us to connect bots to the server that model player behaviour.

For the experiments in this research we spawn the bots in the same location and make them fly away from each other, as shown in Figure 6. We can thus tweak the amount of connected bots, their speed and their field of view to adjust the number of chunk creation requests. For all experiments we connect 5 bots that start with a speed of 1 block/s and have a field of view of 8 chunks. We found that these values are high enough for both the serverless and local creation systems to be able to cope with chunk creation demand. The individual experiment setup is presented in the sections below and summarized in Table 1.

We run the experiments on the DAS-5 cluster [4]. The specification of the compute nodes is summarized in Table 2. To avoid a server bottleneck, we run the Opencraft server and Yardstick instances on separate machines. This yields a setup similar to that of MVE servers running in datacenters, due to the high compute capability and high memory configuration.

To get a fair comparison, the world seed does not change across configurations and experiments. The baseline seed we used is -5492030302352580259.

5.2 Loaded world overhead

The first experiment tests the performance of the server with certain simulation features disabled to observe possible bottlenecks for the world creation component in the system.

The Minecraft server runs at a rate of 20 ticks (updates) per second. If ticks last more than 50ms for long periods of time, the server becomes overloaded and the QoS cannot be maintained. Upon inspecting different simulation sub-components in Opencraft, we noticed two that cause a significant increase in tick duration: chunk unloading and randomized block pulsing.

Chunk unloading is a periodic event triggered by default every 5 minutes that causes chunks that are not in the field of view of any player to be unloaded from memory and written to disk. This is an optimization to save memory that ends up causing lag spikes in the server as thousands of chunks are serialized and written to disk. An efficient implementation of chunk unloading would run on a separate thread and unload chunks as they become irrelevant to players.

Randomized block pulsing triggers block updates every tick (such as water turning into ice, or plants growing) for three random blocks in each chunk that is loaded in memory.

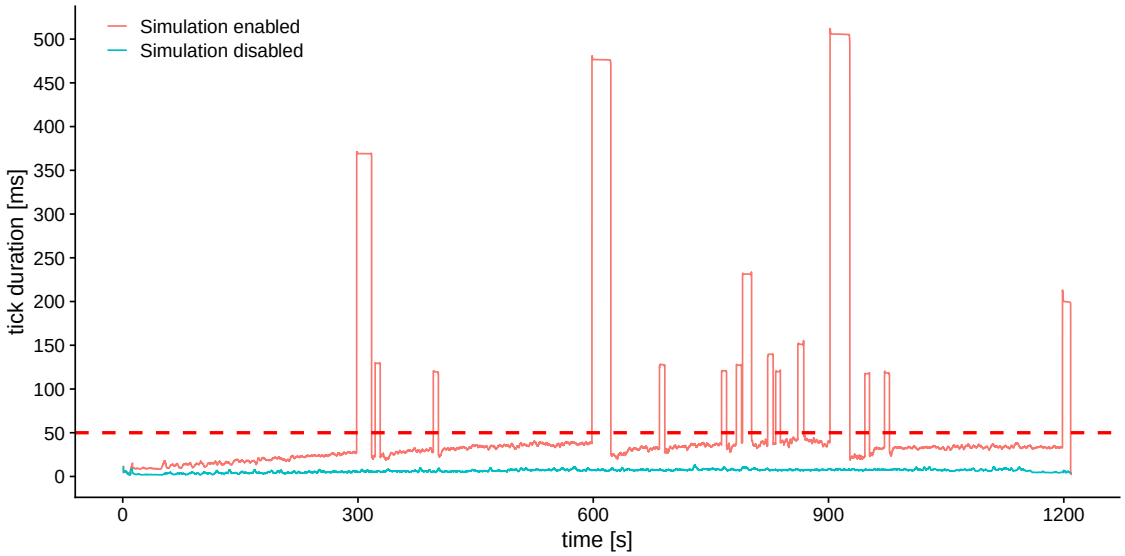


Figure 7: Comparison of tick duration with chunk unloading and block pulsing enabled/disabled.

