Vrije Universiteit Amsterdam

Bachelor Thesis

Duplicraft: Serverless Non-Persistent Instances for Modifiable Virtual Environments

---

Author: Sven Lankester (2668125)

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

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

May 24, 2023

Abstract

Minecraft is the most sold online video game of all time, its main characteristic being that it is a Modifiable Virtual Environment (MVE), meaning that a player can change the in-game world in real-time. Within Minecraft, servers that host a variety of minigames are among the most popular servers in terms of concurrent player count. The problem with real-time alterations to an online environment is that large amounts of data need to be transferred and processed between players, causing MVEs to not scale well. A common solution to this problem consists of a setup where players connect to a proxy server which redirects players seamlessly from a hub world to individual servers hosting minigames to partition the workload each server has. However, this burdens the server operator with a large management overhead, because these servers need to be preemptively set up and actively managed. In this work we design, prototype and evaluate Duplicraft: our serverless system which deploys game instances on-demand, handles the switching of player connections, and allows for the server operator to pay only for the resources used. Through analysis and real-world experiments, we find that Duplicraft, depending on how often servers are deployed, can compete with similar services with a monthly cost. We also deduct that, even though the system meets performance and latency requirements, Duplicraft still suffers from large loading times due to the serverless cold start problem, of which the majority of the loading time is due to AWS.

Contents

1 Introduction **1**

1.1 Problem Statement 3

1.2 Research Questions 3

1.3 Research Methodology 4

1.4 Plagiarism Declaration 5

2 Background **7**

2.1 Minecraft-like Games 7

2.2 Serverless Platforms 8

3 Design of serverless non-persistent instances of MVEs **11**

3.1 System Requirements 11

3.2 Design Overview 14

3.3 Workflows of Instance Creation and Destruction 16

3.4 Design Detail of the Harness and Container Manager 17

3.5 Design Decisions 18

4 Duplicraft: serverless non-persistent Minecraft instances **21**

4.1 Duplicraft Overview 21

4.2 Selection of Cloud Services 23

4.3 Game and Resource Configuration 27

4.4 Low-Level Description of how Duplicraft Works 27

4.5 Implementation Alternatives 30

5 Evaluation **33**

5.1 Experimental Environment 33

5.2 Experiment Design 33

5.3 Experiment Results 38

6 Related Work **49**

7 Conclusion **51**

7.1 Answering Research Questions 51

7.2 Limitations and Future Work 53

References **55**

A Experiment Configuration **59**

B Additional Graphs **61**

C Sanity Checks **63**

1 Introduction

Video games as a form of entertainment have been steadily growing in popularity and revenue in all aspects throughout the years (1), which includes online gaming generating an estimated 18 billion U.S. dollars in the year 2020 (2). Besides their use in entertainment, games are increasingly used in the fields of education and social development, where games have been shown to improve social and communicative skills (3). Minecraft is a canonical example of this, with a separate version of the game existing for the sole purpose of assisting in the education of kids. With Minecraft being the most sold online video game, overshadowing the second most sold online video game Grand Theft Auto V by over 50 million sales (4), and the game being used for both entertainment and education, it benefits society in several ways to have it function well in a large-scale multiplayer environment. However, a Minecraft world is a Modifiable Virtual Environment (MVE), which means that players can interact with and change parts of the world at will. These changes have to be visible for all other players on the server with low-performance variability, meaning that large quantities of players making changes to the environment and updating their location in the environment increase the strain on the server significantly more than players normally would in an online video game using a static environment. A wide variety of research has been done on the scalability of games (5, 6, 7), but as opposed to most games, large amounts of data to be processed and transferred cause Minecraft servers to scale significantly worse and take significant resources to accommodate large amounts of players in the same world (8). Because this processing of command streams and updating the virtual world are a problem when put under intense workloads, the server ends up being a bottleneck. Adding to or improving existing servers is challenging, this is due to servers requiring maintenance in the form of continuous updates and the cost and availability of hardware, alongside the need for people to function as system administrators with certain expertise.

A large percentage of the most concurrently played Minecraft servers are those hosting minigames. It is hard to get scientific data for realistic concurrent and average player counts, because they are easily misrepresented, but multiple sources of publicly available data such as server lists and online articles often show minigame servers, namely Hypixel and Mineplex, to be the most popular (9, 10). On these minigame servers the issue of scalability is currently solved by having a central server where players cannot modify the terrain and use it as a hub to connect the players in smaller groups to different servers hosting minigames where the players can modify the terrain. Running a server this way requires all the individual servers to be preemptively set up and managed, which adds a layer of complexity for the server host and it does not permit paying for the resources used on a fine-grained level, i.e., only when the resources are in use and players are connected to the server.

Cloud computing allows for on-demand resources which can be used for more fine-grained scalability, payment and simplification of system components because they can now run independently on the network of a cloud service provider. This opens the possibility of not having to preemptively set up many small servers, but deploy them as soon as they are necessary on cloud computing infrastructure. Running the small servers on demand allows the server host to pay on a fine-grained scale for the necessary resources and takes away the complexity of the current mandatory management of the minigame servers, making cloud computing an intuitive solution to the aforementioned issues of payment and management. This solution does pose challenges, in that the minigame servers need to start up quickly to not make the players wait for long periods of time. There are also latency requirements and performance constraints that need to be met to keep an online game immersive and enjoyable to play.

This thesis focuses on designing and prototyping a serverless approach to running non-persistent instances of a Minecraft-like environment. The aim is to deploy servers with non-persistent game states on-demand and connect players from the central hub server to one of these freshly deployed servers. Because the on-demand deployment of these servers removes the need for preemptive setup and allows for fine-grained payment for resources, this could help in working towards a more user-friendly environment for hosting these large-scale servers. This will be challenging because online games have strict latency, performance and variability requirements, which are hard to consistently fulfill when operating on cloud infrastructure.

1.1 Problem Statement

The video game industry has consistently increased in market value for years, having an estimated total worth of 178 billion U.S. dollars and over 2 billion gamers worldwide (11). Online video games are among the most sold video games of all time, making the great challenge of scalability in computer science a major point of concern for these games. Especially challenging is the scaling of MVEs because they have to transfer every change to the environment in real-time, to every player of interest, under strict consistency and latency constraints. One of the solutions commonly used for this problem is to divide up large quantities of players into smaller groups on different servers. However, this solution can be refined, because it currently requires the server host to preemptively set up these servers. This study aims to improve the existing system for this division of players by designing and prototyping a serverless system to allow for fine-grained payment of resources and lessen the work for the server host.

1.2 Research Questions

For this research, we aim to see whether it is possible and productive to develop a serverless system for non-persistent instances of MVEs. To do this, we split the task up into the following three research questions (RQs) to be answered throughout the paper:

RQ1: How to design a system for serverless multiplayer minigames in MVEs?

As mentioned in the background section, the serverless approach might be a way to start finding a solution to making Minecraft-like environments scale well and allow for fine-grained payment of server resources. However, designing a serverless system for instances of modifiable virtual environments is challenging because there are strict constraints that should be met to keep an online video game immersive and enjoyable. After all, large latency and long wait times reduce the quality of experience a player has.

RQ2: How to implement a prototype of such a design?

This implementation would provide a proof of concept if the system works and allows us to evaluate how well the design works in practice. Prototyping the aforementioned design allows us to see if this theoretical system is feasible to be used under realistic workloads and if it can do so under strict performance constraints. The main reasons which make the development of this prototype challenging would be not having proper mechanisms for hiding latency, attempting to create serverless instances in a short enough period to keep the game engaging for the players and making the transition to these instances seamless to a degree from the player's perspective. Combining several different technologies to tackle these challenges will be difficult to do.

RQ3: How to evaluate the design, through real-world experiments using the prototype?

To evaluate the efficacy of our design and implementation is crucial to see whether the design is feasible for real-world use. There is no agreed-upon way to measure game performance because there are many metrics to be taken into account, and the quality of gameplay experience is the most important. It will be challenging to find the correct metrics to determine if the prototype can function under the given performance constraints, or even function at all. This indicates that latency-related metrics will be of great importance but will need to be weighed against the cost of reaching the measured performance, because not only finding the right metrics is important, but also finding a balance between the importance of each one of the metrics. An example would be that an instance could be created in a location with optimal latency for all players, but finding this location took several minutes which means the players were kept waiting for that duration, or whether the implemented system is worth it to switch over to in the first place. Another challenge will be determining and obtaining meaningful workloads which simulate a real-world workload. Not only is determining these metrics difficult, but it is also challenging to gather them. Because there is no industry standard for gathering metrics from MVE instances, we will have to manually adapt existing tools to allow us to do so.

1.3 Research Methodology

1. (Matches **RQ1**) Good design is essential to identifying problems and ensuring these are handled properly in the implementation. For the design, we will be working with the AtLarge design framework (12). First, the requirements should be analyzed in terms of functional and non-functional requirements to get a good overview of what the system will need, with a focus on non-functional requirements such as performance constraints and fault tolerance. We must also research and understand what alternatives are available for the choices we end up making for the final design to help us reason why the choices we made were the correct ones. Then once a good idea of the system has been established, the interaction between its components will

need to be laid out, followed by experiments on the cloud service provider of choice we will use to tweak the design made up until this point for it to be realistic within the boundaries imposed by the tools we are using. Once this realistic idea is properly finished a high-level design can be made which will then lead to a detailed design for the implementation.

2. (Matches **RQ2**) This research question builds directly upon the first, aiming to become a working prototype of the initial design. The main goal of this implementation is to see whether or not we can allow for fine-grained payment of server resources and lower the amount of setup required for a server owner. To build a prototype, we work with tips described in The Pragmatic Programmer (13). This prototype will be built using containers on AWS, because it is a very popular and widely available cloud service provider, and using a modified server image to start the container with to allow us to perform the necessary customizations, such as world management. In the approach to solving this research question, there will be hands-on experience in working with AWS, which will require the knowledge gained in the process of solving research question 1.

3. (Matches **RQ3**) For evaluation we will use a tool called Yardstick (14), an existing benchmarking tool for Minecraft-like environments that has been used to evaluate modifiable virtual environments in the past. The prototype will be tested under various workloads to see if it can function under the strict performance constraints of real-time online video games. Yardstick will most likely have to be tweaked to see if desired results can be reached and to produce a realistic enough workload under which the prototype can be evaluated.

1.4 Plagiarism Declaration

I confirm that this thesis work is my own work, is not copied from any other source(person, Internet, or machine), and has not been submitted elsewhere for assessment.

2

Background

This section details the technologies that this thesis uses as a foundation. First, §2.1 Provides a high-level understanding of Minecraft-like games and discusses an existing solution to the scalability problem. Following, §2.2 mentions existing serverless platforms, explains common deployment strategies and elaborates on how they can benefit the user.

2.1 Minecraft-like Games

A client-server architecture is popular in massively multiplayer online games, Minecraft-like games generally being no different. It is also common for online games with many players from around the globe connecting at once to have individual servers for specific regions of the world to lessen the workload of each individual server for the purpose of meeting non-functional requirements such as low latency. Due to Minecraft-like games providing the player with a sandbox experience, where every modification a player makes to the virtual environment has to be forwarded to every other player of interest, the server becomes an even larger bottleneck than in most other types of online games.

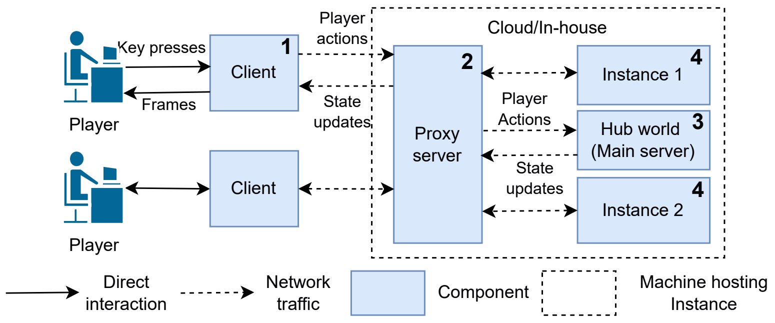

Figure 2.1: Model of a common method to connect Minecraft-like servers.

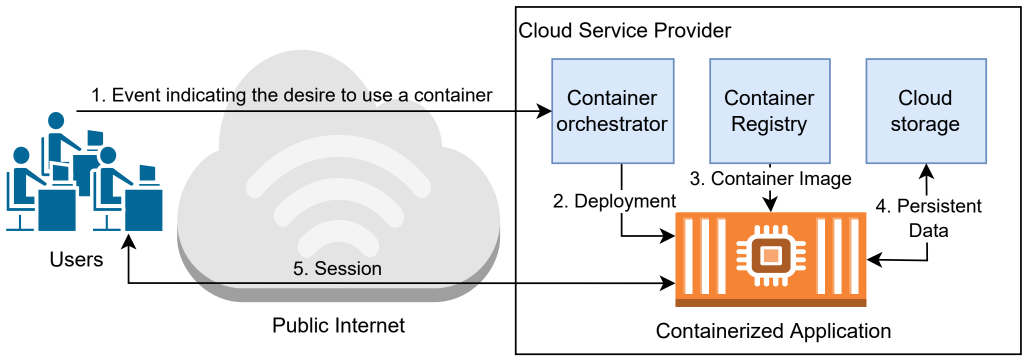

Figure 2.2: Overview of the Container as a Service deployment strategy.

As was shown in §1, a large portion of Minecraft's players are interested in minigames. The concept of partitioning the server workload among smaller servers can apply here as well to solve a part of the scalability issue, as is displayed in Figure 2.1, where the smaller servers (**4**) are hosting non-persistent instances of a Minecraft-like server with the selected minigame as its world. This is a commonly used solution in Minecraft's largest minigame servers, where, for example, a modified Minecraft server called Spigot is often used alongside BungeeCord (15) which acts as a proxy (**2**) between the client (**1**) and server (**3**) to connect different servers (**3**,**4**) together seamlessly, shown in Figure 2.1. This allows for the minigame server to have the main host on a single IP address to function as a hub world (**3**), where players cannot modify the terrain to allow for more players to be connected at once. The players then maneuver through the hub world and can connect to these non-persistent instances through in-game actions such as chat commands or interacting with an in-game object in a way that avoids having to notify all other players (e.g., clicking on an object in the game without having it visually represented to other players). This construction of having many servers connected seamlessly allows server hosts to provide a large number of players with many different minigames through what is perceived by the individual players as one singular server they only have to manually connect to once.

