*software*

Article

# Athlos: A Framework for Developing Scalable MMOG Backends on Commodity Clouds

**Nicos Kasenides * and Nearchos Paspallis ***

School of Sciences, University of Central Lancashire Cyprus, Larnaca 7080, Cyprus
* Correspondence: nkasenides@uclan.ac.uk (N.K.); npaspallis@uclan.ac.uk (N.P.)

**Abstract:** The development of resource-intensive, distributed, real-time applications like Massively Multiplayer Online Game (MMOG) backends entails a variety of challenges, some of which have been extensively studied. Despite some advancements, the development and deployment of MMOG backends on commodity clouds and high-level computing layers continues to face several obstacles, including a non-standardized development methodology, lack of provisions for scalability, and the need for abstractions and tools to support the development process. In this paper, we describe a set of models, methods, and tools for developing scalable MMOG backends and hosting them on commodity cloud platforms. We present Athlos, a framework that allows game developers to leverage our methodology to rapidly prototype MMOG backends that can run on any type of cloud environment. We evaluate this framework by conducting simulations based on several case-study MMOGs to benchmark its performance and scalability, and compare the development effort needed, and quality of the code produced with other approaches. We find that MMOGs developed using this framework: (a) can support a very high number of simultaneous players under a given latency threshold, (b) elastically scale both in terms of runtime and state, and (c) significantly reduce the amount of effort required to develop them. Coupled with the advantages of high-level computing layers such as Platform, Backend, and Function-as-a-Service, we argue that our framework accelerates the development of high-performance, scalable MMOGs, that leverage the resources of commodity cloud platforms.

**Keywords:** software architecture; online games; MMOGs; framework; cloud computing; distributed systems; real-time systems

check for updates

**Citation:** Kasenides N.; Paspallis N. Athlos: A Framework for Developing Scalable MMOG Backends on Commodity clouds. *Software* **2022**, *1*, 107–145. https://doi.org/10.3390/software1010006

Academic Editor: Claus Pahl

Received: 26 January 2022
Accepted: 3 March 2022
Published: 7 March 2022

## 1. Introduction

The unprecedented availability of computing power provided by cloud computing technology has led to a drive towards its adoption by organizations and individuals alike [1]. Especially for businesses, the cost-effectiveness of this technology distinguishes it among other options and has made cloud computing a popular option for hosting enterprise applications, especially at the Software-as-a-Service (SaaS) layer [2]. The ever-increasing drive to migrate services to the cloud stems not only from its relative cost-effectiveness, but also from the ability to innovate and optimize business processes, especially those related to mobile and web technologies [3]. Most enterprise applications must be scalable to accommodate increasing numbers of customers while maintaining a good quality of experience and economies of scale. The services provided by such applications are often optimized for throughput and parallel execution, which makes them easy to scale due to a limited need for synchronization when accessing resources [4]. For the most part, the logic behind the services provided by such applications can be easily migrated to either a private or public cloud.

On the other hand, resource-demanding applications such as backends of Massively Multiplayer Online Games (MMOGs) are traditionally characterized by the need to rapidly synchronize and update their state. This requirement imposes several constraints which have inhibited their migration to the cloud. To cope with such high resource demands, developers and researchers have employed several technologies and approaches. Firstly, MMOGs are traditionally configured to run in *sessions* or *rooms*, utilizing *zones* to divide gameplay requirements among peers in a network [5,6]. These techniques limit the number of concurrent players to cope with the high resource demand. Despite increases in computing power in recent years, even *Multiplayer Online Battle Arenas* (MOBAs) [7] cannot support more than several hundred simultaneous players. In addition, hugely successful MMOGs with persistent worlds such as World of Warcraft have faced problems in managing such a high influx of players and had to break them down into more manageable units, with "*each server [hosting] a community of about 20,000 players*" [8]. Moreover, MMOG backends are categorized as *soft real-time* applications and must deliver a satisfactory *Quality of Experience* (QoE) in a market of ever-increasing competition. To achieve this, they must be able to handle these resource-demanding tasks quickly and thus offer low-latency gameplay [9]. While achieving such high performance relies on having access to raw computing power, MMOGs must also employ a "*cost-efficient design*" [10]. This highlights the need to have high-performance, cost-efficient, and adaptive development methods and tools that can enable MMOG backends to achieve a high level of QoE. Alternatively, recent studies have began exploring the potential of utilizing IoT (Internet of Things) technology to support and extend the main infrastructure. By using smaller but still capable devices, it is possible to *"offload [the] processing [of] demanding [. . . ] tasks to the edges of the cloud and in close proximity to the end users"* [11], thus reducing the network distance and latency, and ultimately increasing the QoE.

Naturally, the vast majority of the research and development for MMOGs in the cloud focuses on the Infrastructure-as-a-Service layer (IaaS), which is inherently more suitable in this scenario due to finer control and less overhead when compared to other layers, such as Platform-as-a-Service (PaaS) or Software-as-a-Service (SaaS). Game developers have traditionally hosted their MMOG backends on dedicated servers and private clouds, rather than public clouds. Nonetheless, recent trends show that there is a slow shift toward higher abstraction layers such as PaaS, and perhaps the use of commodity clouds [12]. One of the probable causes for this shift might be the inherent elasticity of these higher layers. Even though there have been several resources citing the use of higher layers such as Platform-as-a-Service and Backend-as-a-Service (BaaS) for MMOG backends [13,14], the hosting and development approaches used are not standardized in any way. Consequently, there has been very limited effort to develop and evaluate models and methods to facilitate the development of this type of software on such infrastructure.

In this paper we investigate how MMOG backends—which are resource-intensive and latency-sensitive applications—can be developed for and deployed on commodity cloud platforms. Our study is primarily focused on proposing solutions to the problems of scalability and state consistency. We specifically target games with persistent, scalable states that have a high number of concurrent players, but which do not require very low latency. We predict that the use of public clouds, in conjunction with higher abstraction layers such as PaaS and FaaS (Function-as-a-Service) will enable MMOG backends to become more elastic, economic, and may allow developers to utilize common models, methods, and tools to rapidly develop MMOG prototypes.

We present Athlos, a framework for developing scalable MMOG backends that can run on and leverage the resources of commodity cloud infrastructure. Within this framework, we propose a model that aims to abstract the development process by decoupling the game logic and state management from other game development processes such as networking, storage, and data management. Our model utilizes several concepts in business applications, but also provides solutions that allow MMOGs to attain scalable states and runtimes at low latency. In addition, we propose the use of several standardized methods to develop and host MMOG backends. This aims to facilitate the rapid development and deployment of game prototypes using a selection of approaches, both on and off the cloud and at varying layers of abstraction. Our framework also provides tools with which it is possible to create and manage abstract game definitions that do not follow a specific approach and are not linked to a runtime environment. These definitions allow developers to experiment with different approaches and perform analyses before expanding their prototypes to fully implemented MMOGs. By using Athlos, we aim to reinforce the notion that higher computing layers such as PaaS, FaaS, and BaaS are suitable approaches for the development of MMOG backends. Despite our focus on scalable, persistent states, our approach can be generalized to support non-persistent, low-latency, moderately scalable game states as well.

To evaluate our framework, we develop three MMOG prototypes that leverage technologies based on the PaaS, FaaS, and BaaS layers. We guide our evaluation based on three core requirements: (a) provide a feasible method to develop MMOG backends, (b) deliver satisfactory performance, and (c) reduce development effort and improve the quality of the produced code. To assess the ability of our approach to meet these requirements, we explore the following research questions:

1. MMOGs that are hosted on cloud PaaS and utilize the Athlos framework inherit the underlying scalability to achieve a better (lower) *ratio of latency to the number of active players* compared to custom approaches that use single-machine dedicated architectures and do not utilize Athlos.
2. MMOGs based on Athlos and hosted on cloud PaaS sustain a higher *total number of active players* than single-machine, non-Athlos approaches, under the threshold latency of 1000 ms.
3. MMOGs that utilize our framework are able to feature very large (within the limits of the hardware resources being utilized) and expandable game states.
4. When using Athlos, the time taken to retrieve a sub-state of a game world remains constant regardless of the world's size.
5. The development of scalable MMOG backends using Athlos simplifies the development process and results to lower effort and time taken to develop an MMOG backend.
6. The Athlos approach produces high-quality, readable, and re-usable code.

To explore these questions, we deploy the game backend prototypes on a popular public cloud platform and conduct a series of simulations to evaluate their performance under different loads and scales. Then, we conduct an empirical analysis of the prototypes' code to assess its quality, as well as the effort undertaken to develop such games using Athlos. Our results indicate that games based on our approach can take advantage of the framework's methodology to achieve much larger scales compared to dedicated approaches that are used by online game developers, both in terms of runtime and pure state size. In terms of scalability and quality of experience, we show that MMOG backends based on Athlos have the potential to serve a much larger number of concurrent players under a given latency threshold, especially when hosted on the PaaS layer; rather than other layers of cloud computing or using dedicated hosting. Through our code evaluation, we show that Athlos allows developers to create MMOG backends using significantly less effort than other approaches we used, even though the quality of the code could be improved.

The rest of this paper is organized as follows: Section 2 explores the related work, followed by Section 3 which describes our motivations behind this study. In Section 4, we outline our methodology for the development of scalable MMOG backends on commodity clouds. In Section 5 we demonstrate the use of our framework via three case studies. Then, we discuss our experimental approach and evaluation strategy. Finally we present our results in Section 6, and summarize with conclusions and future work in Section 7.