This causes an increased tick duration over time as players discover more terrain, causing the server to eventually get overloaded. The tick time increase attributed to block pulsing is a byproduct of the inefficient chunk unloading.

Both configurations run local creation. We plot the results of this experiment in Figure 7. First, we see the lag spikes occurring every 300 seconds due to periodic chunk unloading. The graph shows a rolling average of the tick values; in reality the tick duration jumps to over 10 seconds, before returning to normal. We also notice the increasing tick time attributed to the block pulsing, as well as occasional smaller spikes. We do not have direct evidence, but we conjecture that the smaller spikes in tick times come from complexity differences in updating certain block types. Upon disabling these two simulation features we see a constant tick duration over time, as well as no sudden increase in tick duration.

To not bottleneck the world creation component and to get a fair scalability comparison between the serverless and monolithic systems, we disable chunk unloading and randomized block pulsing for the next experiments. Although the chunk unloading is present to prevent the server from filling the whole memory with chunk data, we have found that during this experiment a maximum amount of 4GiB of memory was used out of the 64GiB that the system is configured with, therefore we have enough memory to hold all of the chunks loaded during our experiments.

5.3 AWS Lambda Cost Performance Trade-off (leads to MF1)

A Lambda function set up to have more memory is also given more computational power. In this experiment we test the serverless function performance and cost-performance trade-off with different memory configurations.

We found that the minimum amount of memory our function needs is around 320MB, and the maximum amount allowed by Lambda is 10,240MB. We therefore run the same

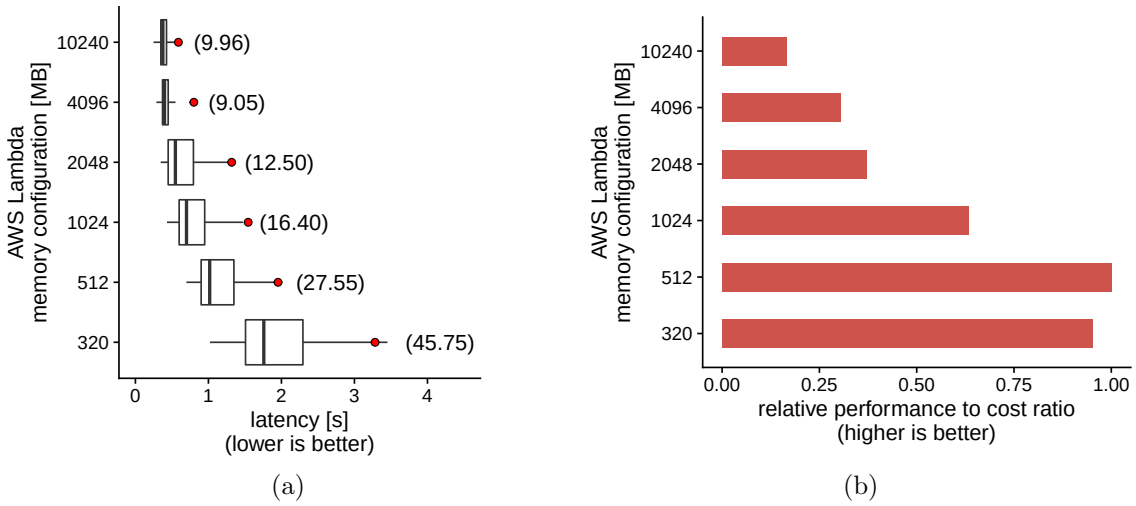


Figure 8: Serverless creation latency by amount of configured memory. Outliers not shown for better readability. The mean is marked with the red dots and the max is written in parentheses.

Lambda function configured with 320, 512, 1024, 2048, 4096, and 10,240 megabytes of memory on the same workload as the stress test (unchanged parameters) and we compare the execution latency. The way AWS Lambda gives functions with less memory less computational power is by a time-sharing system: a function with 1024MB of memory will get 1/10 the CPU time of a function with 10240MB. Therefore, we would expect execution times to be directly proportional to the amount of configured memory when not taking network time into account. When accounting for network latency, we expect diminishing marginal returns on overall performance.