2.2 Serverless Platforms

Figure 2.2 depicts a common deployment strategy in serverless computing. Serverless computing is an execution model in which a cloud provider dynamically allocates cloud-computing resources without the user requesting them having operational concerns, such as the maintenance of the server. The requested resources execute the code requested by the user in an environment specified by the user, and only the used resources are paid for on a fine-grained scale (i.e., the user pays only for those resources that are used as a result of their request). Besides the specifics of the request made by the user about the environment and what code should be executed, all other concerns such as management of the host's operating system by means of updating it and scaling the resources needed are managed by the selected cloud service provider.

Popular deployment strategies when working with serverless platforms are *Function as a Service* (FaaS) and *Container as a Service* (CaaS). This work focuses mainly on CaaS. With FaaS you host a bit of code on cloud computing infrastructure which gets triggered based on an event, and once it is completed will return a result. The cloud computing service provider will automatically scale and manage the resources allocated for the program when necessary, which is one of the main benefits of serverless computing. In contrast, CaaS deploys containers on cloud infrastructures, of which a simplified overview is depicted in Figure 2.2. This overview shows a workflow where a user wants to use a container, this container is then deployed and it retrieves the container image and persistent data from additional resources on the cloud. This workflow is simple because, in reality, many more cloud components are often working in collaboration, most of which are optional based on the user's needs. Containers are software packages that include the code and all its dependencies so it will be able to function without the programmer having to concern themselves about the infrastructure it is running on. When using CaaS, the cloud service provider will deploy the specified container as an isolated environment on the computing resources, allowing the user to host software packages (e.g., a Minecraft-like game server) and only pay for the amount of resources allocated for the amount of time they are in use while having none of the operational concerns. Popular examples of CaaS services would be AWS Fargate and Azure Container Instances.

To deploy a serverless system, one needs access to cloud-computing resources, generally provided by a cloud service provider. Many such providers exist on the market, the largest one by market share at the time of writing this paper is Amazon Web Services, followed by Microsoft Azure and Google Cloud(16).

3

Design of serverless non-persistent instances of MVEs

In this section, we introduce and explain the design of Duplicraft in detail. Duplicraft is our serverless system for non-persistent instances of modifiable virtual environments. It enables on-demand access to server resources and allows for fine-grained payment for the allocated resources.

3.1 System Requirements

This design focuses on the following requirements:

R1: Duplicraft should support at least 50 players playing concurrently on the same instance.

An important feature of the non-persistent instances that Duplicraft aims to deploy is that they host worlds that are of interest to groups of players. Supporting at least 50 players is important because several Minigames depend on more than 30 players interacting with each other in real-time, meaning that supporting more than 50 players would allow Duplicraft to host a large variety of minigames. Supporting at least 50 players is difficult, as simply always providing excess resources to an instance would lead to an increase in cost, which is an undesirable property for a serverless system to have. So designing a system with this in mind requires a component to be able to specify to the cloud provider how many resources each specific instance needs and for this component to be integrated well with the other components in the system.

R2: Latency to an instance should meet latency requirements of real-time MVEs during play.

Seeing that our main concern is minigames in MVEs, which often rely on fast reactions to in-game events for their gameplay, we see meeting latency constraints as a very important property for the system to have. The experience a player has in a game can be heavily influenced by high latency. Not receiving the environment state within the time a server sends a game state update to other players could cause inconsistencies between players, which negatively impacts the quality of experience (17).

R3: Duplicraft should manage instances in a way that allows for fine-grained payment of resources used without paying for reserved but unused resources.

As one of the main advantages of serverless infrastructure is the fine-grained payment scale of resources used, it is important to design a system that allows for fine-grained payment. One of the current flaws we identified with existing solutions was the issue of having to pay for the non-persistent instances even when the resources are not being used. This makes it important for resources to be allocated on-demand and deallocated when they are no longer in use. The challenge in this requirement comes from designing the system to manage instances efficiently. That implies that the system should be designed to be able to ensure resources are used efficiently and unused resources are stopped as soon as a stopping condition is met to avoid having the user pay for an unused instance.

R4: Duplicraft should allow for an instance to be initialized with a world selected from a library of worlds.

A large part of the appeal to hosting non-persistent instances which often only run for several minutes is the variety of choices. Minigames are a prime example of the appeal of variety, with the most popular Minecraft minigame servers advertising choice between upwards of 30 different minigames. To ensure this variety, Duplicraft must support instance creation with a dynamically chosen world, in this case using a minimum of five worlds.

R5: Duplicraft should be easy to use.

A large motivator for this research was lessening management overhead from the server host. For that reason, we wish to design a system that is nearly plug-and-play at its core and requires little manual adjustments to the existing infrastructure needed to host a system. The only alterations made to pre-existing components of the MVE should be limited to allow it to communicate with the system we design. This is challenging as we are designing a system that seamlessly needs to integrate with an existing system to introduce new features. Doing so will require well-thought-out methods to circumvent limitations imposed by the existing infrastructure, such as the seamless world switching R4 aims to achieve that is not inherently provided by many existing games.

R6: Duplicraft should be compatible with multiple MVEs.

To make this research appealing to the largest group possible and broaden its use, we wish to design a system that does not specifically target a single MVE. Similarly to R5, this is challenging because it implies that we will have to design around existing infrastructure without making it game-specific. It will require an understanding of common multiplayer infrastructures in MVEs and thorough consideration in regard to building a system around the existing components of such an infrastructure.

It is notable that many, if not all, requirements related to distributed systems are important to online video games. However, addressing each one individually is far beyond the scope of this research. Consistency, for example, is highly important for real-time multiplayer video games. In the case of minigames in MVEs, players experiencing the same in-game environment at the same time can be integral in a scenario where real-time interactions are of high importance. However, consistency in real-time multiplayer games, especially in MVEs, can be particularly challenging and often requires intricate mechanisms such as situation-specific consistency policies. This can introduce far more complexity to a design, placing it beyond the scope of this research. Other examples of such requirements are reliability and availability. There are several reasons why these requirements are not addressed. The main reason is once again similar to why we do not address consistency; the complexity that this requirement would introduce exceeds the scope of this research. However, we also wish to design this system to work with a cloud service provider, in which case it is hard to control variables such as availability as it is now a concern for the cloud provider instead of the server host. Mechanisms could be designed to, for example, switch between cloud providers if one becomes unavailable. However, such a system once again can quickly become very complex and require significant time to design and potentially prototype. These requirements are only a few examples of common system requirements to consider when designing a distributed system.

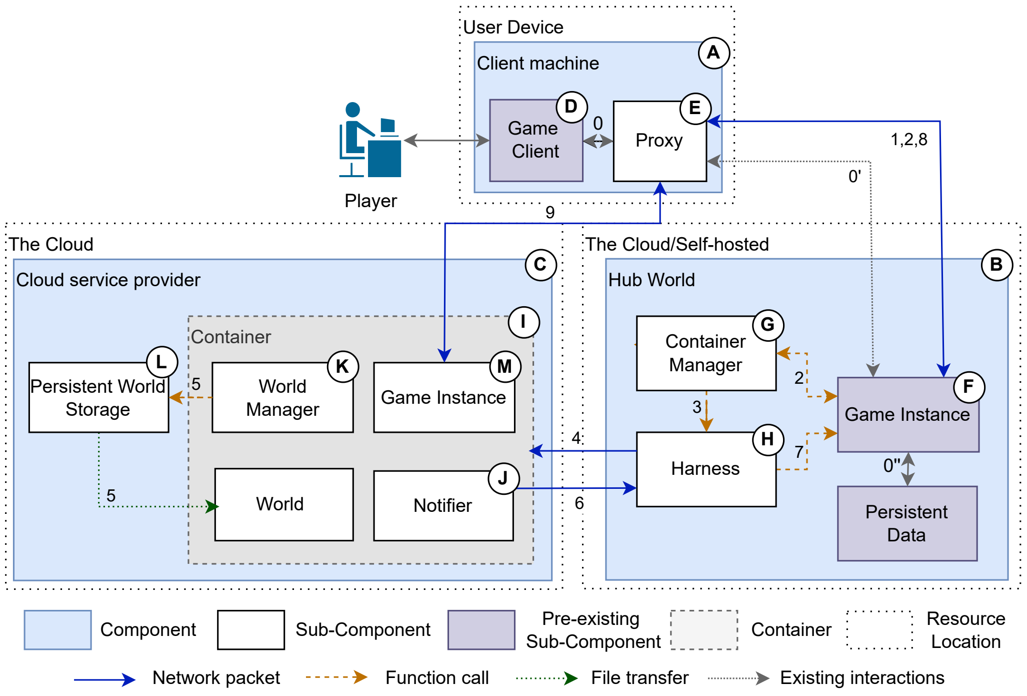

Figure 3.1: Design overview of Duplicraft.

3.2 Design Overview

Depicted in Figure 3.1 is an overview of the design of Duplicraft. It shows how the components introduced by Duplicraft work together with the existing client-server architecture of an MVE. For our design, we mainly look at a set-up using three machines: A client machine (**A**) which takes player input and hosts the game client that the player sees, a Hub World (**B**), which is a server that is constantly running (either self-hosted or rented from some provider of computer resources), and the machine on the cloud (**C**) (i.e., the resources allocated by the cloud service provider). In our design these machines are separate, but the Hub World could also be running on the same cloud service provider as the deployed containers or could be self-hosted on the client machine.

The Game Client (**D**) needs to be able to have its connection switched from the Hub World's game instance (**F**) to a Game Instance deployed on the cloud (**M**) with a world of the player's choice. To make this possible, the Proxy (**E**) runs on the client machine and the game client connects to the Hub World through the proxy. The proxy can then seamlessly switch connections by forwarding the packets to the location of another Game Instance.

To allow the Game Client to switch connections, the Proxy needs to know where the container is. The Hub World's Game Instance informs the Proxy of the location of the requested Game Instance because it is the main point of handling communication with the Proxy from the Hub World. To handle this connection switch we specify a user input, a chat command for example, of the developer's choice to be handled in the Game Instance which serves as a request to deploy a container, to which the Game Instance can respond with a message to the Proxy with a location of the deployed container. First, we need to make sure that multiple containers can join the same Game Instance so they can play together, so we add a Container Manager (**G**) which keeps track of existing containers and if they are accepting new connections.

To address **R1** (support up to at least 50 players) and **R2** (low latency), we need to communicate with the cloud service provider for a container (**I**) to be deployed on the cloud with specific requirements. For supporting up to at least 50 players, this implies we need to dedicate an amount of memory and computational power to reach performance requirements under the expected workload for the container. The necessary amount of resources a container is deployed with can be specified either for each type of container or for the number of players the container should be deployed for. For the latency requirement, we specify the location the server will be deployed in, chosen as the hosting location having the lowest mean latency for each player connecting to the instance, to achieve a latency lower than 50 milliseconds during play. For players who inherently have a latency higher than 50 milliseconds due to being geographically far apart, two separate instances are started in different regions. For this, we add a Harness (**H**), which handles all communication with the cloud service provider, to have control over the container specifics, and the deployed container so that the location of the container can be forwarded to the Proxy through the Game Instance of the Hub World. This harness is the most complex part of our design, which we discuss more in-depth in §3.4.

To meet **R4** (Switching between a library of worlds), the container should be able to be started with a variety of worlds. To avoid being reliant on downloading these worlds from the public internet, which may be slow, we store them in a Persistent World Storage (**L**) provided by the cloud service provider in the form of a blob or bucket storage for example, so that we have easy access to worlds we store ourselves. The image with which the container is started is a slightly altered MVE server. Two components are added to the container image. To allow for the retrieval of the world specified by the Harness from the Persistent World Storage, we provide the container image with a World Manager (**K**) to handle communication with the cloud service provider's Persistent World Storage from inside the container.

To satisfy **R3** (fine-grained payment), the container will need a way to notify the Hub World of its IP address when it has finished deploying or notify the Hub World of it soon being stopped. To only pay for resources that are used, containers must be released when either no players are connected to the world or a stopping condition, such as the minigame being finished, has been reached. To achieve this, a second component, the Notifier (**J**), is included in the container's image to handle the network traffic between the container and the Hub World from inside the container.

In a common client-server setup for an MVE all traffic would be sent directly from the client to the server. To allow these existing interactions (**0, 0', 0"**) to function properly, we must adjust them to fit our system's components. Player inputs, such as mouse movements and key presses, are processed by the game client as it normally would and the game's visual output to the player works the same way as in a regular setup. However, the Game Instance must now connect to the server through the Proxy running on the local client, which forwards these client and server messages to the correct location.

We satisfy **R5** (easy to use) and **r6** (compatibility) by having designed all our components to exist outside of the pre-existing ones, with the pre-existing components being based on a typical client-server configuration to account for compatibility. In doing so, we allow the system to seamlessly integrate into a typical MVE client-server setup without the need for complex alterations of the existing infrastructure. The only direct alterations needed in this design would be to the Hub World's game instance (**F**) to allow for it to communicate with the components we introduce, which is in the form of function calls.

3.3 Workflows of Instance Creation and Destruction

To create an instance, the player performs an in-game action that shows the intent to play on a non-persistent instance (e.g., sending a chat command). The Proxy forwards this interaction to the Hub World (**1**). The Game Instance checks with the Container Manager whether or not an instance of the selected world already exists and is still accepting players, and if it does, informs the Proxy of the location of the container so that the game client's connection can be swapped correctly (**2**). If an instance of the requested world does not exist yet, the Container Manager informs the Harness that a new container should be deployed (**3**). The Harness requests the cloud provider for enough resources to be allocated to ensure performance and a container to be deployed with a specified world (**4**) in a region close to the player. Upon creation, the container's World Manager retrieves the world from the persistent cloud storage (**5**) so that the container deployment can be completed and the server will be functioning as intended. Upon completion of the creation process, the Notifier informs the Hub World's Harness of the location of the container (**6**), which proceeds to pass this onto the Game Instance (**7**). Once the Game Instance forwards the location of the container to the Proxy (**8**), the game client's packets can be successfully forwarded to the MVE server running in the container.