## 2. Related Work

While substantial research has taken place in development practices for MMOG backends since the late 2000s, the use of cloud computing technologies is relatively recent. In the second half of the last decade, we have seen explosive growth in terms of leveraging cloud resources for various types of applications and MMOG backends are no exception.

In this section, we provide a brief overview of research in the area of MMOG backends in terms of evaluation practices, the types of architecture and infrastructure used, and other aspects. We have selected our related works in each of the following subsections based on their affinity to the topic, and in helping understand and address our research questions, mentioned in Section 1.

### 2.1. Performance

The nature of MMOGs presents a need to support large numbers of concurrent players while ensuring an acceptable Quality of Experience (QoE) for everyone [15]. It is therefore no surprise that a lot of studies have explored ways to evaluate their performance.

Perhaps the most important performance indicator for an MMOG backend is the "*global response delay*", also known as "*latency*" [7,15–19]. The latency, commonly measured in the order of *milliseconds*, is dependent on a variety of system parameters and configurations such as processor and memory capacity, network speed, distance, and so on. In [15], the authors evaluate the performance of a cloud-based MMOG hosted on Amazon's EC2 platform using simulations to measure its latency. They then use an empirical approach to measure the degradation of the QoE as a function of the number of players and allocated Virtual Machines (VMs). The results of these simulations confirm the authors' hypothesis that latency increases as more players get connected, while the cost per player is reduced. Using these results and an analytical model, the authors discuss how their VM allocation algorithm aims to "*minimize customer cost while ensuring a minimal threshold on the QoE*".

Another study investigates how to minimize the cost of running an MMOG backend while providing satisfactory QoE [10]. The authors present a resource allocation framework that uses virtualized resources and an "*accurate delay model*" to control the QoE. They evaluate their delay-aware, cost-minimization resource allocation in a private cloud using simulations that are distributed across a large geographical area. Results from this study show that the author's resource allocation scheme outperforms other approaches in terms of cost efficiency, while maintaining a satisfactory QoE.

In a similar fashion, others have also explored the use of virtualization and private clouds in improving the scalability and performance of MMOG backends. Researchers have proposed the utilization of a novel MMOG "*ecosystem*" that uses virtualized resources [20]. The authors evaluate the effect of using virtualized resources on the QoE using a popular MMOG called RuneScape. Their experiments show that the use of virtualized resources can have a negative effect on MMOG game session performance at high volumes, which presents a problem at the large scales seen in MMOGs. They demonstrate the viability of their ecosystem by obtaining results that show that resource under-allocation and the penalties of virtualization overheads can be minimized. An older study by the same authors explores the impact of data center policies on the QoE [21]. Explorations in this study reveal that static resource provisioning techniques are relatively inefficient, and that dynamic resource provisioning algorithms are better at allocating resources under high loads. Their experimental results, which were obtained from trace-based simulations, highlight the advantages of real-time parallelization and load balancing in MMOG backends.

To minimize network traffic, bandwidth use, and latency, some have proposed software-based methods. Various independent studies [17,22–24] have proposed a variety of methods to control or limit the number of messages sent by a backend to the connected players. The Area of Interest (AoI) is a concept that enables MMOG backends to reduce bandwidth consumption when communicating the game's state to the connected players. This is achieved by using a variety of factors to determine whether a player should receive an update that was initiated by an event within the game world. Examples of such factors include a player's proximity to the event, the view of the player, the relevance of the event to the player, and more. Using the concept of AoI and other similar techniques, processing time and bandwidth use can be significantly reduced, enabling MMOG backends to support higher numbers of concurrent players below certain latency thresholds.

### *2.2. Architecture*

A paper by Kasenides and Paspallis [12], provided a systematic study of architecture types used in MMOG backend development. These were categorized into three groups: *Client–Server*, *Peer-to-Peer*, and *Hybrid* architectures. The type of architecture used in MMOG backends is important to their development as it can affect performance, security, and deployment strategy. Client-server architectures are by far the predominant type of architecture used for this type of application [12]. Studies have shown that the security and efficiency of a distributed, centralized architecture can allow MMOG backends to support thousands of concurrent players while taking advantage of various parallelization techniques such as *zoning*, *replication*, and *instancing* [20,22]. On the other hand, researchers have also explored the possibility of utilizing decentralized, Peer-to-Peer (P2P) architectures [17,25,26]. While P2P architectures appear to improve performance in several cases, their decentralized nature means that they leave MMOGs vulnerable to exploits, cheats, and hacks. To combat this, research has focused on encryption to eliminate state manipulation issues. However, the use of encryption algorithms brings overheads, which negates the performance advantages of this approach. To eliminate these issues, others have proposed hybrid architectures [18,25], which allow game content to be distributed in a P2P network, while all game-specific tasks are handled by a secure, centralized system.

### *2.3. Infrastructure*

Despite advancements in MMOG backend research in the last decade, a large number of games still utilize dedicated infrastructure to enable gameplay through the Internet. The dedicated hosting approach is considered as a "traditional" and well-tested approach to hosting an MMOG backend. Various components of this infrastructure can include database servers, game servers, web application servers, and so on [27]. While there are several advantages in this approach, such as full control over the hardware and data, and perhaps a lower initial latency compared to clouds, it is very difficult to scale. To eliminate scalability issues faced in this approach, past studies have utilized dedicated server clusters [10].

Even though clusters offer a temporary solution to the problem of scalability, they still require the manual installment, configuration, and maintenance of hardware. On the other hand, more recent studies have explored the use of private clouds for MMOG backends [15,21,28]. This type of infrastructure enables MMOGs to attain many positive attributes of cloud computing, such as high availability, elasticity, resource virtualization, and automation. These studies have shown that MMOG backends on private clouds are not only feasible but also greatly benefit from resource virtualization, load balancing, and other attributes of on-demand computing. Specifically, regarding scalability, the need to manage infrastructure directly is eliminated, and resources can be utilized on-demand, leading to reduced costs. On top of these, novelties such as the "*Provisioning Manager*" (PM) presented in [28] enable further optimizations to take place by collecting live performance data, thus allowing a more efficient allocation of resources.

Even though private clouds may offer several advantages compared to the dedicated approach, their prohibitive cost means that they are only accessible to large businesses. Small businesses or individuals simply cannot purchase the hardware or manpower required to run even a small private cloud. In contrast to private clouds, public/commodity clouds are more accessible to businesses or individuals with limited budgets, allowing them to use otherwise unavailable resources [29]. For MMOG backends specifically, the use of public clouds can uplift the development process by utilizing services such as Amazon's EC2 and Microsoft's VMs in the IaaS layer, or Google's App Engine (GAE) and Firebase in the PaaS, FaaS, and BaaS layers. According to [30], utilizing these resources requires offloading data-intensive tasks to cloud servers instead of running them on client devices. The authors argue that even though these types of clouds offer resources at unprecedented scales, their data centers are rarely located close to users resulting in "*large communication latency*". Various studies have proposed solutions to this problem by providing complementary resources closer to their clients—such as Cloudlets [31] and by using *Fog Computing*. In recent years, public cloud providers are attempting to solve this problem on their end, by creating smaller, but still powerful data centers in various locations around the world. These can provide resources at a much lower cost compared to private clouds or dedicated infrastructure, which significantly reduces the cost-per-player for MMOG backends [32]. In turn, this enables the development of a wider variety of MMOGs. Studies have shown that public clouds can also enable games to scale "*an order of magnitude more players than state of the art [. . . ] game servers currently support*" [32].

In spite of most research targeting the IaaS layer, some have argued for the use of PaaS services to power MMOG backends. One critical advantage of higher layers such as PaaS, FaaS, and BaaS, is the immediate deployment of backends without a need for configuration. Secondly, these serverless approaches allow developers to focus on backend logic, rather than server and hardware maintenance. Perhaps the most crucial advantage of these layers is the fact that the deployed applications are elastic by default, without the need to manually handle scaling and load balancing [33]. An example of a serverless approach is Google's App Engine, which allows the deployment of applications on Google's scalable infrastructure without the need to manually manage scaling and hardware during deployment [14]. App Engine allows applications to utilize scalable storage and persistence options, authentication APIs, email options, and more, while offering various options in terms of development environments, programming languages, and frameworks. In general, serverless approaches offer a more streamlined development process over IaaS, which can lead to a reduced Time-To-Market (TTM) [33].

## 2.4. Consistency

An important factor influencing the development effort in MMOG backends is consistency, which is defined as "*the need to provide players with mutually consistent views of the gaming arena in a timely manner to allow fair game play*" [34]. Due to the large scales encountered in MMOGs, both in terms of the state and the number of concurrent players, maintaining consistency becomes increasingly difficult. Hence, various studies have attempted to provide solutions to this problem by offering ways to manage consistency. The provision of *manageable consistency* can be categorized into two groups. Firstly, geographically influenced consistency or *zoning* divides the game world into regions or zones [20]. This reduces the problem by enforcing several rules that dictate how objects between these regions can interact with the game world. The second group divides worlds based on *behavioral* patterns that emerge from the player's actions. These patterns help determine the view and ability of a player to affect the game state, similar to the AoI concept discussed in Section 2.1. Meanwhile, others have explored the possibility of offering different levels of consistency for various types of events depending on their context. For instance, *viewing* events, during which players can only observe the game world can be categorized as *weak consistency* events, whereas intricate *interactions* between the players require *strong consistency* [17]. For the latter, consistency can be ensured by ordering events and utilizing synchronization processes to ensure correct gameplay. Other systems, such as EventWave [4] offer the ability to run multiple events in parallel while ensuring consistency by identifying *state-dependent* and *state-independent events*. The execution of state-independent events can run in parallel—thus improving performance—while state-dependent events require a higher level of synchronization.