Figure 8 shows the performance of the serverless chunk creation function when configured with different amounts of memory. We measure the latency as the time between sending the request to AWS and receiving the chunk data. In Figure 8a we plot the mean latency of serverless execution, including the network time. We observe significant improvements in performance with higher memory configurations: from just over 3 seconds for the 320MB function to under 1 second for the highest memory configuration. The mean is far to the right of the median, suggesting a long-tailed distribution. Indeed, when looking at the maximum value, we see extreme outliers, present due to serverless function cold starts.

There are, however, also cost considerations with serverless platforms. AWS Lambda billing is proportional to the amount of configured memory for the function. 8b shows the relative cost efficiency (value) by configuration. We calculate the value by multiplying the mean execution time by the amount of memory, then normalize by dividing the smallest value by all others. As expected, when giving Lambda more memory, the cost efficiency of using higher memory configurations is lower, but interestingly 512MB of memory yields better value and significantly better performance than the minimum amount (320MB). For these reasons, we run all other experiments with 512MB of memory.

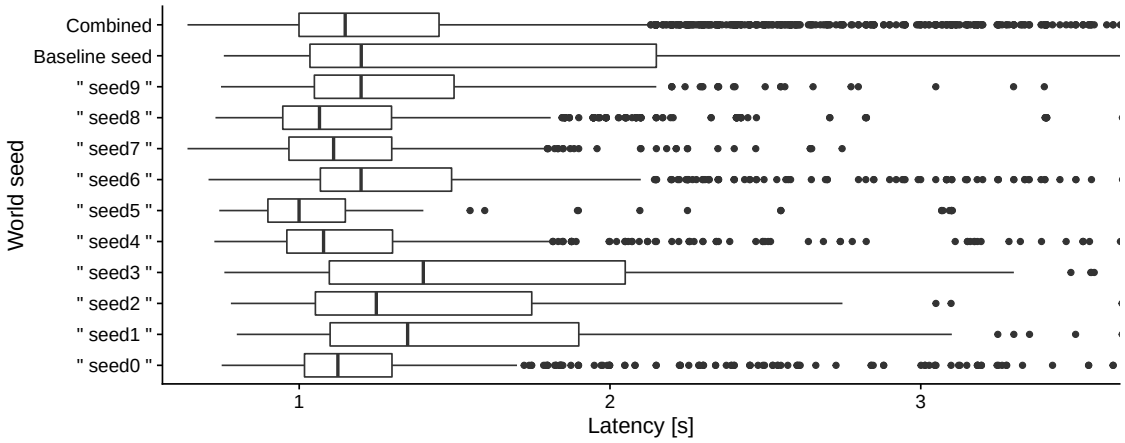


Figure 9: Serverless creation latency by world seed.

5.4 Performance Variability (leads to MF2)

We would expect that over very large areas of terrain the chunk creation latency would converge to the same value regardless of world seed. In practice, we cannot run experiments over the whole Minecraft map, rather over only a small portion that will mainly represent a single type of terrain. In this experiment we aim to test how much of a difference terrain type can make in serverless chunk creation latency.

We therefore run the baseline seed against 10 other random seeds using the same stress test workload, while player number and speed remains unchanged. For simplicity, the 10 seeds we used in this experiment are human-readable strings that get hashed to numbers internally.