To avoid paying for resources currently not in use, the container should be shut down upon reaching a final state or having all players disconnect. To achieve this, the Notifier informs the Hub World that the container is being stopped by sending a message to the Harness (**6**). The Harness forwards this to the Game Instance (**7**) in a similar fashion as is done for the instance creation, so that the Game Instance can notify the Proxy of the need to change the connection back to the Hub World (**8**). Upon successful completion of the connection switch, the game client will be connected to the Hub World again and the container will be destroyed as it is no longer in use.

3.4 Design Detail of the Harness and Container Manager

The Harness acts as the interface for the Hub World to interact with the cloud service provider, so it is necessary to have this component be highly configurable to meet several of our system requirements. To meet latency requirements, the Harness can be configured by the Hub World operator to deploy containers in several regions provided by the cloud service provider, so that a Game Instance can be started in a location close to the user requesting it. This can be achieved dynamically by having an algorithm decide on the location where the server should be hosted based on the location of the players requesting the container to be used. Another important requirement that is met by the Harness is supporting up to at least 50 players. To support enough players in certain worlds, the Harness can be configured to request varying amounts of resources from the cloud service provider, such as CPU units and the amount of memory, based on the number of players and the type of world that will be requested. The final important configuration requirement that the Harness helps to meet is to support a library of worlds. The Harness must have a way to inform the container with which world it needs to be started.

Players should be redirected to existing containers if an instance with the correct configuration for their request already exists and is in a waiting state. For this to be possible, it is important for the Container Manager to store the relevant data for each running container, acting as a database of deployed containers. To ensure players can only join instances that are still waiting for players, the Container Manager should only keep track of the instances that are still in an awaiting state. Players should not be connected to containers located on the other side of the world to prevent large amounts of latency, so the Container Manager should also store the location of the container. To properly redirect new player requests to the correct instances, it is also important for the Container Manager to know which world each instance is using.

3.5 Design Decisions

The Proxy could run on the Hub World instead of the client machine, where in contrast to the proposed design, all Game Clients would connect to a single proxy on the Hub World rather than each Game Client having their own. This would allow for centralized management of all connected players, but we do not do this for the following reason. We identified in §1 that the server handling many clients at once is a big bottleneck for MVEs, thus we would like to avoid processing all player packets in a single location (i.e., the Proxy server running on the Hub World in this example). Having the single Proxy handle too many connections at once could be a cause for performance issues and packets being delayed, potentially violating **R1** and **R2**.

To handle client requests for instances to be deployed, we propose that the game instance handles a chat command or other interaction of the developer's choice to be used as a deployment request. However, if the Game Instance is not easily modifiable to do so, a way to circumvent it is as follows. A proxy server would be running as the Hub World's Game Instance, and the actual Game Instance would run in a container to avoid two applications trying to listen on the same port. The proxy server then forwards all game-related packets to the port the container is listening on through a self-feedback loop, so that the actual Game Instance can process them as it normally would, but the proxy server can then be configured to request a container to be deployed based on certain packets. This consideration, however, adds extra complexity whilst achieving the same requirements as processing the commands directly in the Game Instance, and was thus omitted from the final design.

Rather than having a library of worlds hosted by the cloud service provider, each world could be packaged into its own container image with predefined settings specific to the world. This allows for faster load times, because the world does not have to be retrieved separately anymore, and for optimization of container images on a per-world basis. However, this consideration is not in the final design, because this design is partially proposed to solve the amount of management required for the user hosting the servers. Creating each server image individually for every single world would both take up more storage space on the cloud, increasing consistent cost and requiring more extensive preparations by the user who manages the Hub World and its interactions with the cloud service provider.

4

Duplicraft: serverless non-persistent Minecraft instances

In this section, we discuss the details, considerations and challenges of a Minecraft-based implementation of the design shown in §3.2. We focus mainly on the implementation of components E, H, I and L and their interactions with existing components D and F.

4.1 Duplicraft Overview

Depicted in Figure 4.1 is an overview of Duplicraft, our Minecraft-based prototype which connects several AWS services to deploy non-persistent Minecraft servers, which the Duplicraft proxy then can switch the connected client's connection to once it has finished deploying. Duplicraft is based on Minecraft because it is the most popular MVE it has a large community that offers many resources to aid the development of third-party applications. Given that Duplicraft is based on Minecraft, and Minecraft is written in Java, we chose to develop Duplicraft in Java to ensure compatibility.

To configure Duplicraft, the user is required to configure Duplicraft to utilize AWS resources and specify the details an instance should be deployed with (detailed in §4.3). For the use of AWS, an AWS account and three separate services in the regions in which they will be deploying containers is required, the reason for selecting these services and their use is detailed in §4.2.

After the setup is completed, the user can start the prototype, which initially only runs a proxy server. The proxy server connects to a Hub World specified in the configuration. Once a user connects to the proxy server all packets essential for the communication between a Minecraft client and server are forwarded from the connected game client to the

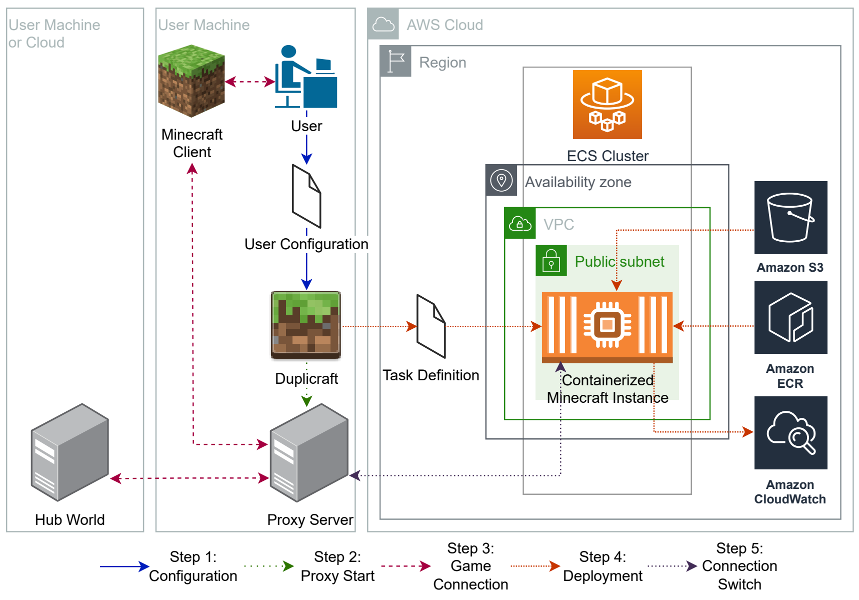

Figure 4.1: Duplicraft usage by step.

Hub World. The proxy intercepts two specific chat messages, if the user types "/swap" in the in-game chat, the proxy request for a container to be deployed on AWS. Once the container deployment is finalized and the Minecraft server is accepting connections, the proxy will forward the game client's packets to the container. If the user types "/hub" in the in-game chat, the user will be reconnected to the Hub World and if a container is running, it will be destroyed.

One of the most challenging and complex parts of the implementation is the proxy server. The proxy server was difficult to make because the seamless swapping of servers comes with many issues when taking an intuitive approach, such as indefinitely getting stuck on the screen indicating that a world is loading, and discovering how to properly set up a new connection, then swap from the existing connection to the new one without disconnecting the player. We detail the issues and solutions in §4.4.

The other difficult part to implement is the AWS interface Duplicraft. There are two main reasons for this being challenging. We want the system to be highly configurable so each server instance can be configured depending on the expected workload, you would not need the same amount of resources for a 2-player minigame as you would for a 16-player minigame. This directly relates to the second reason it was difficult to implement, being the required understanding of AWS services and how they can interact with each other. To make the system configurable, we need to know what AWS allows us to configure and what steps are required to configure our AWS services. To reduce complexity for the user, we then need to implement components that are simplified interfaces for the user to configure, which we elaborate more upon in §4.4.

4.2 Selection of Cloud Services

In §2.2 we mentioned several cloud service providers. For this prototype, we have decided to make use of Amazon Web Services for various reasons. One of the primary reasons for it is that similar to Minecraft in the MVE game genre, it is currently the most popular cloud service provider and thus it has a large number of users, meaning there are many resources and helpful communities online making it more accessible. AWS also has a large variety of tools that make container deployment widely accessible to new users, most importantly their Elastic Container Service (ECS) combined with their Elastic Container Registry (ECR). Finally, AWS has many available regions and availability zones all across the globe, with the ability to deploy containers on them dynamically based on your needs. This is important because deploying a container close to a user allows you to meet latency requirements.

AWS has several services for the deployment of containers, such as their Elastic Kubernetes Service (EKS), Amazon Lightsail, Fargate and Elastic Compute Cloud (EC2), the latter two being services offered under ECS. Lightsail is a service based on a fixed, monthly price and thus does not meet our fine-grained payment requirement. We deem Kubernetes to add too much complexity for this prototype, thus EKS is also taken out of consideration. EC2 is a server-based service in which you use a virtual machine image to host virtual machines on your AWS account. Fargate, on the other hand, is a serverless service that allows for payment on a per-minute scale where you only get charged for the amount of virtual CPU (vCPU) and memory your container ends up using. For these reasons, we end up using AWS Fargate for this implementation.

Step 4 of Figure 4.1 displays how Duplicraft combines several AWS services to deploy a container, and Figure 4.2 shows the cost at the time of writing this thesis of Duplicraft compared to other services, which we further discuss in §4.2.1. ECS allows Docker images to be run by defining them as tasks, with tasks being instances of Docker containers. All container configurations, such as how many vCPU units the container should have, the amount of memory it can use and the environment variables are specified in the Task Definition.

Service	How payment is calculated	Price
Duplicraft (Fargate)	Billed per second, per resource used + Monthly cost for additional services	€0.046 per vCPU/hour €0.005 per GB memory/hour €0.62 per GB of logs/month €0.03 image on ECR/month €0.024 per GB on S3/month
EC2 on-demand	Billed per second, based on instance	2vCPU, 4GiB memory: €0.082/hour 4vCPU, 8GiB memory: €0.15/hour
Lightsail	Monthly cost based on container type	2 vCPU, 4GB memory: €39.45/month 4 vCPU, 8GB memory: €156.80/month
Minecraft Realms	Set monthly cost	€7.19/month

Table 4.1: Payment details per service.

The container is started using the specified Task Definition as the container configuration, where we download the container image from ECR rather than the public internet, so we are not depending on downloads from the public internet as a means to shorten the deployment time. World files are subsequently retrieved from Amazon's Simple Storage Service (S3) for the same reason. In this case, we store objects in a bucket on S3 and assign a publicly accessible URI to them, so we can use this URI later to retrieve the files we need.

4.2.1 Cost Analysis

Figure 4.2 displays the costs we showed in Table 4.2 compared to each other based on the cost to run a certain amount of minigames, each lasting 10 minutes. For the AWS services that are not Fargate, we show the prices of a selection of the computing resources the respective services offer, which have comparable resources to the containers we deploy using Duplicraft. Besides the individual pricing pages for each service, we used the AWS Pricing Calculator to assist us in determining accurate prices for the other services. We omit supplemental monthly costs from non-Duplicraft services as they are optional for a bare-bones system in the other services, but included in the cost for Duplicraft for reasons

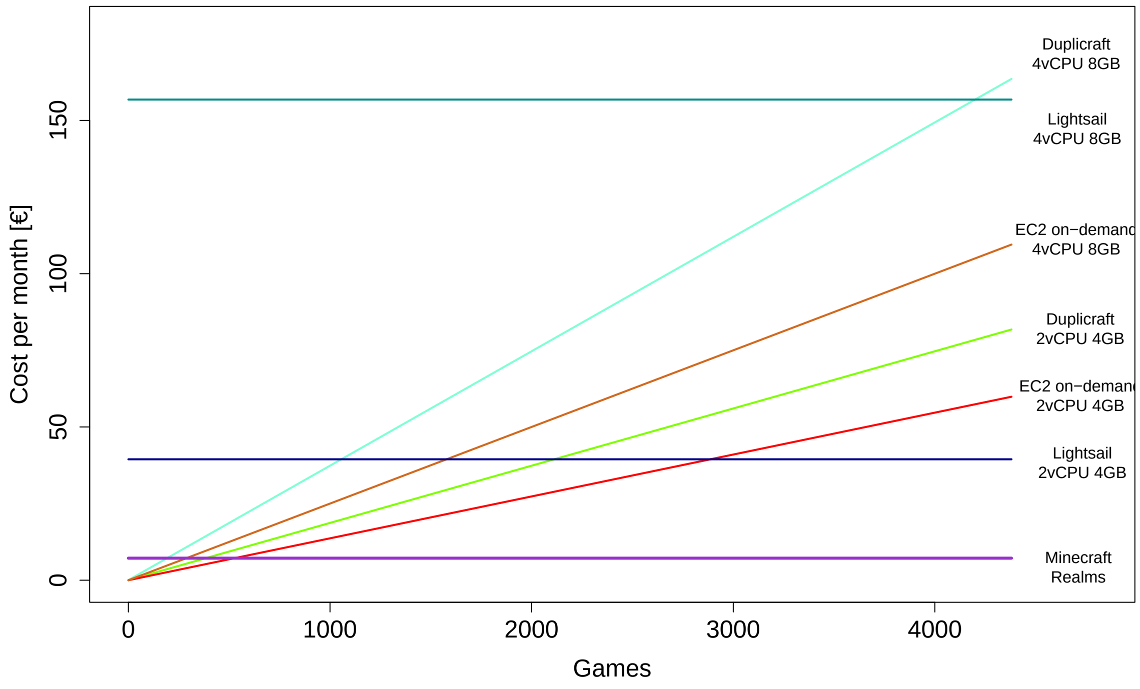

Figure 4.2: Cost of each service with minigames lasting 10 minutes over the span of a month.

named in §4.2.