## 2.5. State Persistence

Developers and researchers have utilized various techniques and tools to enable MMOG backends to persist in their state. The requirements of MMOGs mean that persistence systems must operate at low latency and be (a) highly available, (b) consistent, and (c) fault tolerant. The CAP theorem suggests that systems can only have two of these three properties [35]. To work around this trade-off, some have suggested the use of *eventual consistency* [36] which is a form of weak consistency. This guarantees that when no new updates are made to an object, accessing the object will always return the last updated value. This approach works by allowing some inconsistency for the sake of improving performance under highly concurrent conditions. Compared to other types of software, latency is considered far more important than throughput in MMOG backends. These systems also require a higher ratio of updates to reads or writes compared to other applications [37], and the players can be relatively tolerant to the loss of data, as long as the recovered game state remains consistent.

Persistence systems can be categorized into three groups: RDBMSes, key-value datastores, and caches, each of which has their own advantages and disadvantages and is more suitable depending on the task. While RDBMSes such as MySQL, Oracle, etc. provide a more rigid schema that ensures consistency across database records, they are harder to scale. On the other hand, key-value datastores such as Google's Firestore or Amazon's DynamoDB are easily scalable and offer flexible schemas, but are less standardized and lack support for complex queries. MMOG backends can also benefit from the use of caches, such as Redis, Memcached, etc. due to their very fast access speeds, which can significantly reduce latency [37]. Depending on the requirements, developers can utilize each of these types of systems through services provided in either public or private clouds to persist information.

The current state of the art indicates that developing MMOG backends faces a diverse set of important challenges. While various studies have made significant progress in this area, there is evidence that more research is needed to better address the research questions we identified in Section 1.

## 3. Motivation

The development of scalable MMOG backends is a complicated and lengthy process that, as seen from the related work, spans across many areas and stages of the software development lifecycle. Many game development frameworks attempt to provide multiplayer gameplay by offering third-party solutions. Some popular examples are Photon Engine [38], which allows games developed using Unity to be hosted on the cloud, and Amazon's GameLift [39], which supports the operation and scaling of dedicated cloud servers for multiplayer games. Such frameworks provide tools with which developers can easily manage social aspects of games, game economies, matchmaking, and so on. However, despite their usefulness, these frameworks have their own limitations. To the extent of our knowledge, none of these frameworks offer ways to create scalable MMOG backends without having to implement code from scratch, despite offering dynamically scaled servers. Secondly, despite offering a large variety of features through their APIs, existing frameworks restrict developers to using specific programming and runtime environments. Furthermore, the existing frameworks provide limited game models to support the development process, and in many cases are incompatible with other frameworks that offer such models or even other useful features. Motivated by these aspects, our research attempts to advance the development process of MMOG backends by offering methods with which they can be inherently scalable, and can be developed using a variety of tools and development environments. We aim to provide a language-agnostic, approach-independent game model that supports the development of MMOG backends on the cloud and envision that this will allow developers to utilize existing code that implements various game features without the need to start working from scratch to realize scalable MMOG backends. To help with the presentation of our results, we introduce a sample MMOG game, described in the following section.

### 3.1. Mars Pioneer

As an example, we present a simple 2D strategy game concept called Mars Pioneer (MP). With this, we aim to provide an example that motivates the creation of a generic game model. The objective of the game is for players to colonize the planet Mars by building various types of constructions to gather resources, conduct research to improve their gathering rate, and ultimately, to win over other players by controlling a larger percentage of the planet. The game area is infinite and can expand as more players join the game. It is divided into cells, each of which contains resources that can be gathered by the players after spending resources to construct buildings.

From this very basic game concept, we can extract several items that can be abstracted into a game model. For instance, the game area of a *world*, often called the *terrain*, can be managed in different ways depending on the type of game. In MP, we utilize a 2D grid to represent terrain, which means that everything within the game area must exist on one of the *cells* in the grid. However, other types of games may need to utilize a different type of world, in which *entities* can move around the terrain freely, following a coordinate system. To allow the terrain to expand indefinitely and be generated efficiently we organize it into *chunks* and generate each chunk of terrain as-needed, using a *terrain generator* that implements a procedural generation algorithm [40].

By developing Mars Pioneer and other games, and by reflecting on their models and development methodologies, we have gained insights that guided the formation of several abstractions. More importantly, we have identified several requirements that the model must fulfill in order to be utilized in development. Most importantly, the model must be generic enough so that it can be used in a variety of games and support a variety of technologies. Without this property, the model's usefulness will certainly be limited. In addition, the concepts outlined above must offer a balance between abstraction and development effort. One of our objectives is to decrease the development effort required to create such games. A model that abstracts key components, but which can also support concrete, custom-tailored solutions is important to this goal. With regards to customizability, the model must also be flexible enough to handle different games, and perhaps unforeseen modeling requirements. Hence, our aim is to create a model that can feature certain core elements that can be customized or extended to meet the requirements of a wide range of MMOGs.

## 4. Methodology

Researchers and game developers have used a plethora of methods to implement and deploy their MMOGs on the cloud. Based on the results of previous studies [12,41] and insights from studying various games, we have developed a set of models, methods, and tools with which MMOG backends can be developed for and hosted on commodity cloud platforms. In this section, we describe our approach, presenting Athlos, a framework for developing scalable MMOG backends on commodity clouds.

### *4.1. Model*

Our model abstracts the details of various types of game objects and the relationships between them and can be extended or modified to adapt to the model requirements of any game. Using this model, we aim to facilitate rapid development methodologies by abstracting information related to game items and then allowing developers to write their code, without having to implement their own model from scratch.

We categorize game-related objects in two groups based on how they can be extended to form sub-types: *Non-extensible* (NX) types are types that cannot be specialized to form sub-types. Such types keep a single information model across all objects. An example of a non-extensible type is the *Player*, which holds personal information about players in a game. On the other hand, *extensible* (X) types are types that can be specialized to form sub-types. Such specializations are based on each game's unique mechanics and modeling requirements. Examples of such types are various types of actions that may be carried out by a player (e.g., constructing or selling a building), or different types of entities that may exist within a game world (e.g., buildings, trees, player entities, and more).

Our model defines several game types which are common to many games:

**Player (NX)**—We define a player as an actor or character that takes part in a game. A player may control one or more *entities* and can issue *actions*. In addition, a player may be controlled by a human or an artificially intelligent script and can be a member of a *team*. The player type is used to record personal information about a player such as their name, email address, game preferences, and more, without containing any information related to the game world.

**Team (NX)**—A team is a collection of *players* who usually work towards a common goal. Depending on the game's mechanics, teams may or may not be formed during gameplay, and may have their own attributes—such as a team flag, color, or collective resources that are being gathered by their respective players.

**World (NX)**—A world is a physical domain in which *entities* can exist. Depending on each game there may be multiple worlds available for players to join. To support a large variety of games and their world states, we divide the worlds into three categories based on the movement potential of entities within them.

- *Uniform worlds* are worlds in which entities can exist in a 3D coordinate system but are not divided in any plane, thus allowing entities to exist and move freely between points. Examples of such worlds include those found in combat or racing games, in which entities can move around without being attached to a particular set of points.
- *Square-tiled worlds* are worlds which are divided in a two-dimensional grid which consists of *tiles* or *cells*. One or more entities can exist in a tile, depending on the game's rules. In such worlds, entities can usually move either up, down, left or right, to the adjacent cells. While the state of such worlds is modeled in a 2D plane, it can still support 3D spaces using a *height* property for entity locations. Examples of such games are commonly found in the Turn-Based Strategy genre, as well as puzzle games, where entities can only move based on a pre-defined set of points in the world.
- *Hexagonal-tiled worlds* are similar to square-tiled worlds, but allow a wider range of movements to be made by entities: up, up-right, down-right, down, down-left, and up-left.

This categorization allows a game to utilize a suitable type of world depending on its state requirements and rules. For instance, games that feature grid-based states may benefit from square-tiled or hex-tiled worlds which make it possible to move between hard-coded positions in the grid. On the other hand, other types of games such as racing or combat games may utilize uniform worlds to allow a greater freedom of movement.

To form the terrain of worlds, we use *terrain cells* and *chunks*. A world may have geographical limits on its terrain, outside of which entities may not exist. To impose these limits, we use attributes that indicate the maximum number of rows and columns of terrain cells that can exist in the world. In a similar way, we use a height limit to impose height restrictions on the worlds. These restrictions work by (a) forcing entities to only move within legal bounds, and (b) limiting the valid chunks that a terrain generator may create. For finer tuning, games can customize these restrictions by creating out-of-reach zones that encompass specific parts of the game world, depending on each game's rules. Our framework supports infinitely large terrains by setting these attributes to negative values, which indicate the absence of these limits.

**Terrain cell (NX)**—A terrain cell is an individual cell containing information about the terrain in a world. The state of terrain can be used as part of a game's mechanics and rules. Terrain cells can be used even in games without an actual terrain to form levels or stages or to represent a basic unit in the state of a world or level.