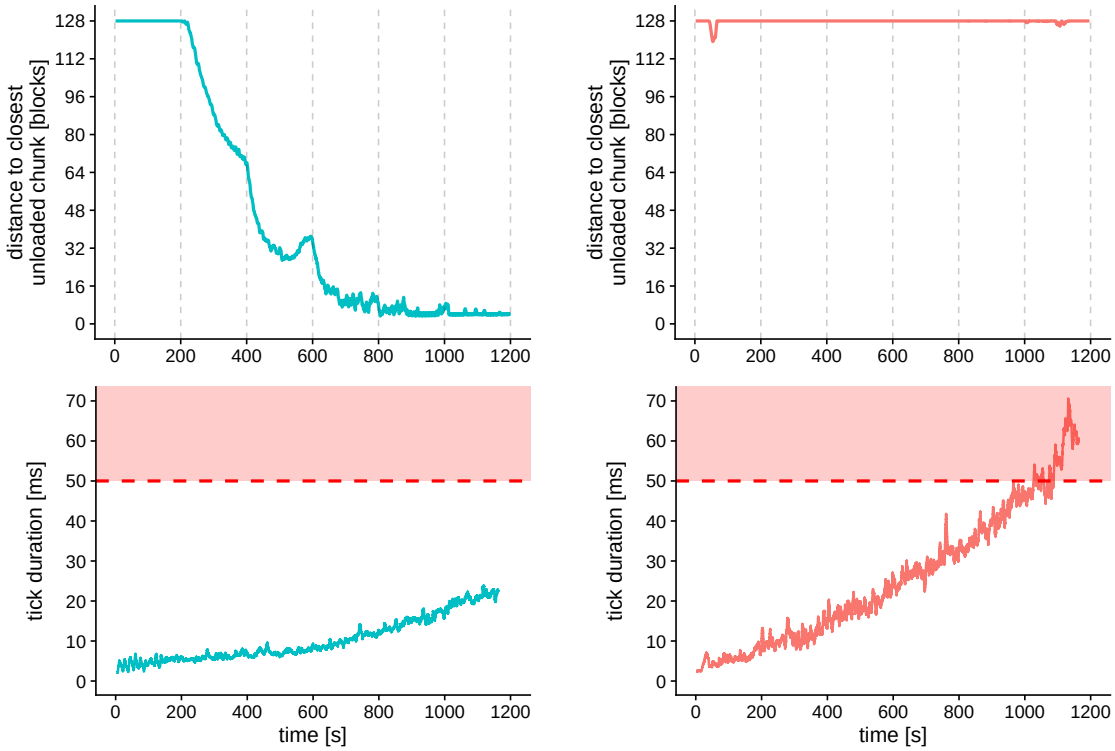
Figure 9 shows how the ten seeds perform against the baseline. The plot is trimmed as to better visualize the boxplots. We observe that the median latency of serverless chunk creation varies between 1 and 1.3 seconds. However, the variance for some seeds is much higher than for others. This is most likely due to changes in biome during the course of the experiment. Both the median and the inter-quartile range of the baseline seed exceed the overall combined value, therefore we conclude that the baseline seed offers a fair representation of the average seed performance.

5.5 World Creation Scalability (leads to MF3)

With this experiment we aim to stress-test the system's world creation component to observe scalability and throughput differences between the serverless and monolithic systems.

To test the limit of the systems, we introduce an incremental workload: the bots start moving with a speed of 1 block/second and every 200 seconds their speed increases by 1 block/second, therefore reaching a maximum speed of 6 blocks/second.

To measure player-perceived drops in Quality of Service (QoS), we measure the minimum distance from any player to the center of the closest un-populated chunk and plot it over time. The result is displayed in 10. First, we notice the AWS Lambda cold start impact on the performance of the serverless system around 30 seconds into the experiment as performance drops, then quickly recovers. We can observe that the serverless system



(a) Local creation.

(b) Serverless creation.

Figure 10: The charts on top show the QoS metric comparison between serverless. We plot the minimum distance between a player and the closest unloaded chunk, as calculated each tick. The distance is measured from the player position to the middle of the unloaded chunk. The plots on the bottom show the tick duration during the experiment. When tick values reach the red area the server is overloaded.

manages to create all the chunks in the field of view of all 5 connected players regardless of speed. The local creation system can only keep up with demand at the starting speed. When the speed increases (after 200s), the QoS drops significantly, finally reaching below one chunk distance as player speed increases to 4 blocks/second (after 600s). This means that the server can only create the chunks immediately adjacent to the player.