Besides the resources of the container itself, Duplicraft has extra costs. The cost to store the log seems high compared to the container resource cost but looks deceptively so due to measuring in gigabytes. The amount of logs an instance produces is dependent mostly on in-game events, so in a realistic setting the amount will vary from our results, but 178 instance deployments of our system have generated half a megabyte of logs. If we were to assume 20 containers to be deployed every day, equalling 600 deployments over a month of 30 days, we would have 1.76 MB worth of logs. Assuming a retention rate of a month for these logs on CloudWatch, we pay €0.001 per month to store our logs. Given that we only store our Minecraft server docker image on ECR, we pay €0.03 consistently each month. Assuming you store less than 427 MB of worlds, which is within expectations as small Minecraft minigame worlds generally are less than 10 MB, sometimes even less than 1 MB, the cost of S3 storing these items is less than €0.01 per month. For these reasons we determine the constant monthly costs of Duplicraft (i.e. the costs of storage) to be €0.03 for our calculations.

The figure shows two configurations for EC2 On-Demand, which is the EC2 service for instances with no long-term commitments and payment by the hour, that we want to compare to Duplicraft. We can see directly that 2 vCPU in Duplicraft, without accounting for the memory costs, is already more expensive than the complete EC2 instance with 2 vCPU, the same goes for the other instance. This difference in cost is also clearly visible in the line graph. The difference in cost is mainly due to EC2 having more management overhead. In EC2 you have to manage your own cluster and instance, whereas in Duplicraft the cloud service provider does more management for us. So even though EC2 On-Demand Instances are less expensive, we pay the extra costs to remove management overhead for the Duplicraft user, because it is an essential trait of a serverless system to not have to manage the servers.

Lightsail has a fixed, monthly price for running an instance, we have again selected two instance types to compare to. If we were to run a Duplicraft Instance with 2 vCPU and 4 GB of memory without accounting for the €0.03 monthly costs, it would cost €0.112 per hour. For the other configuration, Duplicraft would cost €0.224 per hour. You could run Duplicraft 351 hours and 699 hours per month respectively for the same price as the Lightsail instances of similar resources. This is less than the number of hours you would get from running an instance all month, without stopping it. However, based on the popular minigame server Hypixel's minigames, the instances would be deployed for around 10 minutes each. This would allow us to deploy 2106 and 4194 instances each month respectively, to match the price of a single Lightsail instance of similar capacity. This would also allow us the flexibility of deploying multiple instances at the same time, which commonly happens on popular minigame servers. In cases where we use Duplicraft instances less than the calculated amounts, the user pays less. So although Duplicraft is technically more expensive to run the same resources for a full month, it comes with benefits that are important for our serverless system.

Minecraft realms is a difficult service to directly compare prices with, because to the best of our knowledge, the exact amount of resources they provision their servers with is not public information. Assuming we want to have 10+ players on our server, the Minecraft wiki (18) recommends us to have at least 8 GB of memory allocated. So for this comparison, we will assume a Duplicraft instance with 2 vCPU and 8 GB of memory, which would cost €0.132 per hour, excluding monthly costs. Compared to Minecraft Realms, we would be able to run the Duplicraft instance for 54 hours per month, which would mean deploying approximately 324 instances of a Minecraft minigame, equalling 10 instances per day for a month. Depending on how popular the hub server is, this could be a viable alternative. However, a Minecraft realm only allows a maximum of 11 players to join, so the comparison is not as clear, keeping in mind that Duplicraft aims to support tens of

Variable name	Description
CPU	The maximum amount of CPU units to be present for the task
MEMORY	The maximum amount of memory in MiB to be present for the task
WORLDURI	URI of the world to start an instance with
IMAGEARN	Location of the container image on ECR
HUBWORLD	IP address of the Hub World
INSTANCEMEMORY	Amount of memory the Minecraft server will use

Table 4.2: Table of important environment variables used to configure Duplicraft.

players. Duplicraft also has the benefit of being able to simultaneously deploy instances and being less expensive when less than 324 instances are deployed in a month.

4.3 Game and Resource Configuration

In Table 4.2 we show the main configurable environment variables with which Duplicraft can be configured. As mentioned in the previous section, the way to deploy containers on ECS is through a Task Definition. It acts as an interface to describe to ECS what parameters the container should be deployed with. Duplicraft stores the task definition as a JSON file locally, with the important configurable parameters filled in with placeholder values. This task definition can then have its placeholder values dynamically filled in so it can be registered to ECS and containers can then be deployed based on its specified parameters.

To fill in the placeholder values in the Task Definition template, we work with environment variables, or alternatively through a *.env* file. The environment variable file specifies all placeholder values as key-value pairs, from which the values are taken to replace the placeholder values. The most important environment variables that the user wants to dynamically configure are described in Table 4.2. The prototype comes with an environment variable file so the user can fill in the values for all the keys. Omitted from Table 4.2 are the environment variables related to the AWS services the user has to set up before being able to use the prototype.

4.4 Low-Level Description of how Duplicraft Works

Figure 4.3 shows the successful deployment and connection switch processes in more detail starting from the state in which the client is already connected to the proxy. Beforehand,

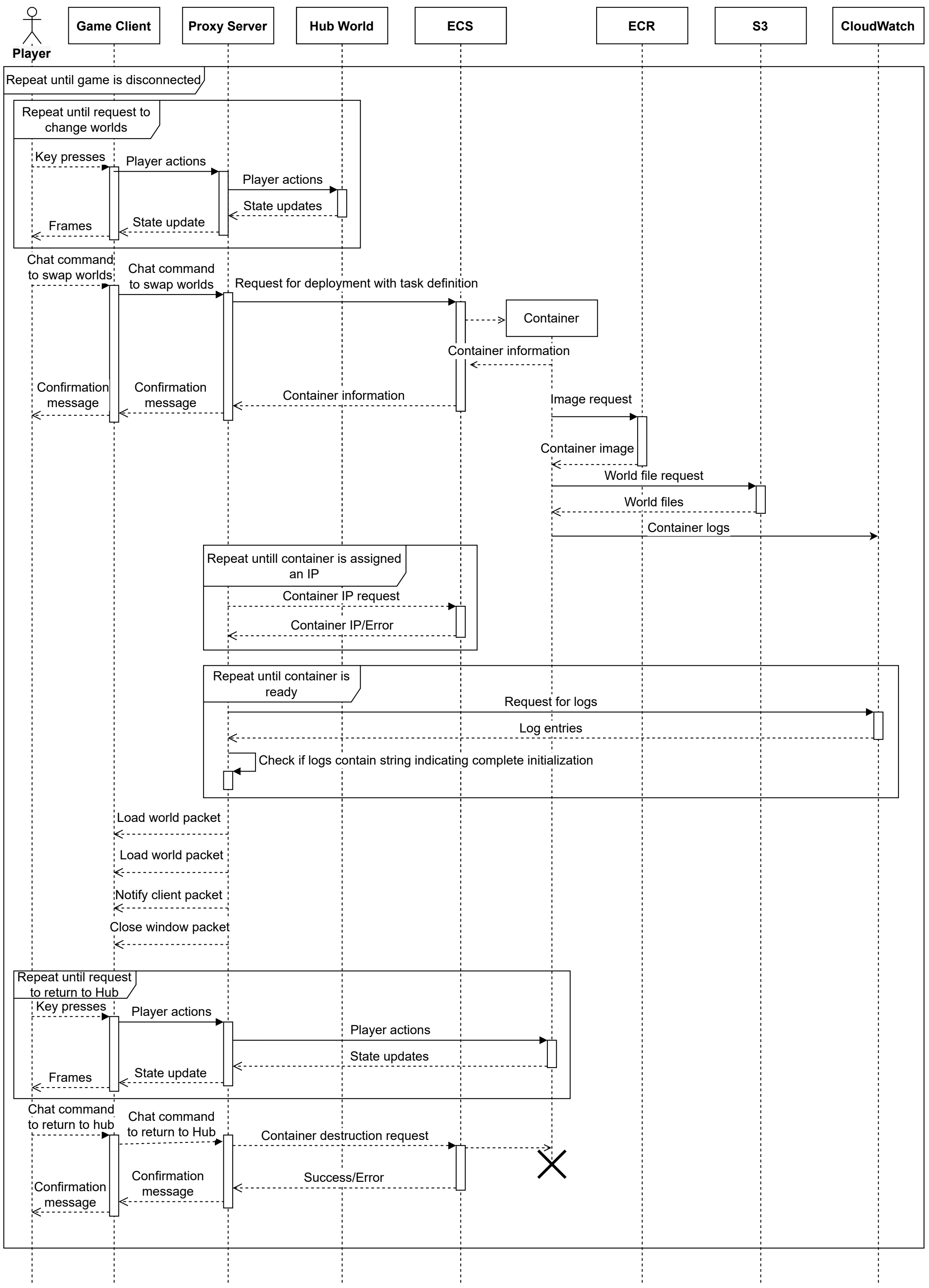

Figure 4.3: Sequence diagram of the Duplicraft container deployment and connection switch process.

the proxy initializes a connection with the Hub World and awaits a client connection. The game packets then get forwarded as if the client is sending them directly to the server by setting up 2 session listeners which listen on the TCP sessions between the Game Client and Proxy and between the Proxy and Hub World.

Once the user sends a chat message containing the text "/swap", the proxy server tells ECS to deploy a container based on the configuration specified in the previous section. Once ECS has confirmation that the container is being initialized, the Proxy Server gets a confirmation message in return and sends the Game Client a confirmation message to display to the user. We chose to use chat messages to indicate these interactions because users can send them from the client in any world, and the Proxy server already has the capability to intercept chat messages and analyze their contents.

For the container image, we use a docker image of a Minecraft server published by a user named itzg on DockerHub (19), because it was made to be highly configurable through environment variables which is something we can deploy a container with using our Task Definition. Relevant configurations that this container image allows us to specify are, for example, the amount of memory the Minecraft server can use, allowing bots to play on the server so we can evaluate the prototype and the ability to specify a world URI which it will automatically download and use as the Minecraft server's world. This allows us to retrieve the worlds we store in a public S3 bucket by specifying their respective URIs in the environment variable.

To get the IP of the running task, the Proxy gets the network interface from the container details. It then repeatedly sends requests to get information about this network interface until it eventually returns an IP address. If there is no IP within a configurable amount of seconds, the task is terminated. Then, for another configurable amount of seconds, the server polls the CloudWatch logs each second to see if there is a log entry that matches Minecraft's default message to indicate server initialization has been completed successfully, if it takes longer than the configured amount of seconds, the task is terminated. We detail this choice and mention an alternative in §4.5.

However, only starting a new connection, removing the old session listeners and replacing them with new ones causes problems, such as the Game Client getting stuck on the *loading world* screen and the Game Client displaying the container server's world, but handling the player collision as if it were on the Hub World. To fix the disparity between the display and collision handling, we manually send two packets to the Game Client to tell the client to reload the world. We send two packets because the Minecraft protocol (20) specifies issues might occur if you try to send a respawn packet to the same dimension.

To fix the issue of getting stuck on the *loading world* screen, we send the Game Client a packet that forcefully opens a window on the Game client, then immediately follow it up with a packet that forces it to close, which forces the Game Client out of the *loading world* screen.

After that the packets are forwarded in the same way they were to the Hub World. Once a player sends a chat message containing the text "/hub", the active container is destroyed and the Game client's packets are forwarded back to the Hub World in the same way they were when the connection was switched to the container.

4.5 Implementation Alternatives

To improve performance for every individual world, we could preemptively define a set of Task Definitions on ECS with a predetermined amount of resources to balance out operational costs and server performance. However, given that one of this Duplicraft's main goals is to take away from the currently required preemptive configuration for existing solutions, we want to make a dynamically configurable prototype. This leads us to believe the better alternative is to template a Task Definition and update it for every to-be-deployed container.

To reduce the time it takes for a server to be deployed, we can include the world and server jar in the container image. This has the advantage of not having to download the resources as an instance is deployed. However, we choose not to do it this way, because it would require us to create and store a separate container image for each world, increasing both cost and management overhead for the system user. By allowing the server jar and world to be dynamically configurable we increase the time it takes for the server to be deployed. However, because one of our main goals with Duplicraft is to reduce management overhead for the server operator we opt not to implement it this way. Our experiment in §5.3.3 shows that the downloading of the server jar and world files are only a fraction of our deployment times, convincing us that this is the correct choice.

To see if an instance has finished deploying we currently repeatedly analyze the container's logs to find a predefined sentence to indicate the world has finished loading. If such a sentence were not consistent, or we run into issues such as a rate limit on retrieving logs from AWS, we can alternatively attempt to connect through the server using a TCP client and wait for the connection to be accepted. Both options result in similarly accurate times at which the instance is accepting connections, and can thus be used interchangeably, but we opt to poll the logs, because we can also use the data from the logs to perform experiments and analyze our server deployment, making it useful in Chapter 5.

A final alternative is to utilize a different cloud service provider. This can impact our system in various ways. If a cloud service provider has more locations to host containers, it could mean that switching to that provider allows for Duplicraft to have lower latency during play. Other cloud providers could also charge different amounts for similar resources, meaning that the cost would vary depending on which provider Duplicraft uses. Every cloud service provider that allows for the deployment of containers on-demand can support Duplicraft, but we choose to use AWS for the reasons specified in §4.2.

5

Evaluation

In this section, we explain how we evaluate our Duplicraft prototype and discuss its capabilities and characteristics to decide whether it meets our system requirements and if it would be a viable alternative to existing services.

5.1 Experimental Environment

To run our experiments, we deploy each container server instance using Duplicraft as described in §4.1. These container server instances are Minecraft servers for the game version 1.12.2, which allows us to experiment using the realistic workloads described in §5.2.1. These containers are deployed for every experiment with a flat, unexplored world with no structures or otherwise natural terrain generation. We use these worlds as Duplicraft is intended to run minigame worlds, which tend to be small in size ($<$ 10MB), which we determined by downloading several of the most popular Minecraft minigame worlds and looking at their sizes.

We deploy our containers running Minecraft instances on AWS and vary the amount of resources of each deployed container per experiment. All other tools and programs are deployed locally on a home desktop computer setup, described in Appendix A. All containers are deployed on an AWS region geographically closest to Amsterdam, which in this case is the region *eu-central-1* hosted in Frankfurt.