**Terrain chunk (NX)**—A terrain chunk is a collection of terrain cells. Chunks are used to efficiently generate and store a number of cells instead of managing them individually, allowing terrain to be accessed more efficiently. The number of cells stored in each chunk can vary, but the maximum size of all chunks must remain constant to ensure that terrain can be accessed in a reliable way and scaled smoothly. The problems faced and the reasoning behind the use of terrain chunks are discussed in Section 4.2 and evaluated in Section 6.

**Entity (X)**—An entity is an object that can exist inside a *world*. Entities can be positioned in a world based on the type of world. We categorize these into three groups, based on their potential to change their state during gameplay:

An entity may be *static*, meaning that its state does not change. An example of a static entity would be an indestructible item, such as an ammunition box. Some entities can be described as being *stationary*, meaning that their position in the world cannot change, even though the rest of their attributes can. For example, a tree or shrub could be described as a stationary entity because it can be grown or destroyed—which means that its state can change. However, a typical tree would not be able to move and could thus be described as stationary. Finally, *dynamic* entities are entities of which the state can be changed fully.

The categories mentioned above are abstract and are only defined to identify the potential of an entity to change its state. This categorization attempts to minimize the number of operations required during each game cycle by leveraging the properties of the entity types mentioned in various contexts. For instance, static and stationary entities may be excluded from several state updates as their state is guaranteed to remain unchanged, in order to save processing power and bandwidth.

**Action (X)**—An action is carried out by a *player* inside a specific *world* and may or may not be related to one or more entities within the game. In general, actions have an effect on the state of a world, and the properties of its entities and terrain.

**Game session (NX)**—A game session is a management type used to authenticate and identify a player within a game. Game sessions are created when a player is authenticated (i.e., signs in or goes online in a game). For games without the need for authentication, game sessions can be used to simply identify a player.

**World session (NX)**—Similarly, a world session is used to identify a player within a world. World sessions are an extension of game sessions and are created when a player joins a world. These are used to identify and verify player actions during gameplay.

In addition to these main types, we define several utility types which are all non-extensible by default:

**Partial State (NX)**—A partial state enables the storage and communication for a part of a world's state. It is composed of a set of entities and terrain cells and is specific to a player's perspective within a world—thereby tying it to a world session. The use of this type allows us to limit the size of the state accessed at a single time. Due to the very large scales seen in MMOG worlds, only a part of the entire world state must be made available to any entity, player, or even the entire backend at any time. This approach may enable backends to (a) efficiently carry out operations on smaller pieces of data during game cycles, which may reduce latency, (b) retrieve only the necessary information at a time, which may reduce database operation costs, and (c) reduce bandwidth usage resulting in better economy over time.

The partial state is formed based on how entities perceive the world around them. By default, each entity has an *area of interest* [42], within which it can perceive the state of the world. We define this area to be spanning outwards from an entity's position for a specified distance, which is determined by game-specific rules. For example, an entity with a long-range weapon (e.g., a sniper rifle) may have a larger area of interest compared to one with a shorter range (e.g., a pistol), thus allowing it to see and interact with entities that are farther away. Depending on the AoI and various other factors, each partial state may contain different sizes of data.

**State update (NX)**—A state update models a change in the existing state of the game, and is received after an initial partial state has been received by the clients. State updates are meant to update an even smaller subset of the world's state when an event occurs, by sending information only about the updated items instead of an entire partial state. For instance, if a player constructs a building at a certain location in a world, a state update of this event would only include the newly created entity (building) and the terrain cell onto which it was built instead of the entire partial state that includes them. Partial updates can thus be described as state "deltas" or "diffs", which include small, usually incremental changes in the state of the game that should not cause the system to retrieve unnecessary items. Consequently, state updates can help reduce the latency of the system, avoid unnecessary database operations, and reduce the bandwidth requirements of an MMOG.

**Positioning and direction (NX)**—Further to these core concepts, we also identify several properties of entities that can be used to describe their position and direction in a large set of games. Firstly, we define *MatrixPosition* and *GeoPosition*, which locate an entity within a world. Matrix positions are generally used in tile-based worlds, whereas geo-positions are used in uniform worlds. Secondly, we define various *Direction* types, which enumerate the possible directions an entity may be facing toward. For both position and direction properties, we include types that are associated with changing their state—such as *Movement* and *Rotation*.

**Games and rules**—While not a usable type, we identify a *Game* as a namespace which encompasses all of the information about a particular game and contains its related objects—such as the *players*, *teams*, and *worlds* in it. This type can be optionally implemented for a particular game when there is a need to persist globally accessible information, especially when it is not related to any other type.

Further to these, we identify *Rules* as an important, common component of MMOGs. Each MMOG is governed by a set of rules which dictate what actions can be made by the players and how these actions can affect the game state. However, we argue that due to the large variety of games and game types that exist, a generalization of rules for all MMOGs—or even a subset of MMOGs—is not possible. To the best of our knowledge, we claim that games are made unique by having different rules and that attempting to abstract them may result in inefficiencies during the development process.

## *4.2. Methods*

A common problem seen in multiple types of software is the lack of support for scaling, maintenance, and evaluation in terms of the development process. Monolithic applications tend to follow a rigid paradigm resulting in tightly coupled code bases that are hard to change. While this approach has its own advantages—such as enabling the quick deployment of products—we argue that MMOG backends can benefit by moving away from this, and towards a more modular approach. The modular approach can (a) facilitate code re-usability thus reducing development effort, (b) enhance code structure leading to improved maintainability, and (c) enable software components to be swapped in and out as required, encouraging the evaluation of different methods. Throughout this section, we describe new methods which facilitate the modular development of MMOG backends through the Athlos framework.

**Infrastructure**—We firstly categorize the possible approaches in terms of infrastructure to differentiate the development methodologies that can be used in each approach. Our framework focuses on enabling the development of MMOGs running on serverless technologies and attempts to standardize the development methods used in this approach. To a lesser extent, we also aim to support the development of MMOGs that run on the IaaS layer or dedicated machines because some types of games may benefit from this approach.

The IaaS or dedicated option is best suited for the development of games that require very low latency, such as First-Person Shooter (FPS) games or several types of Real-Time Strategy (RTS) games. This approach is ideal for cases where the runtime of a game needs to offer a wide range of customization options, as well as improved performance. Despite its advantages, IaaS/dedicated is less optimal for elasticity and depends on either vertical scalability or the use of containerization systems to provide horizontal scalability. Where needed, this approach can be utilized in conjunction with public cloud container orchestration platforms, such as Amazon's Elastic Container Service (ECS), Google's Kubernetes [43], or Agones [44] to achieve elasticity.

On the other hand, the serverless approach is ideal for MMOGs that do not exhibit very low latency requirements and do not require a fully customized runtime. Several types of games may benefit from utilizing this approach, such as some types of Turn-Based Strategy (TBS) games, various Role-Playing Games (RPGs) featuring fully persistent worlds, and other turn-based games. The serverless option allows MMOG backends to be fully elastic, scaling automatically based on demand according to preset policies and configurations. While enabling scalability and providing a higher abstraction that hides hardware complexities, this approach also makes it harder—but not impossible—to customize the runtime of a game. For instance, background code execution options are very limited compared to the IaaS/dedicated approach. Another problem that has to be circumnavigated when using this approach is the implementation of state update mechanisms. The limited customizability of serverless technologies makes it hard to implement bi-directional communication between clients and servers, which is required by several types of MMOGs. Most serverless approaches rely on web container technologies such as Java Servlets to make services scalable. These technologies are inherently unidirectional and follow the request-response cycle to serve requests [45]. To overcome this problem, we divide the state update requirements of games into two groups based on the latency requirements of games. Firstly, we utilize *long polling* or *short polling* to offer state updates to less demanding games, such as turn-based games, puzzle games, and so on. To meet the low latency demands of high-performance MMOGs, we allow the utilization of various real-time, bi-directional technologies such as WebSockets, which are either included in products like Google's App Engine Flexible or are provided by third-party vendors.

**Architecture**—While the architecture and networked components used in each game can change, we identify that there are certain required architectural components when developing MMOG backends. Previous studies have suggested that the most suitable type of architecture to host MMOGs is the client-server model, due to its improved security and reliability [22,46]. Supported by the evidence in these studies, Athlos utilizes a generic client-server architecture, as shown in Figure 1.

The responsibilities of a *client*, in order, are to:

- Receive input from the player and translate it into a corresponding *action*.
- Optionally enforce a subset of the game's rules based on the *local* state, such as physics, collision detection, and so on.
- Update the player's local state.
- Transmit the state of the player to the server.
- Receive and present an updated view of the game's *global* state to the player.

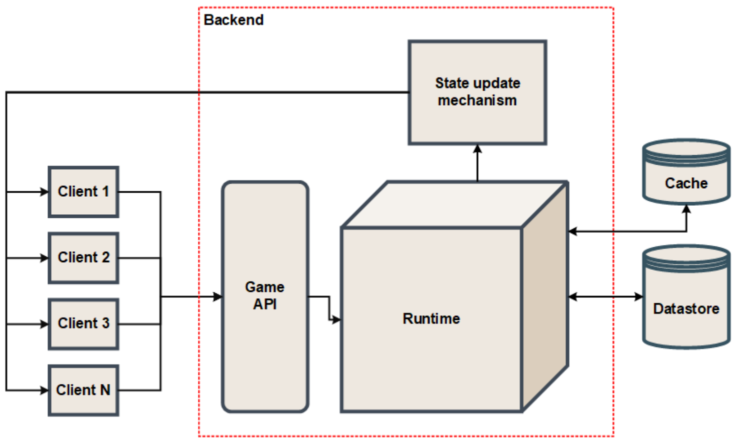


**Figure 1.** A common architecture used in MMOGs developed with Athlos.

On the other end, the server environment consists of an *Application Programming Interface* (API), a game *runtime* and a *state update mechanism*. A game's API is responsible for exposing game functionality to the clients as endpoints, enabling the server to receive player actions, decode them, and validate them before they are sent to the runtime for execution. Upon receiving action requests, the runtime is responsible to enforce the game's rules and execute actions accordingly. We recognize that several games may require resource-intensive operations to enforce game rules, such as the calculation of complex geometry for objects, collision detection, ray-casting, and so on. Even though our framework allows developers to choose where these operations should be executed, we propose that complex operations involving local states should be offloaded to client devices in order to alleviate the server's runtime. However, in terms of the global game state, our framework works in favor of *client passivity*, meaning that all local rule enforcements should only affect the local state.