In Figure 10 we also plot the tick duration over time to see if the server gets overloaded. The graphs suggest that when using the serverless system the server gets overloaded after players reach a speed of 6 blocks/second. The increase in tick time is caused by a (less disruptive, but still present) simulation component that manages the state of certain loaded chunks. Since the serverless system creates more chunks, the tick time also increases. Therefore, this does not necessarily indicate poorer scalability with the serverless system. With the implementation of efficient chunk unloading, the tick duration should remain constant over time.

We then plot the chunk creation throughput, in chunks per second. The results are shown in Figure 11. Note that the first data-point appears around 30 seconds into the experiment, i.e. the duration of server start-up. We immediately notice that the local

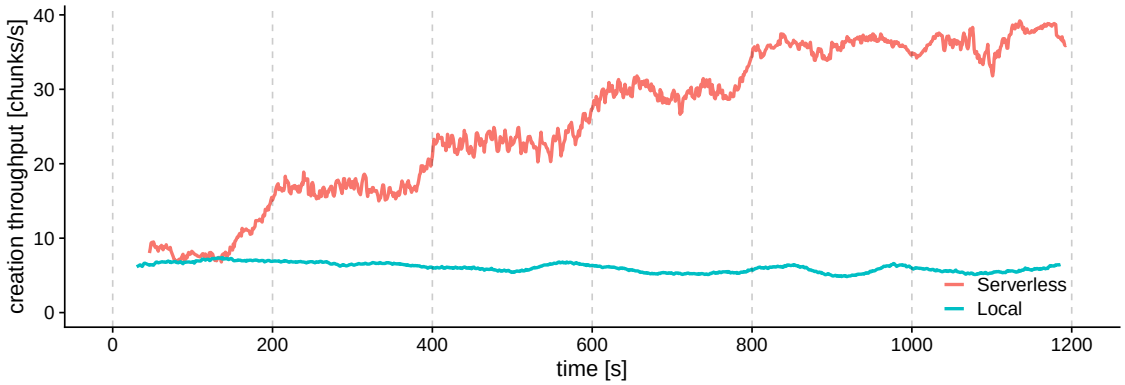


Figure 11: Chunk creation throughput comparison.

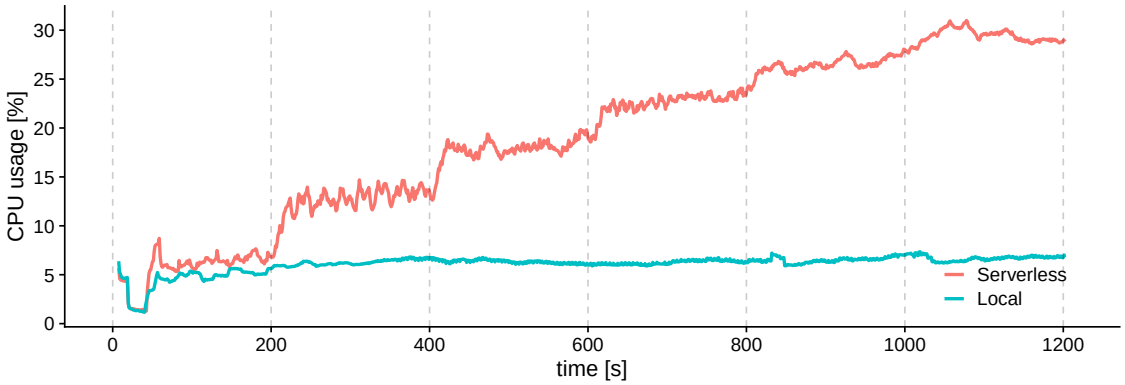


Figure 12: CPU usage comparison. (Maximum is 100%)

creation system's throughput remains approximately the same throughout the experiment. This explains the drop in QoS as player speed increases. This result reflects the poor performance of the local creation system, induced by locking mechanisms that prevent proper use of multi-threading. When looking at the serverless system, we notice the throughput increasing every 200 seconds, as to accommodate the higher creation demand when player speed increases. This result tells us that the serverless system achieves higher performance due to increased use of parallelism, thanks to asynchronous requests. In the end the serverless system achieves a throughput approximately 7 times higher than with local creation.