5.2 Experiment Design

Table 5.1 shows the design of our experiments. To evaluate Duplicraft, we determine several important metrics which give us insight into whether our system is viable for real-world use or not.

	Metric	Server type	Workload		Tools	Rep.
			Bots	World		
§5.3.1	performance	Spigot	0-50	default	Duplicraft Yardstick	1
§5.3.2	latency	Official	16	flat	Duplicraft Yardstick	1
§5.3.3	cold start	Official	-	flat	Duplicraft	30
§5.3.4	resource efficiency	Official	0-50	default	Duplicraft Yardstick	1

Table 5.1: Overview of experiments.

To evaluate performance, we wish to measure the amount of time it takes the Minecraft instance to process a game update. A key feature of Duplicraft is that a user can configure the amount of resources a container is deployed with for each deployment. Because the cost to the user scales with the amount of resources used, we wish to test the performance of Duplicraft instances under several different configurations. We can measure this by deploying instances with different amounts of resources and measuring the amount of time it takes to process a game update under the varying workloads deployed by Yardstick. This allows us to determine how well the tested instance configuration performs under several workloads and if our system meets performance constraints. We use a popular alternative to a Minecraft server that allows plugins, called Spigot, to use a plugin called TabTPS (21). This plugin displays the milliseconds per tick (MSPT), a measurement of how long it takes the server to process a game update, to the user several times per second with a precision of 2 decimals. To generate our data we use Duplicraft to deploy Minecraft server instances using the Spigot jar and the TabTPS plugin under different amounts of resources. We then connect to the instance using the proxy so we can intercept messages from the server. Once we enable TabTPS to display the MSPT to our connection, we intercept the packets detailing the MSPT alongside packets from the server indicating a player has joined. Our modified Duplicraft then writes the amount of resources, the amount of players currently connected and the MSPT to a file each time the MSPT is updated to collect the data. After measuring for a minute with no bots connected, we start deploying 5 bots each minute using Yardstick to simulate a realistic workload, up to 50 bots, because minigames rarely have 50 players or more. To make the experiment reproducible, we specify that these bots are walking around in a bounded area of 32 blocks around coordinates (200, 240) in the seed 7030672831634543076 generated using default terrain generation. We can then plot this data as a line graph to see how each configuration performs under the different workloads, and conclude which configurations meet our performance requirements under specific workloads.

We also evaluate the latency of our system. Most minigames depend on real-time player interactions, which leads to player enjoyment being impacted heavily by high latency. To gain insight into the latency between a player and an instance deployed by Duplicraft, we deploy several instances with different resources. We then use Yardstick, a tool described in §5.2.1, to deploy a realistic workload on the instance and measure the latency between two players under this workload by starting a timer, sending the server a message of a world alteration from the first player, then stopping the timer once the second player gets notified of the alteration. This data can then be displayed in a box plot, which we can use to determine whether Duplicraft meets latency requirements. It also allows us to reason about the cause of the latency measured in our experiment. To simulate a realistic workload we deploy 16 bots to the server, 14 of which are walking around to simulate realistic player actions and the other 2 bots measure the latency. Yardstick measures latency by having one of the bots repeatedly place or break a block and start a timer, the other bot then stops the timer once it receives a packet indicating the world has changed at the location of the recently placed or broken block. We chose the number of 16 bots as we wish to simulate a realistic setup, which in this case would be a minigame with more than 10 players. Among the most popular minigames, some are played in teams, and 16 is divisible into several different team arrangements with an equal amount of players. Each run of the experiment lasted 5 minutes and used an empty flat world as it was required for the use of Yardstick's latency experiment. To make the experiment reproducible, we specify the seed used to generate the flat world is -2907575813890716159 and the players walk in a bounded area of 32 blocks around coordinates (110, -410).

Moreover, we evaluate the cold start times of Duplicraft. Loading times can negatively impact player experience, making it intuitive to gain insight into the loading times of Duplicraft. We measure the loading time of Duplicraft as the time between a player requesting an instance to be deployed and Duplicraft being able to connect the player to the instance. We do this by modifying Duplicraft to start a timer as soon as the player sends a message to the proxy indicating the desire to swap worlds, then ending the timer as soon as Duplicraft reads from the instance's logs that the instance is accepting connections, repeating this process 30 times to determine the variance in loading times. With this data, we can generate a box plot of the total loading times to get an indication of how long a player must wait. However, total loading times do not help us understand what segments of our loading times can be reduced. To give better insight, analyze the logs of each cold start and determine several segments, such as how long it takes to download the jar and how long it takes to configure the server, then produce a stacked bar chart showing the amount of time each segment takes. With this bar chart, we will have an insight into which segments of Duplicraft loading times scale with resources and which do not.

Finally, we evaluate the resource efficiency of Duplicraft instances. To make the most use of the pay-per-use model provided by serverless computing we ideally only allocate as many resources as we will use. To determine the resource efficiency of Duplicraft we will use a highly similar setup to the performance experiment, using the same world and intervals of bot connections with the only change being that we deploy bots in batches of 10 using the same behaviour pattern. We then use the container insights provided by AWS to generate line graphs of the CPU and Memory usage, choosing these two metrics as they are the core variables we change between runs of the experiment. By running the experiment with varying amounts of resources we can observe the changes in resource efficiency as we allocate more resources. We can then cross-reference the data of the resource usage with each individual instance's logs to determine at what point bots started being deployed every minute and mark this spot on the graph to visualize how the increasing workload impacts the resource efficiency of a instance. This graph will give insight into resource efficiency under our workload.

5.2.1 Workloads

Table 5.1 already shows a few key details of the workloads used to run our experiments. To run our real-world experiments we want to simulate realistic workloads to evaluate Duplicraft. Van der Sar, et al. implemented a benchmarking tool for Minecraft-like servers named Yardstick (14), which tests the scalability and performance of these servers. The tool deploys realistic workloads by emulating players on a Minecraft-like server and has the ability to capture performance-related metrics. Yardstick also has options for customization of the deployed workload, allowing the user to simulate specific types of realistic interactions such as players sending chat commands and moving around. Once deployed, Yardstick can measure system- and application-level performance metrics by monitoring the server's host machine and derive service-level metrics, such as the number of messages per second or frequency of game loop updates, based on collected data.

To simulate realistic workloads in our experiments, we deploy bots that walk in a bounded area around a given coordinate to simulate players maneuvering around the world. We do this so we can measure the performance of the containers in terms of latency and performance during play. This workload varies per experiment to help us measure if our system scales up to 50 players as specified in our system design requirements detailed in §3.1.

Part of our workload is the world we play in. We downloaded several of the most popular minigame worlds and concluded that minigame worlds consistently take up less than 10 MB of storage, occasionally even less than 1 MB. For these reasons we deploy our servers with either an unexplored flat world, which is required for Yardstick's latency experiments, or with a world using Minecraft's default world generation with the spawning of animals enabled, to simulate a more realistic workload as flat worlds tend to consist of only 3 layers of blocks with no further world simulations such as growing trees or moving entities. However, this world only provides a realistic workload for minigames with a limited world where players do not actively alter the world (e.g. Parkour in Minecraft). Determining and accurately simulating a realistic workload for minigames is challenging, as there is a large variety of available minigames with vastly different workloads, some of which support many in-game particle effects and require multiple players to alter the world at once among more intricacies.

5.2.2 Metrics

To evaluate Duplicraft, we wish to collect four metrics to determine its viability:

Metric 1: Performance.

Minecraft-like games scale poorly because they have to handle large amounts of data for game state updates. For this reason, we measure the performance of a Minecraft server as the amount of time it takes to process a game tick.

Metric 2: Latency.

Latency in Minecraft-like games is defined as the amount of time between a player interacting with a server, and other players being notified of that change. For this reason, we test latency between two players as discussed §5.2.

Metric 3: Cold start time.

Cold start describes the overhead caused by a container having to be freshly set up. In our case, this is the delay between a user informing the server of wanting to switch worlds and the world being ready for connections.

Metric 4: Resource Efficiency.

A large benefit of serverless computing is the pay-per-use model. To measure whether a Duplicraft user really pays for the resources they use, we measure what percentage of the allocated resources Duplicraft instances actually use during play.

5.3 Experiment Results

From the experiments we gather 4 main findings:

MF1: (Section 5.3.1) Duplicraft instances meet performance requirements, supporting at least 50 players walking around in a non-flat world without interacting with the world on its minimal configuration. The only time the performance requirement is violated is on the lowest configuration when new players connect to the instance, which causes a temporary drop in performance.

MF2: (Section 5.3.2) Duplicraft on average meets latency requirements (i.e. less than 50 milliseconds of latency) under the low workloads of the experiment with container configurations using as little as 1 vCPU and 2 GB of memory. Even so, each configuration has outliers above 50 milliseconds which could impact the quality of experience.

MF3: (Section 5.3.3) A container with 2 vCPU and 4 GB of memory is an intuitive minimum based on our experiment results, however the average time of the cold start is higher than what a study (22) shows to be acceptable. We cannot scale this cold start time down far enough to be deemed acceptable by the study, but we can argue for the time to be acceptable as Duplicraft allows for the player to move their character in the Hub World as they wait.

MF4: (Sections 5.3.3, 5.3.1 and 5.3.4) Different tasks scale differently with the amount of resources provided, dynamically changing the amount of resources during run-time could significantly decrease cost and improve resource efficiency whilst keeping desirable properties such as low cold start times, low latency and high performance.

5.3.1 Performance

Figure 5.1 shows the results of our performance experiment. We measure the number of milliseconds different container configurations take to process game ticks under different workloads. We wish to support 50 players, and game updates are processed by the client

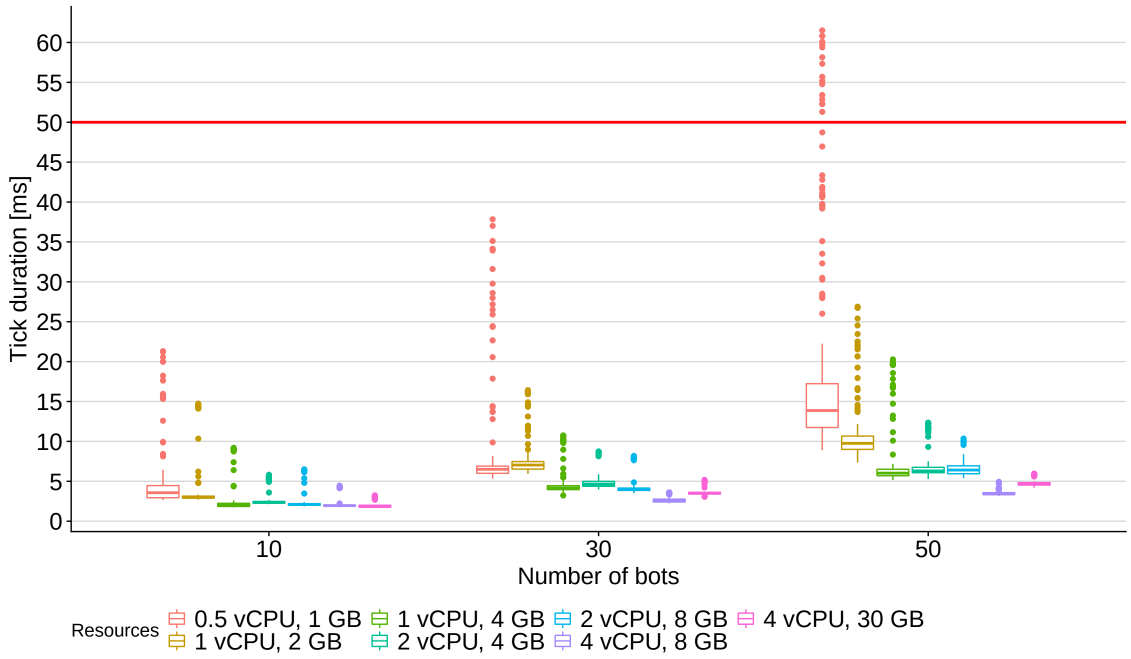

Figure 5.1: Grouped box plot of measured time it takes a server to process a game tick with bots walking around (resources are ordered from left to right, lowest to highest).

20 times per second. This means that we wish to keep the milliseconds per tick (MSPT) of the server below 50. The unmodified official Minecraft server for version 1.12.2 only measures up to 20 TPS, giving us less detailed insight into the performance of the server. Spigot allows us to install a plugin called TabTPS on the server which displays the server's MSPT to the user by sending packets we can intercept and analyze.

In the figure we can see that Duplicraft on average performs at an MSPT below 20 on every single configuration, where the amount of outliers scales down as the amount of provided resources scales up. What is unexpected is that even on the low-resources configuration of 0.5 vCPU and 1 GB of memory the instance on average performs well below half the required MSPT, the required amount being 50 MSPT. This is unexpected because 0.5 vCPU and 1 GB of memory is significantly less than the recommended amount of resources to run a Minecraft server for more than 10 players. This means that even on the lowest possible configuration, Duplicraft instances are able to meet performance requirements on average with at least 50 players. It is noteworthy that our highest resource configuration does not consistently have the best performance, which indicates that allocating excess memory does not always increase performance. We display measurements with all intervals of 5 bots in Appendix B.

To confirm these unexpected results, we ran two sanity checks that we show the results of and elaborate upon in Appendix C. In short, the sanity checks seem to confirm the consistent performance of these instances and lead us to believe the consistent performance is a result of the low workload produced by players only walking, without altering terrain or loading in new parts of the world.

However, for the player experience it is important not only to measure the mean values but also to look at the performance drops indicated by the outlying MSPT values for each configuration. Here we can see that yet again Duplicraft instances consistently stay below the required 50 MSPT with the exception of our lowest resource instance, which has performance drops that would violate the system requirements.

We can see in the figure that the performance of each instance has low variance, except for the lowest resource configuration at 50 bots. However, each instance has outliers above the mean, where the difference between the outliers and the mean scales down as you scale up resources. After analyzing the data we conclude that these drops in performance happen when a new group of bots is connected to the instance, leading us to believe that the new group of connections within a short amount of time causes a temporary drop in performance. This reaffirms the idea that the performance during play is consistent, and performance drops only occur as a result of certain events, such as multiple players connecting to the server within a short period.