When an action is executed, the state of a world within the game must be modified to reflect the action being made. For instance, throwing a grenade may alter the state of the terrain within a certain radius when the grenade explodes. State modifications are a critical process in MMOG backends, as they are done in rapid succession, simultaneously, and by a large number of players. It is therefore important to optimize this process as much as possible, to avoid any negative effects on the QoE perceived by the players.

For better QoE, we utilize a low-latency cache component, ideally one that is optimized for scalability in addition to performance. This may positively impact a game's QoE by enabling faster write/read/update speeds than other types of persistence options. We argue that due to their high-performance nature, distributed caches can be utilized to ensure that games can offer *strong consistency* for their states. Despite that, caching is by definition only a temporary solution for storing the game state.

On the other hand, relational databases or key-value data stores can be utilized to persistently store the game state. Even though it offers a persistent option, interacting with a data store is much more expensive both in terms of resources and time, compared to a cache. To avoid negative effects on the QoE, MMOG backends can either utilize a high-performance, low-latency key-value datastore, or a combination of a cache and a persistent option. For instance, the state of a world can be temporarily stored in a cache for efficient access during gameplay, whereas the backend can execute backup operations using *background tasks* in relatively *large intervals* to ensure that the game state is stored persistently in case of a failure.

Updates made to the world state must also be communicated back to the clients. The large numbers of concurrent players in MMOGs mean that updating the state of all clients every time an action is made is a very costly process. In fact, our experience shows that this can be the weak point in an MMOG backend architecture [41]. For this reason, such architectures must incorporate an abstract yet customizable state-update mechanism, which they can utilize to effectively carry out this task.

To solve the state-update problem, we define the state update mechanism component shown in Figure 2, that is responsible for (a) defining what type of update has taken place (*definition*), (b) identifying which clients must receive the state update based on game-specific filtering logic (*filtering*), (c) composing the state update from the view of each player (*composition*), and then (d) disseminating the state update to them as efficiently as possible (*distribution*). For the first step (definition), we identify two major types of updates—*refresh* and *delete*. *Refresh updates* can be used to refresh either a part of, or the entire client's state, depending on the action that triggered the update. On the other hand, *delete updates* are used to delete parts of the state that should no longer be accessible or have become irrelevant to the client. Furthermore, each state update is given an area within which it is perceived to have an effect. We define this as the Area of Effect (AoE) of an update, the size of which depends on the action that triggered the update. During this important first step, our aim is to limit the scope of the items being updated for each client during the composition process.

In the filtering step, a variety of factors can be used to limit the number of clients that need to receive the state update. One of the most common factors seen in related works is the distance between the AoE of an update and the AoI of the entities of an observing player. Since both of these areas can be perceived as circles, basic geometry can be used to calculate whether a player owns an entity that has an AoI overlapping with the AoE of the update, thus determining if the player needs to be made aware of this update. During this step, developers are encouraged to customize the filtering process in order to incorporate game-specific factors that affect the decision to send an update to a player. For instance, they may choose to add Line-Of-Sight (LOS) filters to block out updates based on obstructing objects, utilize event priority systems, and more.

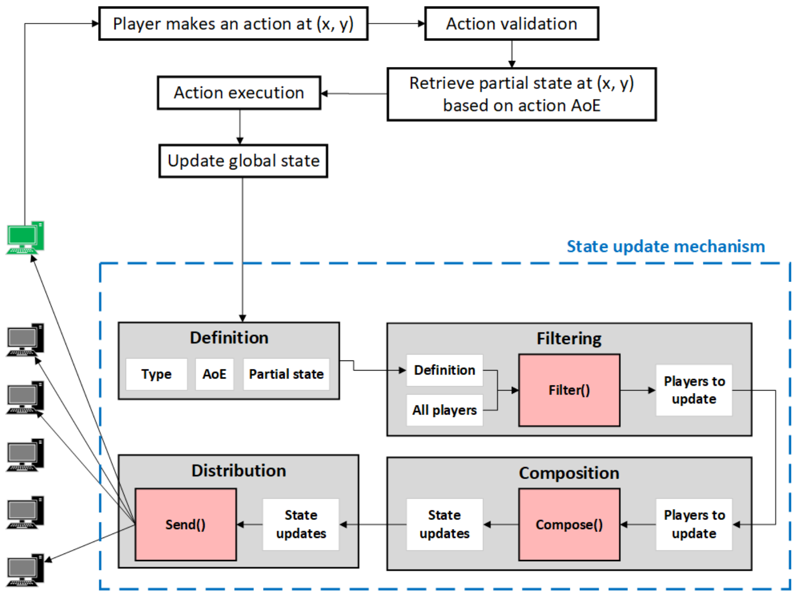


**Figure 2.** Steps involved during the state update process—the state update mechanism.

Since each player has a different view of the game world, there is a need to compose different state updates for each player. In the composition phase, the view of the world is composed from the perspective of each player and stored in memory for distribution. During this step, it is possible to add extra information to the state updates—if needed—by customizing the corresponding function. For instance, games that feature global information such as the weather state or the state of resources in a world may take advantage of this option to inject additional data into the state update while it is constructed. Finally, after composing the state updates, the mechanism must distribute them to their intended players. While Athlos offers default methods for distributing the state based on several bi-directional communication protocols, this part of the state update process can be fully implemented by developers from scratch allowing MMOG backends to utilize tertiary communication platforms or third-party services for state updates.

**Persistence**—In general, applications employing a monolithic design will bind various aspects of the development process, restricting re-use and flexibility. In such applications, the User Interface (UI), logic, and data access code are often intertwined in a single program, resulting in unmaintainable code and a lack of performance insights. To avoid this problem, we use the Data Access Object (DAO) pattern within the persistence layer. The DAO pattern offers several advantages in this context. Firstly, it separates the UI, logic, and data access layers, enabling code re-use and aiding code maintenance. Secondly, it allows the use of a common API to manage database records or objects. Thirdly, it supports interchangeability in terms of the database system used. As a result of the latter, developers may choose to test their games using various database systems while maintaining the same logic and UI components. In turn, this may enable them to experiment, and finally pick the best-performing or most cost-efficient option.

Interactions with the persistence layer occur using DAO interfaces, which expose common data management operations in an organized way and depending on the context. We divide these interfaces into three categories, based on the number of instances possible for each game type. The `UniqueDAO` interface enables interactions with types of objects that are unique and can therefore have only one instance in each game. A common use case of this particular interface is when a game can only have a single, common world for all players. On the other hand, the `MultiDAO` interface exposes interactions with types of objects that can have multiple instances, such as players, teams, and so on. We also employ a third interface called `WorldBasedDAO`, which defines interactions with objects that exist within a particular world context—for instance, entities, world sessions, and more. In Athlos, we define the use of each of these DAOs based on the expected default behaviors of objects in games. However, we also allow developers to customize the type of data access interface used for each type thereby embracing the uniqueness of each game. By using this approach we limit the available operations depending on the context, which in turn helps to avoid mistakes in the code which would otherwise break type definitions.

**Game definitions**—The abundance of approaches and tools that can be used to develop MMOG backends makes it hard to track the various components involved. While each game is unique, we identify various similarities, not only in terms of the model discussed in Section 4.1, but also in terms of the development processes used. In our framework, we attempt to abstract these similarities as much as possible, with the intent to expedite development. At the center of our methodology lies the *game definition*, which is an approach-independent description of the attributes and types making up an MMOG. An Athlos game definition contains important information about the game itself, the objects within it, and the relationships that exist between them.

Firstly, game definitions record information such as a game's name and the type of its worlds. Game projects can be created out of these definitions, with each project based on a particular infrastructure approach as discussed in Section 4.2. Furthermore, we allow complete independence in terms of client-side and server-side approaches by offering several options to choose from for either one within the same project. This allows the client and server to be written in a different programming environment. Most importantly, the game definition contains information about any extensions made to the framework's existing types as well as custom data types that are declared by the developer. Within our framework, we provide a standard set of types and their attributes that are used to model common game elements. We make it possible to customize or extend these types to include additional information that is specific to each game. For standard and user-defined types alike, the definitions hold type declarations which include the names and attributes of each type. We are confident that these declarations make it possible to model a very large variety of games. As part of these type declarations, we also include service definitions that make up a game's API and their related request and response type models. Game definitions can be saved as files, opened, and modified using an editor program with which developers can explore their contents in a controlled way. Finally, these definitions are used to generate a boilerplate code for each game. This code contains the basic project structure, model, and management code for the game that can then be extended by developers into a game prototype. In support of this rapid development approach, the definitions include meta-data that helps speed up and automate the code generation process.