When plotting the CPU usage (Figure 12), we notice a similar pattern as in the throughput graph. We show the CPU percentage used over time (out of 100), calculated as the sum of usage of all cores and normalized by dividing by the number of cores. After the initial 30 second start-up period, the CPU usage curve almost perfectly matches the throughput curve: for local creation it remains almost constant, while in the case of serverless we see increases in the usage percentage every 200 seconds, reflecting an increasing number of requests made to the serverless platform. Although the serverless system uses more CPU, it never surpasses 30% usage, leaving plenty of room for other components, or for a larger thread-pool.

6 Discussion

In this chapter we discuss the significance of the experiment results, our findings, and talk about the future of serverless computing in scaling online multiplayer gaming systems.

Scalability The stress test experiment in Chapter 5.5 shows that, in a vacuum, serverless computing improves the scalability of the world creation component. In practice, the (Opencraft) server cannot currently benefit from such an addition due to bottleneck in other components, as we have observed in Chapter 5.2 (mainly due to the bottleneck of simulation). It is therefore important to see the contribution of the implemented serverless system in the context of a future highly-scalable MVE server, where each component scales individually using serverless computing.

Cold Start In our experiments we noticed a 30 second runtime initialization latency after the serverless function is idle for more than 5 minutes. For our experiments this is not a problem, as the function has enough time to warm before the players notice any significant increase in creation latency. Furthermore, in our workload, players move constantly, keeping the function warm. Although we lack real-world (trace-based) workloads, we conjecture that in such workloads the players explore less terrain. As such, it is likely that the system would experience more cold starts and therefore periodical increased creation latency. If the latency is noticeable enough to the player, the system could benefit from cold-start prevention to keep the creation function warm. We choose not to implement such mechanism in our system for simplicity.

Scheduler and Executor Policies Serverless computing allows us to create a very large number of chunks simultaneously; performance is therefore a matter of properly making use of it in our system. Although our prototype allows for different policies to be implemented for the creation executor and scheduler, in this work we only implement a simple but performant multi-threaded model and leave room for future developments. More complex policies could be added that could take advantage of the Lambda memory cost-performance trade-off we found in Chapter 5.3, or even employ techniques such as player movement prediction, as to create as many chunks in advance and minimize latency. Such optimizations are beyond the scope of this research, and would currently add no benefit to the performance of the (Opencraft) server.

System design The current design of our system is likely to change when using serverless computing for other components. For example, in our system we include a synchronizer that allows the server to write the chunk data back to memory after serverless creation. In an MVE server that uses serverless cloud storage, this component would not be needed anymore, further reducing the load of chunk creation on the server side. As a result, the serverless chunk creation function would only have to output information that can be directly sent to the players, without any further decoding needed to be done on the server's side. Similarly, moving player simulation to serverless would shift the whole world creation component to serverless as well, eliminating the need for game-state serialization on the server side.

7 Conclusion and Future Work

The popularity and societal impact of Minecraft-like games cannot be denied. Past research shows that current MVE technology suffers from poor scalability, even when the

game server runs in high performance datacenters. In this research, we used serverless computing to attempt to improve the scalability of MVE procedural content generation, by answering three **R**esearch **Q**uestions.

We answer **RQ1** (*How to redesign a real-time monolithic system as a heap serverless system?*) by starting from the reference MVE design we have identified and modifying each component for serverless use. We first add an interface with the serverless platform, through which requests for creation can be made. Secondly, we modify world creation to allow for policy based scheduling and executing. Finally, we add a synchronizer that writes chunk data to memory.

To answer **RQ2** (*How to implement/realize such a system in practice?*), we document the challenges we have faced during development and present solutions for these challenges.