From this experiment we can conclude that Duplicraft instances perform well within our system's performance requirements under a small workload with at least 50 players. Even on the lowest configuration the requirements are only violated during the performance drop when the bots connect to the instance.

5.3.2 Latency

Figure 5.2 shows the results of our experiment where we measured the delay between a player placing a block and another player receiving a network packet indicating the change in different server configurations. We deploy the servers by using Duplicraft with different configurations.

In the figure we can see that there is little variance in the average and median latency between the different container configurations, indicating that the amount of resources a container has does not affect the latency in significant ways under the conditions of our experiment. The size of the boxes is also similar for all configurations, reaffirming that the amount of resources has little impact on the latency. However, the latency is not consistent and has outliers that could impact player experience, which becomes much more extreme

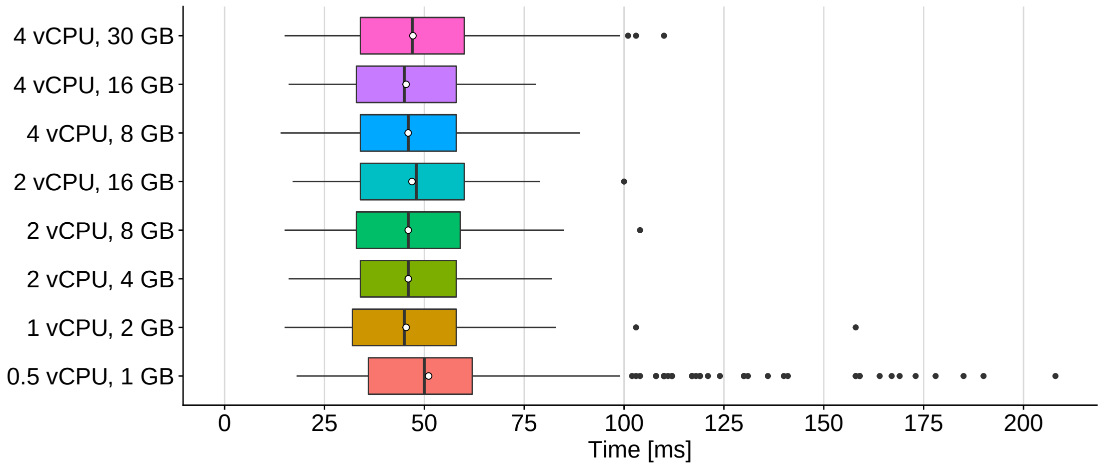

Figure 5.2: Latency between two players of a block being changed under a load of 16 players. Lower is better.

with the worst configuration where the server cannot keep up with the workload. These outliers are the most extreme in the configuration with the maximum amount of resources allocated for the container, and the second most extreme in the container with the least amount of resources. Thus we cannot conclude a correlation between allocated container resources and outliers in latency. There being no significant correlation indicates that latency is mostly dependent on other factors, an intuitive one being network delays based on where in the world the container is deployed relative to the user connecting to it.

We can derive from the figure that the mean and average latency for containers of 1 vCPU and up lies between 45 and 50 milliseconds. The expected latency consists of 3 main segments, the delay of the packet being sent from our testing machine and the server hosted in Frankfurt and is then sent back, which we estimate to be no larger than 10 milliseconds. The second segment is dependent on when the next game state update is sent out by the server. Minecraft servers update 20 times per second, meaning that every 50 milliseconds a game state update has to be sent out. In an ideal case, our packet arrives the exact moment before a game update is processed, and is instantly processed and sent back. In the worst case, our packet is received by the server exactly when the previous game tick has been processed, implying we have to wait 50 milliseconds for the next game state update to be processed and sent out. If we take the average of these two extremes, we determine the expected delay based on tick processing intervals to be 25 milliseconds. The final segment is the time it takes the server to process a tick, which can be variable dependent on the server's workload and resources, among other variables. With the varying amount of resources used in this experiment, we try to show this time to

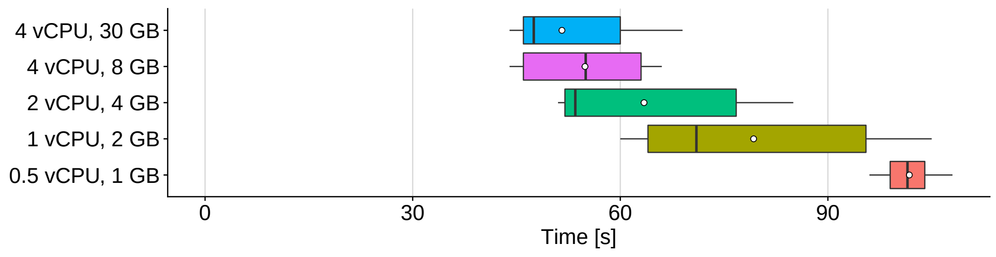

Figure 5.3: Cold start times with different amounts of container resources.

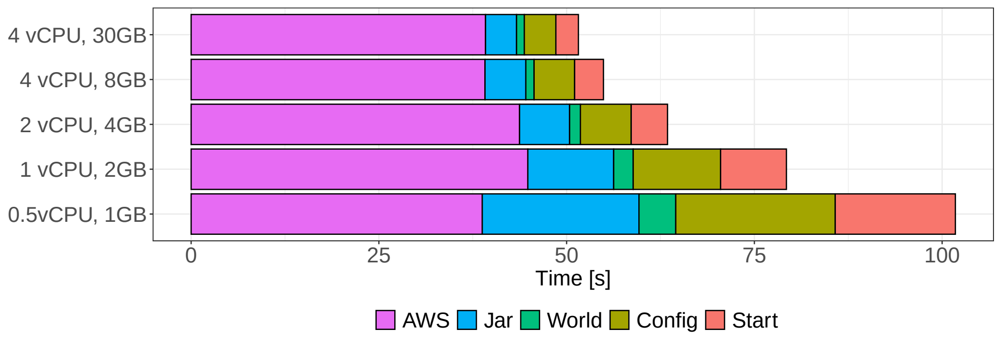

Figure 5.4: Overview of the amount of time each task of a cold start takes on average.

be variable, which is very apparent in the outliers of our container with the least resources, where ticks might take significantly longer to process causing large delays. However, due to our workloads being small the number of outliers is greatly reduced from our configurations of one or more vCPU. We can also note that, with the exception of the worst configuration, all boxes have similar outliers on the right and left sides, which can be explained by the delay based on tick intervals. Based on the numbers calculated in this paragraph, we can estimate that on average it takes 15 milliseconds for our packet to get processed and sent back to us, which does not scale with resources.

With the exception of the worst configuration, the average and median latency consistently lies below 50 milliseconds, which meets system requirement 2, specified in §3.1. However, there is variability causing the latency to go beyond 50 milliseconds, which could impact player experience at times. With this consistency across different resources, we can argue that Duplicraft meets latency requirements under low workloads with resources as low as 1 vCPU and 2 GB of memory.

5.3.3 Cold Start

Figure 5.3 shows the result of our cold start experiment and Figure 5.4 how much time each segment takes during the cold start. We measure cold start as the delay between the moment the proxy receives the user's chat message which indicates the request to swap worlds and the moment the proxy knows the container is ready to be connected to. These two points measure the time the player has to wait before being able to play on the instance. We split a cold start into several segments to analyze what parts of our system scale well. In the second figure, we show the time it takes AWS to provision a container as AWS, the time it takes our container to download the server jar and world as Jar and World respectively, the time it takes to configure the server as Config, and the time it takes the server to launch the world once everything is set up as Start. Since we are hosting the proxy on the client machine, there are no significant network overheads between the client sending the chat command packet and the proxy receiving it, which makes it intuitive to measure the moment of receiving that packet in the proxy as the moment to start measuring the delay caused by the cold start. The lowest amount of vCPU we test with is half a vCPU and 1 GB of memory, which is very little for an MVE game server, but as Fargate only allows certain combinations of vCPU units and memory, this allows us to scale up by doubling the amount of vCPUs and Memory repeatedly. Once we reach the maximum amount of vCPUs available by Fargate (8 GB), we test one more time with the maximum amount of available memory (30 GB). Each configuration has been sampled 30 times, with a delay of three minutes in between each deployment to try to avoid using a container that already exists and has not had its resources deallocated yet, which would not be a cold start. We generate the samples by having a modified version of Duplicraft automatically deploy and destroy Instances, then we analyze the logs on AWS CloudWatch to determine how much time each segment took to complete.

The first figure shows that the cold start time scales with the amount of available resources both in consistency and mean time (indicated by the white dots), which is consistent with other people's findings (23). However, there are a few noteworthy things to observe about our data. The furthest outlier for each of our configurations is always towards the right side of the box, meaning that cold starts can take considerably longer than the expected mean time of each configuration. The median, indicated by the black vertical line, in three configurations lies far to the left of the mean, showing that in these cases the amount of time a cold start takes is generally either of two extremes, rather than any given time between the minimum and maximum being equally likely to happen. We speculated that it indicates that some of the cold starts are not truly cold starts, where resources are being reused between runs of the experiment. We tried to mitigate this possibility by generating 10 more samples for the most extreme case of this phenomenon in our data, the configuration of 2 vCPU combined with 4 GB of memory, by waiting 30 minutes between each deployment. However, the results were highly similar, with the cold start time being either close to 50 or 80 seconds, disproving our speculation. The configuration which uses 1GB of memory also stands out, because it is vastly more consistent than the others. We were unable to determine exactly why this is the case.

The second figure helps us analyze the cold start time further, where we note that every single segment in our control scales with the amount of resources we provide. This does mean that with the state-of-the-art technology that AWS provides us with, it is still a point of concern for our system because we cannot decrease the cold start time significantly by improving the server launching procedure. We can note that our system will not be able to consistently scale lower than 50 seconds regardless of the amount of resources we start a container with. A study by Liu et al. (22) shows that more than 50 percent of participants said it was unacceptable for a level to take longer than 50 seconds to load, which indicates that our system's load times are unacceptably long. However, this number is an average across several types of games with different loading screens. They argue that certain visual stimuli, such as progress bars, can make longer loading times more acceptable. In Duplicraft, a player can still move around the Hub World as a container is being deployed, meaning that loading times in Duplicraft are arguably acceptable. These loading times could further be made acceptable by having potential activities for the player in the Hub World to participate in as they wait for a container to be deployed.

If we combine these results with the results gathered in §5.3.1 and §5.3.2 we can make the observation that, given the workload that we ran our experiments with, the amount of resources allocated for the container should not be static. As pay-per-use is an important benefit of serverless systems, we wish to avoid using more resources than we need. A desirable situation would be to initially allocate a lot of resources to reduce cold start times, and then allocate just enough resources during play to have stable performance and latency during play, meaning we should scale down the amount of resources after the instance has finished initializing.

Even though our data has these inconsistencies and exceptions, we can conclude that the mean cold start time decreases as you scale the amount of provisioned resources up. However, it does not scale linearly with the amount of resources provided and both of our top two configurations have the same minimal cold start time (44 seconds) and a similar

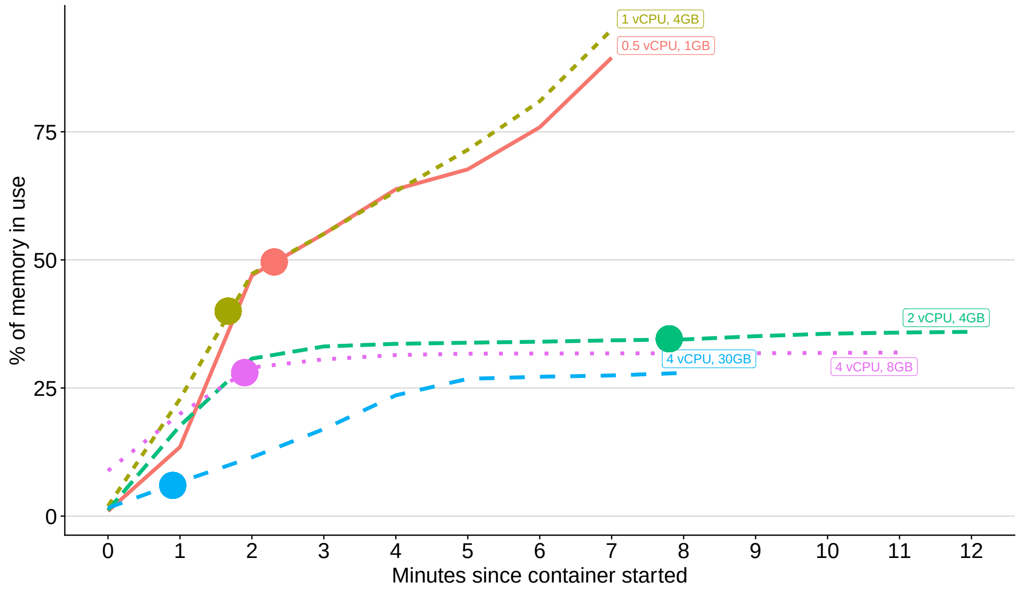

Figure 5.5: Memory usage of each Instance over time, the dots indicate when batches of 10 players started joining each minute (up to 50).

range of cold start times. We can also conclude that the amount of variability in these cold start times decreases as you scale up the amount of provisioned resources, with the exception of the lowest amount of resources. These cold start times are long in comparison to the existing system which instantly connects you to another instance, but are often able to be masked because online minigames require the player requesting an instance to be deployed to wait for others to join the minigame. The cold start is considered the main cost of our system to relieve the server operator of a large amount of management overhead. We can see in the second figure that from our third best configuration onward the cold start time starts to scale significantly less efficiently with the amount of extra resources we provide as we double the value, but see less than half of the performance gains that we see going from the second worst configuration to the third worst. This makes 2 vCPU and 4 GB of memory an intuitive minimum to deploy our containers with.