**Data serialization**—Data serialization is a necessary step to enable nodes in a system to communicate with each other using messages. The large variety of development approaches as well as differences in infrastructure and architecture make the use of a common serialization protocol a problem in MMOG backends. To bridge this gap, our framework uses Google's Protocol Buffers library [47]. Protocol Buffers (PB) is a widely used library with support for a growing set of languages, including those used in popular game engines such as Unity and Unreal Engine. A major advantage of using Protocol Buffers is that serialization is done automatically, without the need to manually convert data into bytes. This removes the burden of having to create game-specific serialization mechanisms which is often an error-prone, tedious, and time-consuming process. Thirdly, PB is a language-agnostic technology, capable of transmitting data across applications running on different language environments. This presents us with the opportunity to support a variety of environments in our framework, which could not have been possible with the use of other language-specific technologies such as streams in C++, C#, or Java. Lastly, binary communication using PB allows the efficient transmission of data across nodes. Studies have shown that PB can transmit game states faster and using a fraction of the size taken by other formats such as XML or JSON [47].

To utilize PB, our framework converts game definitions into Proto files. Using Athlos tools, we then generate two classes per type. The first is a PB class, generated through the Proto file by a PB compiler, which includes the attributes defined in the type definition. PB objects are relatively heavyweight and immutable, so we reserve them only for the communication of data. Furthermore, we generate lightweight, plain-object classes, which can be used during the runtime and while interfacing with the data access layer. The two types of classes can be used to represent the same actual object as they have the same attributes. We therefore offer seamless conversion between a plain object and a PB object through our framework, using the `Modelable` interface which allows a PB object to be converted into a plain object and, inversely, the `Transmitable` interface, which enables a plain object to be converted to a PB object.

**Networking—**To serve user requests, applications employ either the request–response cycle, the publish–subscribe pattern, or both. Even though they have certain peculiarities, MMOG backends function in an identical way. However, the diversity of the approaches, in terms of infrastructure, architecture, and development options, leads to relatively concrete implementations that bind the service logic with each service container. In comparison with other types of applications, MMOG backends also have to endure a large number of incoming requests from the players. This demands services that operate efficiently and can be scaled horizontally to accommodate processing on multiple computing nodes.

To eliminate these problems, we first decouple service logic from containers by utilizing Protocol Buffers to represent request and response data. We define a service as an approach-independent element that simply takes a generic request as input, processes it within a service function, and then returns a generic response. Initially, a client sends a serialized request to a server which is then deserialized, validated, and processed in a service. To allow concrete implementations, we wrap services into the corresponding service container for each approach. For instance, to deploy services using Java Servlets, we first decode the request data and then call `serve()` on the appropriate service within a servlet. Once the service returns, we then encode its response data and send it using the concrete Servlets API. Due to the standardized nature of the request–response cycle, we have been able to implement this in three different concrete APIs and we expect that this approach can support the majority of server-side APIs and programming languages.

While services can be abstracted in this manner, the concrete method used largely depends on the infrastructure employed for each game. In the IaaS and dedicated approach, we leverage the full customizability of the system to serve requests using Google's Remote Procedure Call system (gRPC) [48]. gRPC provides several advantages to the communication layer of MMOG backends. First and foremost, it uses Protocol Buffers, which (a) offers improved performance compared to other serialization methods and (b) makes it compatible with our serialization strategy. Second, it is suitable for latency-sensitive scalable applications deployed on the cloud [49]. Third, it provides server and client APIs with which communication can be implemented more easily compared to manually utilizing low-level sockets. Fourth, it supports asynchronous, bi-directional calls using streams which can improve performance in both the client and server. gRPC also allows developers to implement their server and client programs in different programming languages while following a common game model.

We categorize our services in four groups in terms of the data flow, largely influenced by the gRPC system:

- **Message to message** (*Unidirectional*) services, which transmit a single outbound message and receive a single inbound message.
- **Message to stream** services, which transmit a single outbound message and receive multiple inbound messages.
- **Stream to message** services, which transmit multiple outbound messages and receive a single inbound message when the outbound messages are sent.
- **Stream to stream** (*Bidirectional*) services, which transmit multiple, continuous outbound messages and receive multiple and continuous inbound messages.

Despite its usefulness in the IaaS/dedicated approach, gRPC is not compatible with the unidirectional communication scheme employed by most serverless approaches. Products like Google's App Engine Standard, Google Cloud Functions, or Amazon's Lambda functions are based solely on short-lived connections that do not allow bi-directional data flow. To provide support for abstract services in higher layers such as PaaS, FaaS, and BaaS, we define the `AthlosService` interface, which enables the definition of *message to message* services and makes use of generic requests and responses. Just like with gRPC, these services are platform-agnostic, implement the appropriate methods, and are utilized in each platform's web containers. In this approach, however, we match services to specific web container URIs, since there is no support for remote procedure calling without gRPC. Lastly, we take into consideration that for some serverless approaches it may be possible to utilize bi-directional communications. For instance, Google's App Engine Flexible allows this through its WebSockets API. To leverage the advantages of the publish–subscribe pattern, we embrace the differences of each serverless platform to ensure that games can operate optimally, using in-house components when possible.

On the client side, we utilize service *stubs*, which are observers to incoming messages. In the IaaS approach, stubs are automatically generated by the gRPC framework and must be implemented by the game developer in order to handle inbound messages depending on the service activated. Clients can communicate through an interface provided by gRPC, enabling them to send both *synchronous* and *asynchronous* requests. We follow a similar pattern in the serverless approach, even though gRPC cannot be used. Our framework uses the service definitions to generate custom service stubs that are similar to those created by gRPC. To provide a common interface for the client regardless of the backend's communication approach, we created an API that supports synchronous and asynchronous requests using binary HTTP messages.

**Scalability and performance**—In a previous work, we have demonstrated the limitations of several NoSQL data stores provided by popular public clouds such as Amazon's AWS, Google's Cloud Platform (GCP), and Microsoft's Azure [41]. Using a simplistic, tile-based game, we found that there are significant limitations on how many tiles of game state can be stored in a single object when using these data stores. Most data stores have a limit of around 1MB per object [50], which limits *state scalability* to several thousands of tiles.

To bypass this limitation, we decouple the terrain state from any entities that exist in the world and allow the terrain to exist independently of any entities that are on it and vice versa. This approach complicates state retrieval compared to a coupled terrain–entity approach. Despite that, it presents a better logical connection between game elements as some types of games may not feature any terrain but may still have entities. In addition to that, decoupling is necessary to achieve the required scalability as the size of the terrain is significantly reduced when no entity data is recorded within it.

We also explore various possibilities to allow game states to be scaled. A solution we used in our previous study does not divide its worlds into chunks but rather stores cell information in a single entity. The advantage of that approach is that a single query can be made to fetch the entire world's state. However, this has a major drawback—it assumes that the entire world's state can be stored in a single datastore object but that it is not scalable due to the restrictions in datastore object size discussed before. An alternative solution is to store each cell individually. However, this is also disadvantageous because an $N$ number of queries will be needed to retrieve $N$ cells. Retrieving large numbers of cells is very inefficient as database queries are generally resource-intensive. Additionally, public cloud data stores often charge their users by the number of queries made, which makes this approach more expensive.

To overcome this problem, we depart from using a single database entity to represent terrain. Instead, we model terrain using *cells* and *chunks*. Cells allow the division of a world's geographical area into small units that can have individual states. We organize these cells into chunks, which are formed from a collection of adjacent cells. To allow for reliable state retrieval, each game can set a strict limit on the number of cells that can be stored in a chunk—called MAX_CELLS. While the number of cells can differ from chunk to chunk, this constant may not be exceeded to ensure that individual cells can be retrieved in an efficient way. To group cells into chunks, we use a map data structure, in which the key is a hash value of the cell's coordinate, and the value is the terrain cell itself. It is therefore possible to retrieve a cell that exists inside a chunk in constant time, by hashing its coordinates and retrieving the value of the key matching that hash. Similarly, chunks also have coordinates and given a constant MAX_CELLS, mathematical operations can be used to efficiently retrieve a needed chunk of cells. The diagram in Figure 3 shows how chunks are used to group cells.

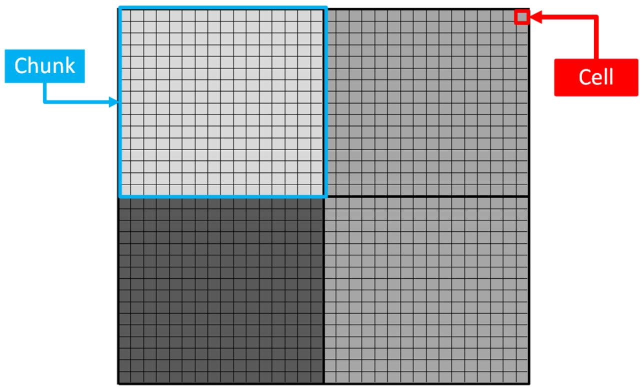


**Figure 3.** Chunks are used to organize cells in groups, allowing terrain to be scaled. In this example, a chunk holds 16 × 16 cells.

Considering that cells are parts of a chunk, some extra operations are needed to retrieve and manage them. For example, a frequent use case would be to retrieve a cell's state. Given that a cell $E$ has coordinates $(r, c)$, we have to calculate the respective coordinates $(R, C)$ of the chunk $H$ in which $E$ is included in order to access it, using chunk arithmetic and the $MAX\_CELLS$ constant. Since the geographical limits of a world are known, calculating which chunk includes the cell is straightforward and has constant time complexity. We believe that this small calculation overhead justifies the added benefit of scalability.