Finally, we answer **RQ3** (*How to evaluate the performance of such a system?*) by proposing a series of experiments and running them on the DAS-5 cluster [4]. Our results show that content generation in MVEs can benefit from a serverless computing makeover, however, its performance is bottle-necked by other components.

In this research we evaluate the scalability of the world creation component using a synthetic workload. Although this gives us a good indication of how the system performs under high load, it doesn't indicate the true impact on scalability of the whole system. The lack of player behaviour data makes us unable to devise a realistic workload. As such, an important direction for future research is modelling player behaviour data in MVEs. This would help all future MVE research in properly benchmarking system scalability.

Serverless computing is a promising technology for solving the scalability problem of MVEs. We would like to see future research attempt to scale different components of the MVE server, most importantly storage and simulation. This would enable further performance improvements, through system (as well as component) design changes and smarter terrain generation scheduling policies.

References

[1] Aws lambda pricing. `https://aws.amazon.com/lambda/pricing/`. Accessed: 2021-07-14.

[2] Opencraft. `https://github.com/atlarge-research/opencraft-opencraft`.

[3] The uncensored library. `https://www.uncensoredlibrary.com/en`. Accessed: 2021-07-14.

[4] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff. A medium-scale distributed system for computer science research: Infrastructure for the long term. *Computer*, 49(5):54–63, 2016.

[5] D. Curry. Minecraft revenue and usage statistics (2020). `https://www.businessofapps.com/data/minecraft-statistics`, 2021. Accessed: 2021-07-14.

[6] J. Donkervliet, J. Cuijpers, and A. Iosup. Dyconits: Scaling minecraft-like services throughdynamically managed inconsistency. In *41th IEEE International Conference on Distributed Computing Systems, ICDCS 2021, Virtual Event, July 7 - July 10, 2021*. IEEE, 2021.

[7] J. Donkervliet, A. Trivedi, and A. Iosup. Towards supporting millions of users in modifiable virtual environments by redesigning minecraft-like games as serverless systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association, July 2020.

[8] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup. A review of serverless use cases and their characteristics, 2021.

[9] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, Mar. 2017. USENIX Association.

[10] M. Hendrikx, S. Meijer, J. Velden, and A. Iosup. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP)*, 9, 02 2013.

[11] N. Herbst, A. Bauer, S. Kounev, G. Oikonomou, E. V. Eyk, G. Kousiouris, A. Evangelinou, R. Krebs, T. Brecht, C. L. Abad, and A. Iosup. Quantifying cloud performance and dependability: Taxonomy, metric design, and emerging challenges. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(4), aug 2018.

[12] A. Iosup, A. Uta, L. Versluis, G. Andreadis, E. Eyk, T. Hegeman, S. Talluri, V. Beek, and L. Toader. Massivizing computer systems: a vision to understand, design, and engineer computer ecosystems through and beyond modern distributed systems. 02 2018.

[13] A. Iosup, L. Versluis, A. Trivedi, E. van Eyk, L. Toader, V. van Beek, G. Frascaria, A. Musaafir, and S. Talluri. The atlarge vision on the design of distributed systems and ecosystems, 2019.

[14] Statista. Video game market value worldwide from 2012 to 2023. `https://www.statista.com/statistics/292056/video-game-market-value-worldwide`, 2020. Accessed: 2021-07-14.

[15] L. Toader, A. Uta, A. Musaafir, and A. Iosup. Graphless: Toward serverless graph processing. In *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 66–73, 2019.

[16] J. Van Der Sar, J. Donkervliet, and A. Iosup. Yardstick: A benchmark for minecraft-like services. In *ICPE 2019 - Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 242–252. Association for Computing Machinery, Inc, Apr. 2019. 10th ACM/SPEC International Conference on Performance Engineering, ICPE 2019 ; Conference date: 07-04-2019 Through 11-04-2019.

[17] Wilkinson et al. The FAIR guiding principles for scientific data management and stewardship. *Nature Scientific Data*, 3, 2016.