5.3.4 Resource Efficiency

Figures 5.5 and 5.6 Show the results of our resource efficiency experiment. For this experiment, we chose 5 of the instance configurations used in our performance experiment, namely the minimum, maximum and three configurations in between. AWS CloudWatch Container Insights only provided us with the average resource usage as the average over a minute, thus it is difficult to draw concrete solutions from this data. However, we can

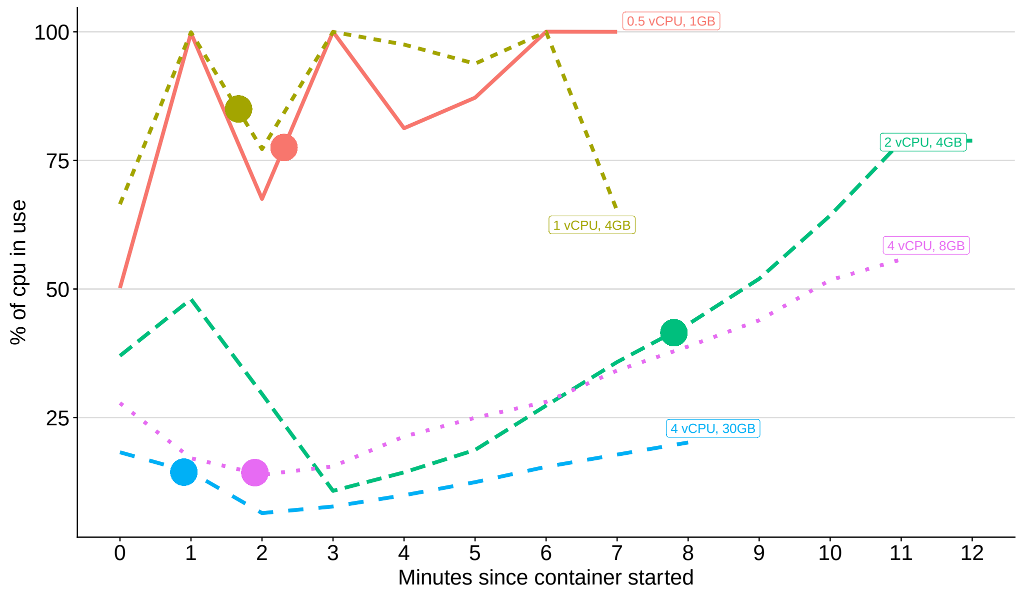

Figure 5.6: CPU usage of each Instance over time, the dots indicate when batches of 10 players started joining each minute (up to 50).

still use it for insight and potential future research. For example, it is notable that the moment bots start being deployed seemingly has no significant impact on either figure for any configuration. We have no clear explanation for why this is the case, but it could be interesting to see how the establishing of connections impacts CPU or memory usage. This level of insight is currently masked by only having the average usage over a full minute when connecting a new batch of bots generally takes place over two to three seconds.

Both figures show a clear division between the lowest resource configurations and the highest ones. We can see that in both cases the configurations with the lower resources are more resource efficient, indicating that during play under our workload, these are better for resource efficiency but are not likely to scale well beyond this point. The main anomaly that stands out is the 1 vCPU 4GB instance dropping its CPU usage by around 30% at the end. This anomaly was caused by an issue where the Yardstick instance would disconnect a portion of the bots at the end of the experiment, which only happened at this configuration and persisted in an attempted re-run of the experiment. For this reason we believe the actual CPU usage with 50 connected bots to be higher, but the average accounting for a period of time where many bots were disconnected. We can also note that both configurations with 4GB of memory have vastly different memory usage, for which we could not find a proper explanation.

A notable result is that the memory usage in the configurations with higher resources remains stable and the resource usage gradually goes up as bots are deployed. Contrary to the sporadic results of the lowest two configurations resource-wise, this meets the expectations of our setup. As under our workload the bots are not exploring the world or doing memory-intensive actions, we expect the memory usage to remain stable whilst more player actions need to be processed and thus the CPU usage to go up. Even though it is difficult to draw conclusive results from this data, we can use this to support that performance stabilizes the more resources are allocated as was shown in Section 5.3.1.

These results also support the idea that dynamic resource allocation at runtime would be a highly beneficial trait for a system like Duplicraft. If we were to deploy a container with 2 vCPU and 4GB of memory as suggested in Section 5.3.3, we would have poor memory and CPU usage under our workload. If we were able to dynamically scale resources at runtime, we could far more efficiently use resources and best make use of the pay-per-use model provided by serverless computing.

6

Related Work

There is very limited research on the serverless potential of Minecraft-like games. Individual projects for serverless hosting of multiplayer instances, such as Minecraft-OnDemand (24), exist as a proof-of-concept without fully exploring the viability of serverless systems for MVEs. A main work in this niche is Donkervliet, Trivedi, and Iosup (HotCloud-Perf, 2020) (25), which proposes a redesign of MVEs in which they become scalable serverless systems and resource management and scheduling is managed by the cloud provider. However, this work does not implement nor test the viability of such a system in a real-world scenario.

However, outside of gaming research in serverless systems is thriving. Promising uses of serverless technology are being discovered in many fields of research. Examples of such fields are distributed machine learning (26), video processing (27), big data processing (28) and many more (29).

Container services are of great importance to this research. AWS Fargate was the container orchestrator of choice, but many alternatives exist. A notable one that went unused in this work is the open-source container orchestration system Kubernetes (30, 31). Kubernetes provides a framework with many benefits to the user, important once being the scaling of containerized applications and resilience. Other popular alternatives are Microsoft's Azure Container Instances(32) and Google's Cloud Run (33), which provide similar services to AWS Fargate.

7

Conclusion

Minecraft-like games are among the most popular online video games and continue to grow, which means that related systems need to grow with it to support the needs of both the players and server operators. However, integrating many different modern technologies into a single system is not a trivial task. It requires us to carefully identify the existing system components, understand what components we need to add to seamlessly string together multiple different existing services and design each interaction carefully around them to ensure it functions as expected. From this design we identify which system components are critical for evaluating the performance and cost of our system, resulting in a functional prototype that takes in many new considerations to solve unforeseen implementation-specific issues. From this prototype we are able to run several experiments and analyze the cost of our system, then determine it has qualities that improved upon existing systems, such as fine-grained payment and less management overhead. We also acknowledge that the system has downsides, namely the cold start times of containers and inconsistencies in latency. An important finding which resulted from our experiments is that there are great benefits that would result if resource allocation is done at runtime, as different tasks scale vastly differently with the amount of resources provided given our experiment configuration.

7.1 Answering Research Questions

In §1.2 we distinguish three research questions we set out to answer.

RQ1: How to design a system for serverless multiplayer minigames in MVEs?

Through the AtLarge design framework, we learn that designs go through many iterations before becoming something that can be implemented in a real environment.

It takes a deep understanding of the system you are working with and its components to be able to coherently put them together. When you have not thought about every system interaction, a design fails. This is of significant importance when designing a serverless system in which many components often hosted on different platforms each having their own infrastructure will work together to achieve a single goal. So thorough consideration and repeated feedback are key for an iterative design to be robust. After many iterations, we are able to present the design shown in §3.2.

RQ2: How to implement a prototype of such a design?

It is important to consider how to build up a prototype based on a design. Based on the time constraints of the project it is important to identify critical design components which have to be implemented to be able to evaluate your system. In our case, that implies we have to either modify or leave out several components to implement a functional prototype, which allows us to run experiments to evaluate our prototype. Based on the ranking of the most critical system components, we implement them ordered from most to least critical, leaving out only the components which implement functionalities that do not significantly affect how we evaluate our system, such as the container manager. During the implementation of a prototype, we also run into several unforeseen issues, such as switching worlds seamlessly. Preparing for every issue is nearly impossible, so time management is very important to ensure a functional prototype can be created.

RQ3: How to evaluate the design, through real-world experiments using the prototype?

Once the prototype is implemented and its limitations and capabilities are known, we experiment on the system. To do so, it is highly important to identify the key metrics which set your system apart from others or measure functionality. In our case, we identify the four key metrics discussed in §5.2.2, because they measure if our system performs according to our system requirements, and show the advantages and disadvantages our system has compared to other services and existing systems. After settling on the metrics to measure, we select the right tools to be able to run experiments and evaluate their outcomes. Once the experiments are finished, it is vital to analyze and generally visualize the data, because it gives deeper insight into the results of the experiments.

7.2 Limitations and Future Work

There are still several flaws with our system, mainly in the prototype. The first suggested improvement is to implement the missing components which did not transfer over from the design into the prototype, with a modified game instance, container manager and managing the interactions with the cloud service provider through the Hub World. This would allow for a more complete prototype where several different players would be redirected to the same container.

A second suggestion is to test more realistic workloads to accurately simulate several different minigames being played, including more accurately simulating player actions to have a more realistic workload than only walking around. This allows us to better understand the performance and capabilities of such a system.

A third improvement is to experiment more thoroughly with latency. The mean latency of Duplicraft instances is within our system requirement, but there are inconsistencies that go above the requirement. More research on how to improve the latency during play while using Duplicraft will give us more insight on ways to improve such a system. Examples would be to experiment with deployment in different regions or trying to connect from different networks. The way we experiment with latency in this work is also not conclusive on which parameters affect latency the most for our system, so thorough latency experiments give more insight on how to improve such a system.

A fourth point of interest is to develop a feature for open-source serverless platforms, such as OpenWhisk, to dynamically scale resources at runtime. This would allow further research into our main finding stating that certain tasks of the instance scale much better than resources than others.

The final suggestion is to test the system with multiple cloud service providers and see if there are significant changes in the four core metrics we evaluated, namely in the cold start time, because we show that the scaling capabilities of our system's cold start times are mainly influenced by AWS. If there is a significant enough difference, it is worthwhile to transfer the system over to another cloud service provider.

References

[1] Tom Wijman. **The Games Market and Beyond in 2021: The Year in Numbers | Newzoo**. https://newzoo.com/insights/articles/the-games-market-in-2021-the-year-in-numbers-esports-cloud-gaming/, December 2021. (Accessed on 7 February 2022). 1

[2] J. Clement. **Online gaming - statistics & facts | Statista**. https://www.statista.com/topics/1551/online-gaming/#dossierKeyfigures, May 2021. (Accessed on 7 February 2022). 1

[3] Chris Bailey, Elaine Pearson, Voula Gkatzidou, and Steve Green. **Using Video Games to Develop Social, Collaborative and Communication Skills**. June 2006. 1

[4] Jordan Sirani. **Top 10 Best-Selling Video Games of All Time - IGN**. https://www.ign.com/articles/2019/04/19/top-10-best-selling-video-games-of-all-time. (Accessed on 8 February 2022). 1

[5] Ashwin Bharambe, John R. Douceur, Jacob R. Lorch, Thomas Moscibroda, Jeffrey Pang, Srinivasan Seshan, and Xinyu Zhuang. **Donnybrook: Enabling Large-Scale, High-Speed, Peer-to-Peer Games**. *SIGCOMM Comput. Commun. Rev.*, **38**(4), August 2008. 1

[6] Ashwin Bharambe, Jeffrey Pang, and Srinivasan Seshan. **Colyseus: A Distributed Architecture for Online Multiplayer Games.** January 2006. 1

[7] Julien Gascon-Samson, Jörg Kienzle, and Bettina Kemme. **DynFilter: Limiting bandwidth of online games using adaptive pub/sub message filtering**. In *2015 International Workshop on Network and Systems Support for Games (NetGames)*, 2015. 1

[8] 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)*, 2013. 1

[9] Edward Hays. **Most popular active Minecraft servers of 2021**. `https://www.sportskeeda.com/minecraft/popular-active-minecraft-servers`, June 2021. (Accessed on 06/14/2022). 2

[10] **Popular All Time Minecraft Servers - Minecraft Server List**. `https://minecraft-server-list.com/sort/PopularAllTime/`. (Accessed on 06/14/2022). 2

[11] **Video Game Industry Statistics, Trends and Data In 2022 | WePC**. `https://www.wepc.com/news/video-game-statistics/`, January 2022. (Accessed on 06/14/2022). 3

[12] Alexandru Iosup, Laurens Versluis, Animesh Trivedi, Erwin Van Eyk, Lucian Toader, Vincent van Beek, Giulia Frascaria, Ahmed Musaafir, and Sacheendra Talluri. **The AtLarge Vision on the Design of Distributed Systems and Ecosystems**. *CoRR*, **abs/1902.05416**, 2019. 4

[13] A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Pearson Education, 1999. 5

[14] 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. 5, 36

[15] **About Spigot | SpigotMC - High Performance Minecraft**. `https://www.spigotmc.org/wiki/about-spigot/`. (Accessed on 06/18/2022). 8

[16] **Canalys Newsroom - Global cloud services spend hits US$55.9 billion in Q1 2022**. `https://canalys.com/newsroom/global-cloud-services-Q1-2022`. (Accessed on 06/18/2022). 9

[17] Mark Claypool and Kajal Claypool. **Latency Can Kill: Precision and Deadline in Online Games**. In *Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems*, MMSys '10, page 215–222, New York, NY, USA, 2010. Association for Computing Machinery. 12

[18] **Server/Requirements/Dedicated – Minecraft Wiki**. `https://minecraft.fandom.com/wiki/Server/Requirements/Dedicated`. (Accessed on 07/29/2022). 26

[19] **itzg/minecraft-server - Docker Image | Docker Hub**. `https://hub.docker.com/r/itzg/minecraft-server`. (Accessed on 07/07/2022). 29

[20] **Minecraft Protocol**. `https://wiki.vg/Protocol`. (Accessed on 07/22/2022). 29

[21] **TabTPS [1.8.8-1.19+] Show TPS, MSPT and more in the Tab menu | SpigotMC - High Performance Minecraft**. `https://www.spigotmc.org/resources/tabtps-1-8-8-1-19-show-tps-mspt-and-more-in-the-tab-menu.82528/`. (Accessed on 08/11/2022). 34

[22] Shengmei Liu, Federico Galbiati, Miles Gregg, Eren Eroglu, Atsuo Kuwahara, James Scovell, and Mark Claypool. **TECH REPORT: Waiting to Play – Measuring Game Load Times and their Effects on Players**. April 2022. 38, 44

[23] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. **Peeking Behind the Curtains of Serverless Platforms**. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, July 2018. USENIX Association. 43

[24] **doctorray117/minecraft-ondemand: Templates to deploy a serverless Minecraft Server on demand in AWS**. `https://github.com/doctorray117/minecraft-ondemand`. (Accessed on 02/25/2023). 49

[25] 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. 49

[26] Hao Wang, Di Niu, and Baochun Li. **Distributed Machine Learning with a Serverless Architecture**. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 1288–1296, 2019. 49

[27] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. **Sprocket: A Serverless Video Processing Framework**. In *Proceedings of the*

ACM Symposium on Cloud Computing, SoCC '18, page 263–274, New York, NY, USA, 2018. Association for Computing Machinery. 49

[28] Jonatan Enes, Roberto R. Expósito, and Juan Touriño. **Real-time resource scaling platform for Big Data workloads on serverless environments**. *Future Generation Computer Systems*, **105**:361–379, 2020. 49

[29] Jinfeng Wen, Zhenpeng Chen, Xin Jin, and Xuanzhe Liu. **Rise of the Planet of Serverless Computing: A Systematic Review**, 2022. 49

[30] Gianluca Turin, Andrea Borgarelli, Simone Donetti, Einar Broch Johnsen, Silvia Lizeth Tapia Tarifa, and Ferruccio Damiani. **A Formal Model of the Kubernetes Container Framework**. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles*, pages 558–577, Cham, 2020. Springer International Publishing. 49

[31] **Kubernetes**. `https://kubernetes.io/`. (Accessed on 02/25/2023). 49

[32] **Azure Container Instances | Microsoft Azure**. `https://azure.microsoft.com/en-us/products/container-instances`. (Accessed on 02/25/2023). 49

[33] **Cloud Run: Container to production in seconds | Google Cloud**. `https://cloud.google.com/run`. (Accessed on 02/25/2023). 49

Appendix A

Experiment Configuration

In this chapter, we show the setup and configuration with which we ran the experiments.