The chunk-based approach makes it possible to simultaneously extend the world state to massive scales and reduce the number of queries needed to fetch a part of the state. It enables MMOG backends to overcome the limitations in game state imposed by public cloud data stores by utilizing multiple chunks, each of which can include from 4 × 4 up to 64 × 64 cells depending on the game's definition and requirements. This range allows a chunk to fit within the datastore limits. Furthermore, the number of queries required to fetch the state of a world can be significantly reduced when compared to the individual-cell approach. Through the use of chunks, the number of queries required to fetch the game state is reduced by a factor of $MAX\_CELLS$ which is a far more efficient order of growth. We also expect that chunks will make the process of generating and transmitting game state data easier and more efficient. Entire chunks of terrain can be generated in a single batch instead of making multiple calls to a terrain generator, which is computationally expensive as it carries out complex calculations.

The second major component of a game's state are entities, which exist and can be persisted as independent objects. Game runtimes handle the retrieval and management of entities by scanning an AoI around a given location and then retrieving the contained entities and terrain. Unfortunately, the complexity and scale of such operations grows linearly as the number of entities increases, which presents a *runtime scalability* challenge. Based on insights from other studies, we mitigate this problem by separating *active* and *inactive* entities on each computational node to divide the processing power requirements. Each node is responsible for processing its own active entities while ensuring that the state of inactive entities is synchronized from other nodes. While this method is effective for IaaS environments, it is of little use in serverless approaches where instances cannot be directly managed. In this case, the automatic scaling and load balancing offered by serverless systems works obstructively as it obscures the different computing nodes and thus makes it impossible to define which instance will host a particular entity. Instead, to improve the efficiency of the runtime in PaaS systems we propose that developers complement this by using game-specific indexing mechanisms where necessary. Such mechanisms can enable MMOG backends to track groups of entities based on a given context (i.e., their proximity, type, etc.) in order to more efficiently query them, and access their state.

## *4.3. Tools*

The development of scalable MMOG backends on commodity cloud platforms entails the use of novel models and methods. To utilize these models and methods, we have devised a set of tools that can accelerate the development process and enable game developers to rapidly prototype scalable MMOG backends. We break down these tools into four groups: our project editor, project generator, the Athlos API, and software libraries.

**Athlos Project Editor**—The Athlos framework provides the foundations to develop scalable MMOG backends using platform-agnostic game definitions, as discussed in Section 4.2. To allow developers to create and manage their game projects in an easy, reliable, and more efficient way, we have developed the Athlos Project Editor. The Editor provides the facilities to create, define, and then manage game projects using a Graphical User Interface (GUI) environment shown in Figure 4.

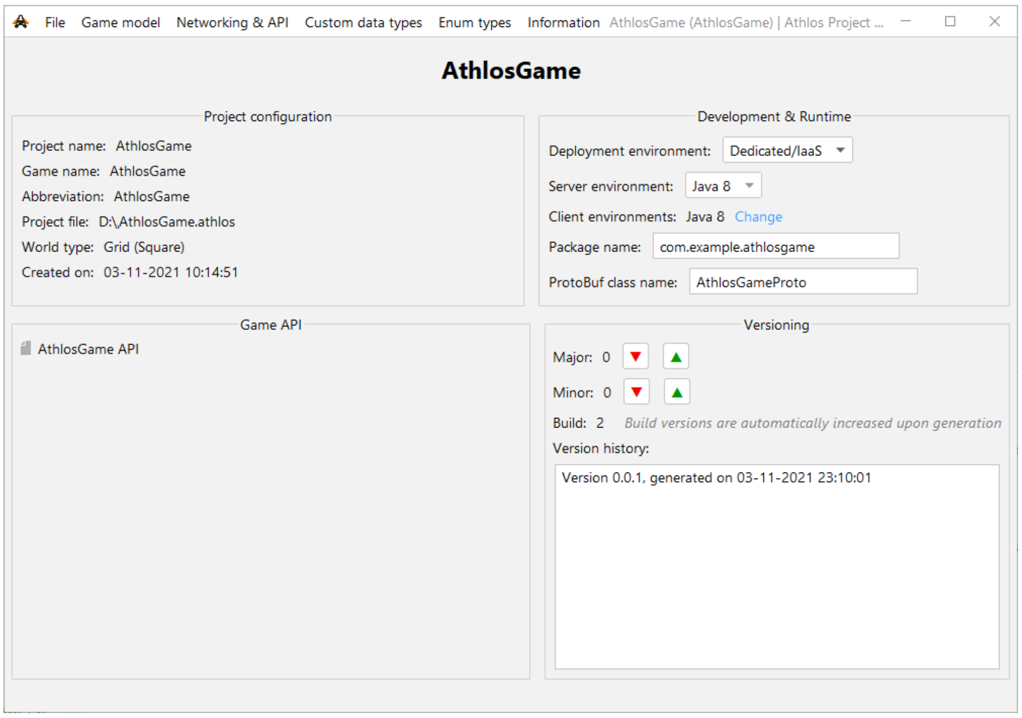
Developers can utilize this editor to create new projects, within which they can define their game model by either customizing or extending the default model types discussed in Section 4.1, or by creating their own data types. Furthermore, the editor facilitates the definition of game APIs through approach-independent services, as discussed in Section 4.2. Developers can create and manage request and response models which are used to compose the services that make up a game's API. In addition to these, the editor also allows developers to quickly generate several default game API services which are commonly encountered across a variety of MMOG types. Examples of these include services related to creating and managing worlds, retrieving game states, authentication, and so on. Moreover, developers can use a feature of the editor to automatically create data management services based on the model they have defined. This automatically generates services for Creating, Retrieving, Updating, and Deleting (CRUD) objects of the types defined within the project.

For new projects, the editor presents developers with various options to select from, such as the infrastructure type to be used, the runtime, server, and client environments to implement, world types, etc. While most of these options can be changed later, we do not allow developers to change their project's infrastructure type. Provided that there are major differences between various components in the IaaS/dedicated and serverless approaches, we believe it is best for developers to use a consistent approach. The projects are saved in JSON-formatted files with the .athlos extension and can be inspected manually using a text editor. These files can be communicated to other developers and opened and modified on other machines to generate code for the same project. Finally, the editor allows developers to track game versions by changing the major and minor releases manually, and by automatically increasing build versions once a successful code generation is made.

**Code generator**—The generator is a software tool that parses game definitions created using the project editor and then generates a concrete MMOG boilerplate code. Due to the complex nature of the code generation process, and the need to facilitate multiple approaches at each stage, the generator works by splitting its tasks in a series of distinct processes within a pipeline, shown in Figure 5.

Several steps in this pipeline use various methods that enable the produced code to function properly in the context of an MMOG backend. For example, the Protocol Buffer generation stage involves the creation of a Proto file which includes the definitions of all data types within the game, including framework types, enumerated types, and user-defined types. Where applicable, gRPC services are defined within the same file.

The output of this step is a set of files containing PB definitions based on the model and, optionally, service definitions for gRPC. These files are used in the next step, where the generator uses the PB compiler to generate implementations of PB classes in the specified backend language. Due to the polymorphous nature of extensible data types, it is necessary to generate interfaces that allow these types to be used interchangeably and be stored in common collections. To allow this, the framework automatically creates game-specific interfaces which are implemented by their sub-types. By using interfaces, it is possible to generalize a group of entities, or even all entities, and access them collectively. To ensure that PB and plain-object classes are interchangeable and can be converted back and forth, the generator injects conversion code to both types of files, based on the class attributes.



AthlosGame

Project configuration

Project name: AthlosGame
Game name: AthlosGame
Abbreviation: AthlosGame
Project file: D:\AthlosGame.athlos
World type: Grid (Square)
Created on: 03-11-2021 10:14:51

Development & Runtime

Deployment environment: Dedicated/IaaS
Server environment: Java 8
Client environments: Java 8
Package name: com.example.athlosgame
ProtoBuf class name: AthlosGameProto

Game API

AthlosGame API

Versioning

Major: 0
Minor: 0
Build: 2 *Build versions are automatically increased upon generation*
Version history:
Version 0.0.1, generated on 03-11-2021 23:10:01

**Figure 4.** A prototype version of the Athlos Project Editor.

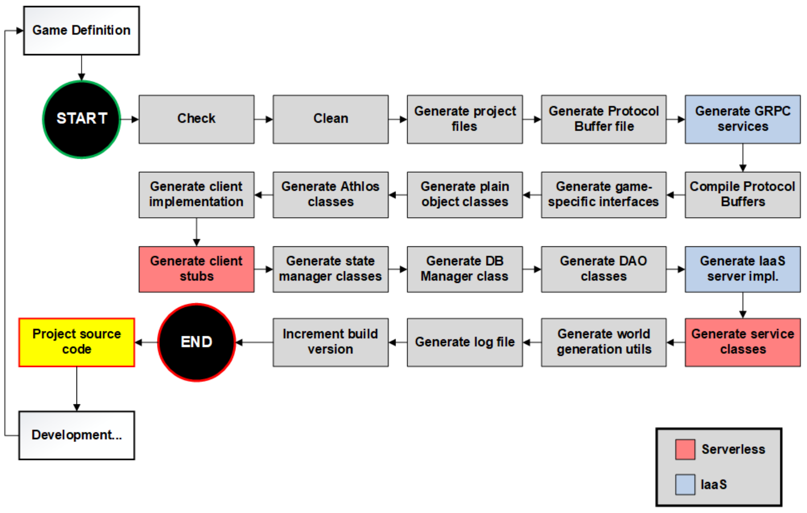


**Figure 5.** The series of steps involved in generating a project's source code.

Another important part of this process is the generation of state management classes, which contain code related to managing the state of a world. To allow developers to quickly obtain the game state where needed, our framework provides a common, modifiable state management API. This API divides the state of a game based on *world contexts*, which allow the retrieval of items related to a particular world. Secondly, to facilitate interactions with the data access layer, we implement a data management API which includes DAO classes as discussed in Section 4.2 as well as a *Database Manager* that organizes them and contextualizes interactions with the data access layer.