Table A.1 shows the home setup the experiments were run on, which is a home desktop computer connected to the internet using Ethernet.

Table A.2 shows the configuration of the Minecraft instance deployed to AWS.

Component	**Value**
CPU	Ryzen 7 3700x
GPU	NVIDIA GeForce RTX 2070 Super
Memory	16 GB DDR 4 SDRAM 1060MHz
Bandwith	350 Mbps

Table A.1: Specifications of the at-home setup used to run experiments.

Name	**Value**
rcon.port	25575
enable-command-block	true
gamemode	1
level-name	world
generate-structures	true
difficulty	0
network-compression-threshold	256
max-tick-time	60000
max-players	999
use-native-transport	true
online-mode	false
enable-status	true
broadcast-rcon-to-ops	true
view-distance	10
allow-nether	true
server-port	25565
enable-rcon	true
sync-chunk-writes	true
op-permission-level	4
prevent-proxy-connections	false
entity-broadcast-range-percentage	100
player-idle-timeout	0
force-gamemode	false
hardcore	false
white-list	false
broadcast-console-to-ops	true
spawn-npcs	true
spawn-animals	true
snooper-enabled	true
function-permission-level	2
level-type	DEFAULT
spawn-monsters	true
enforce-whitelist	false
spawn-protection	0
max-world-size	29999984

Table A.2: Server.properties of the deployed Minecraft instances, empty fields and some default settings are omitted to save space.

Appendix B

Additional Graphs

In this chapter, we present two additional graphs which are omitted from Chapter 5.

Figure B.1 displays the mean times it takes for the server to process a game tick with a more diverse amount of bots walking around.

Figure B.2 shows the highest measured times that it takes the server to process a game tick with a more diverse amount of bots walking around.

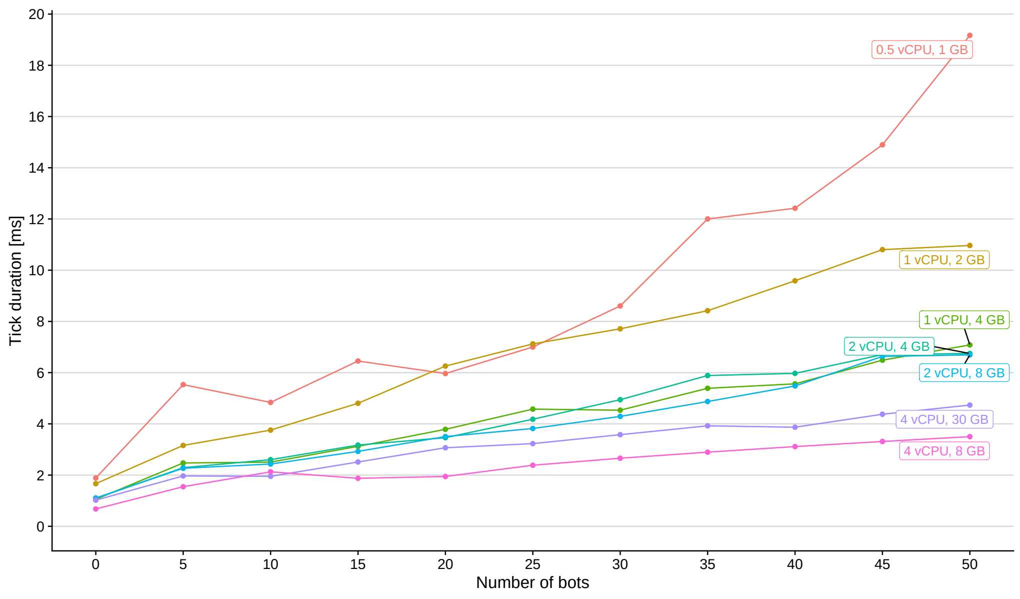

Figure B.1: Average measured time it takes a server to process a game tick with bots walking around.

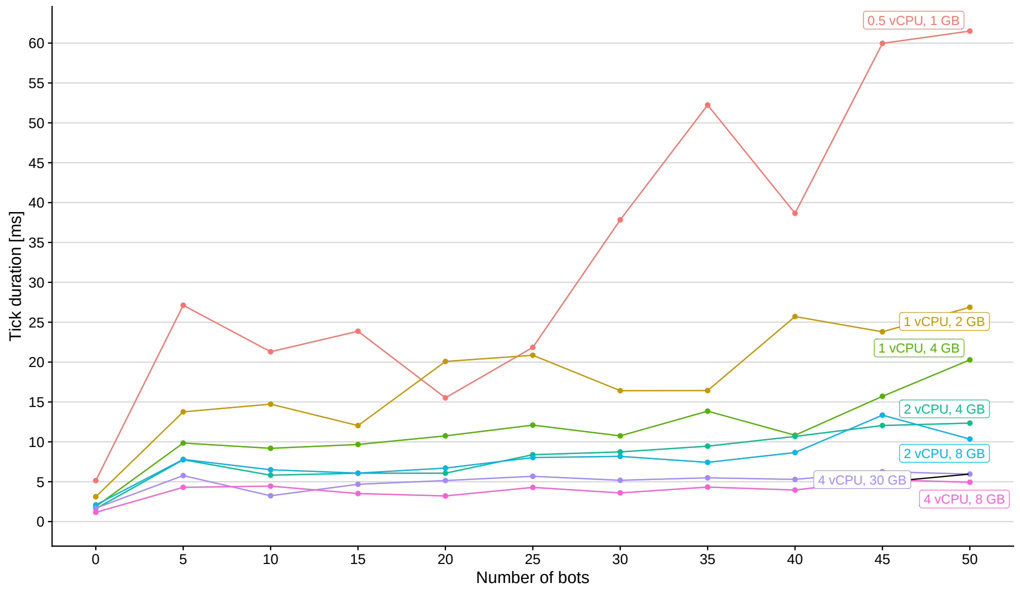

Figure B.2: Maximum measured time it takes a server to process a game tick with bots walking around.

Appendix C

Sanity Checks

In this chapter, we show the results of our sanity check experiments that we ran to verify the results gathered in §5.3.1. Figure C.1 shows the results of the experiment running on a Minecraft instance running on a home computer with 8 GB of memory. Figure C.2 shows the results of the same experiment being run on a Duplicraft instance configured with 1 vCPU and 4 GB of memory. This resource configuration was chosen as it was the lowest amount of resources that got very consistently stable results in §5.3.1. For both experiments, the server setup and method of collecting the MSPT data remains the same as the performance experiment. We now use a single instance of Yardstick to deploy 10 bots every minute, which walk in a square with a diameter of 32 blocks around a coordinate in the world.

The AWS sanity check confirms that the performance of up to 50 bots matches the results we gathered in the original experiment. We note that after 90 bots have joined, each time a new batch of 10 bots joins performance degrades significantly, which reassures us that the good performance comes from the low workload, which after a large number of bots (>100) the server starts to have trouble meeting performance constraints with. However, after 130 bots we can observe an anomaly in our data, where the performance improves the more bots join the instance. Through investigation we find that a single yardstick instance fails to function once the workload becomes too large for it to handle, causing the bots to stop moving and thus greatly decreasing the strain they place on the server.

We also verified the consistent performance by deploying bots on a home computer, up to 30 bots we can indeed see that our original experiments seem to be correct, where the performance remains very stable. However, from 40 bots and upwards we observe this to not be the case. Through testing we find that while running a Minecraft server instance, a

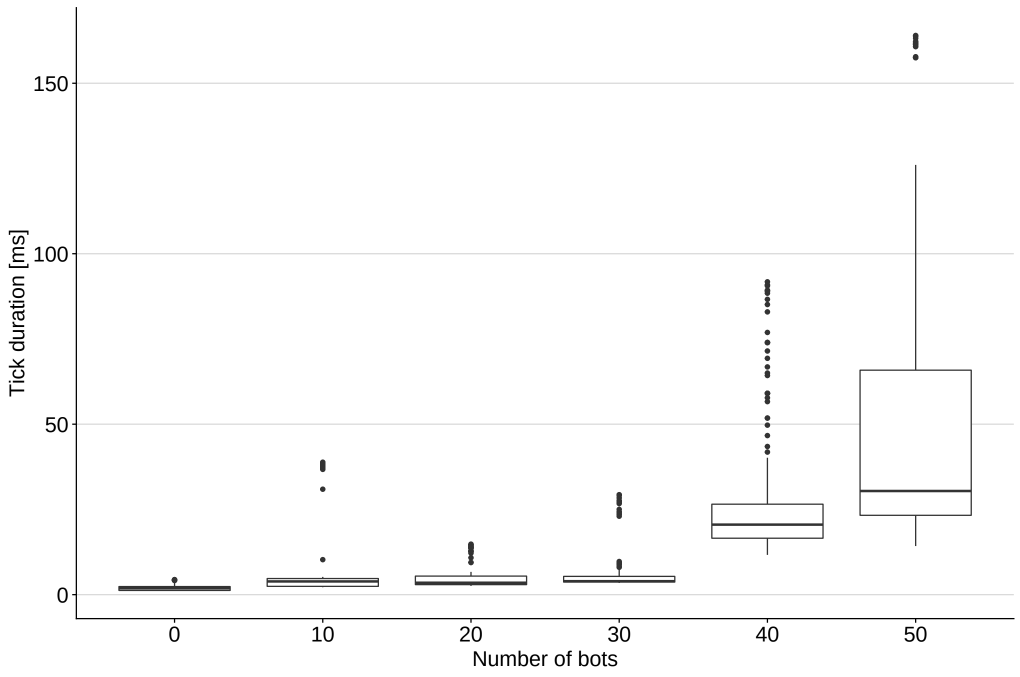

Figure C.1: The amount of time it takes for the instance to process a game tick with increasing amounts of bots walking around on a home computer.

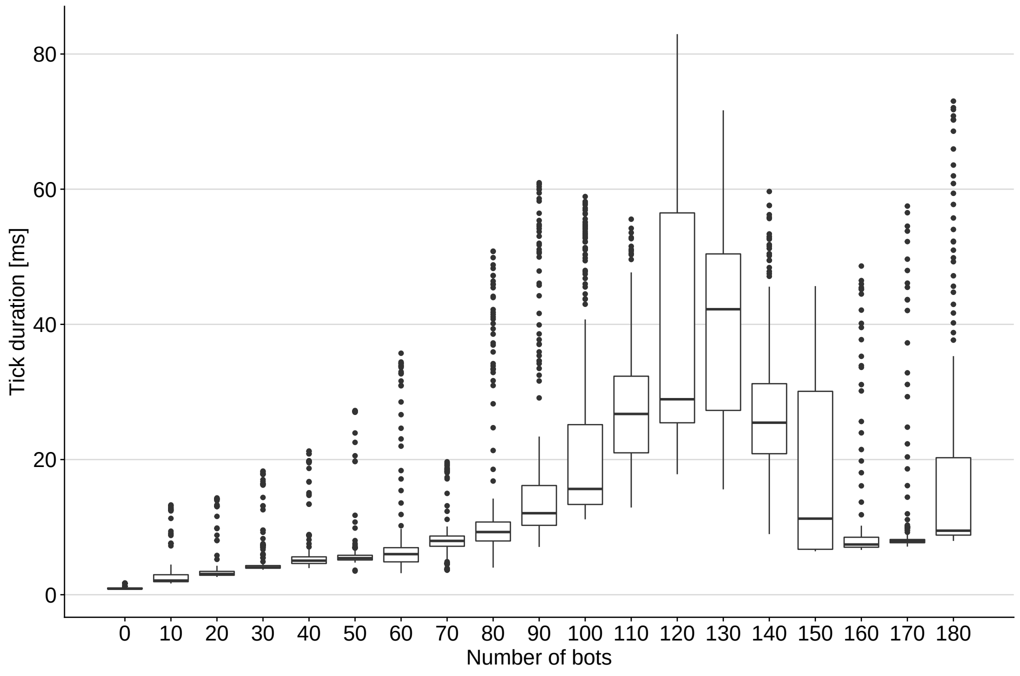

Figure C.2: The amount of time it takes for the instance to process a game tick with increasing amounts of bots walking around on an Duplicraft instance with 1 vCPU and 4 GB of memory.

Yardstick instance simulating bots, a Minecraft game instance and the Duplicraft instance on a single home computer, the machine fails to perform well and starts being unable to provide the computing resources each program needs. This explains why the performance of our home instance starts to have performance issues significantly earlier than the Duplicraft instance.