For the IaaS/dedicated approach specifically, we generate a gRPC server with specific extensions that allow it to function as part of the Athlos framework. Each server instance has its own built-in memory cache which can be utilized to save local data, or as a regular cache in the case of single-instance servers. Conversely, we generate custom service classes for serverless projects. These classes provide an intuitive interface for developers to quickly create service logic and call the corresponding services using stubs in the front-end. At the end of the generation process, the generator also creates utility classes with which worlds can be generated for the specified game. With the help of procedural generation, developers can create infinitely large worlds that increase in size when players explore them. Alternatively, this also makes it possible to load limited-size worlds from databases or files. Finally, the generator outputs a *generation log* which allows developers to review and debug the entire process, and then increments the project's build version. We use this to track game versions and to avoid the replacement of previously generated code. Figure 6 shows the various components involved in the definition and subsequent generation of an MMOG backend that utilizes Athlos, as well as the structure of the framework's architecture.

During a project's definition, it is possible to define a game-specific model by extending or customizing the default Athlos model. The generator uses the definition to create source code, which is divided into several code packages as shown in Figure 6. Some of these are fully generated and there is no need to modify their code. For instance, the authentication, state manager, or service stubs can be utilized without any additional code. Other code components are partially implemented and must be extended in order to function properly—such as the GameServer, which contains basic functionality and can be started without modification, but does not perform any game-specific tasks. On the other hand, some components are generated, but must be fully implemented in order to be utilized. Examples of these include game services, which have to contain game-specific logic, or in most cases the state update mechanism. By generating a large part of a project's structure, our framework aims to reduce the development effort and promote rapid development.

To speed up the game definition and generation processes, we allow developers to utilize the project editor and generator within the same software bundle. The generator itself uses the generation pipeline shown in Figure 5 to allow code generation for a variety of environments. Due to the implementation overheads of creating code generators that handle such a large variety of approaches in so many different components, we have implemented only a handful of approaches as a proof-of-concept. At the moment, Athlos implements code generators in Java for the entire stack and supports gRPC for the IaaS/dedicated approach and three different serverless hosting methods: Google's App Engine Standard, App Engine Flexible, and Cloud Functions. However, the tools we have created can be easily extended to include support for additional programming languages, as well as for a wider variety of serverless options.

**Athlos API**—The Athlos framework API defines a set of abstract, reusable classes that enable the development of scalable MMOG backends. Some of these classes are simply abstractions to the data modeling requirements of the framework, while others also implement functionality related to specific components of the architecture, such as networking, data access, and so on. The framework API is divided into four packages called "core", "server", "serverless", and "client".

The "core" package includes items that are used in both client and server environments, such as data model abstractions, protocol buffer files, and more. Within the "server" package, an important item is the `GameServer` class, which provides abstractions for implementing dedicated game servers and implements functionality that allows servers to run tasks concurrently. Furthermore, this package contains useful state management attributes and methods, provides support for logging, abstracts the persistence layer, etc. On the other hand, the "serverless" package contains abstractions that enable developers to utilize the serverless infrastructure. Firstly, it defines the `ServerlessBackend`, a class that is used to parse incoming data into Athlos services. More importantly, it enables developers to use the framework in support of a variety of serverless technologies, such as Google's servlet-based App Engine, or Cloud Functions, by implementing utilities and patterns that minimize the development effort. Finally, the "client" package contains a generic `GameClient`, which defines an API that enables clients to communicate with their corresponding servers. Within this class, we also provide support for logging, state management, and concurrency. The `GameClient` is extended by the `DedicatedGameClient`, which defines protocols to communicate with a dedicated game server, while the `ServerlessGameClient` defines protocols for communication with a serverless backend.

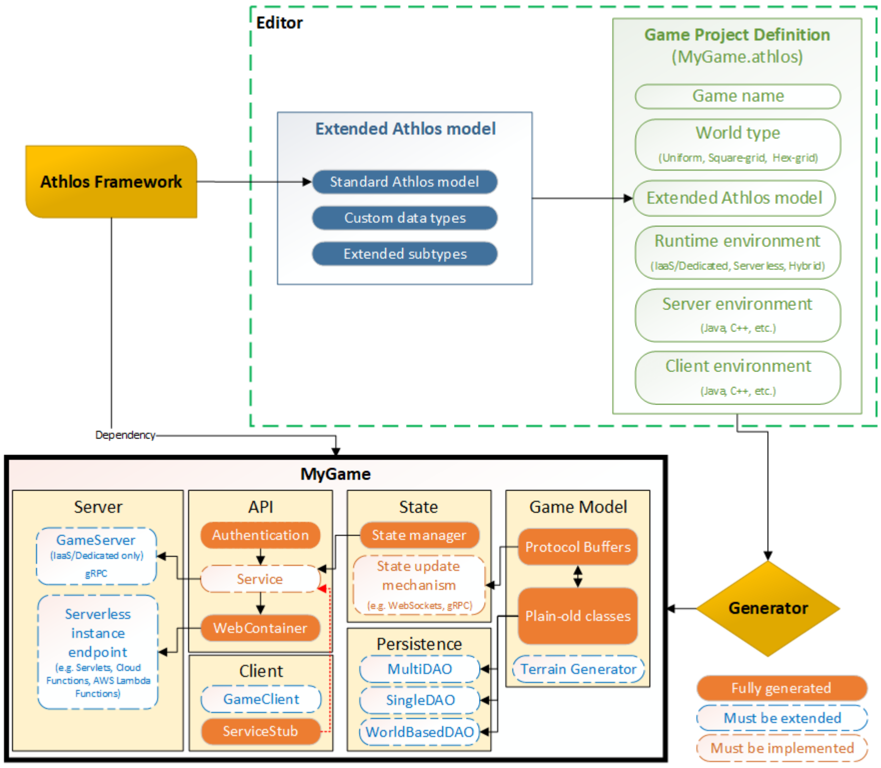


**Figure 6.** The Athlos project structure, including various tools and architectural components of the framework.

While the Athlos API and its components can be utilized in their default form, it is possible for game developers to extend them to support more complex functionality where necessary. As a proof-of-concept, we have implemented this primitive version of the Athlos API in Java 8, and for both dedicated and serverless computing technologies.

## 5. Case Studies

Using the models, methods, and tools we have described in the previous sections, we implemented three relatively simple multiplayer games. The purpose of these case studies is twofold. Firstly, we attempt to provide proof-of-concept, verifying that our framework can enable the development of different MMOGs. Secondly, we use these implementations to evaluate our framework in Section 6. In this section, we describe these case studies, and how they utilize the Athlos framework.

### *5.1. Mars Pioneer*

Our first case study is Mars Pioneer, which was also discussed in Section 3.1. Mars Pioneer is a simplified multiplayer strategy game that features *square-tiled worlds*. Players can join a selected world, and play by constructing various *building entities* that enable them to gather resources.

To develop the game we firstly define the game's properties within a game definition using the Athlos Project Editor. The game model is created by using the default existing types mentioned in Section 4.1, as well as by creating our own custom data types where needed. To expose game functionality to clients and allow gameplay, we define game services and their corresponding request/response models. Figure 7 shows parts of the game's API, as defined in the editor.

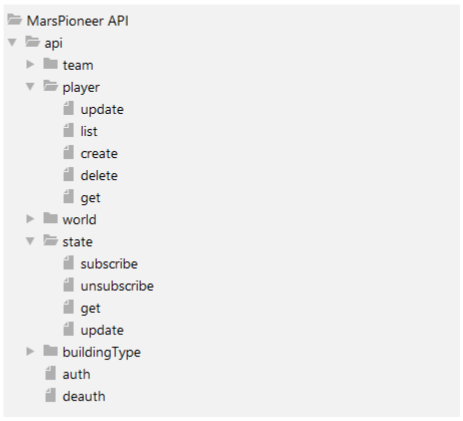


**Figure 7.** Part of the Mars Pioneer game API, as shown in the Athlos Project Editor.

In terms of architecture, we utilize a *serverless backend environment*—Google's App Engine Flexible, that communicates with Java-based desktop clients using HTTP requests. Athlos automatically serializes incoming and outgoing messages using Protocol Buffers when the game's services are called and automatically maps requests to specific service URIs, which enables communication between the client and server without the need to explicitly define code. For persistence, MP utilizes Redis within GCP's Cloud Memorystore, a high-performance distributed cache to enable fast read/write operations. MP performs back-up operations in the background using Google Cloud Tasks that store the game's state on Google's Firestore. These operations are coupled to the normal cache operations within the persistence layer, which means that when an object is written to the cache, a background operation is also launched to write or update the object in the datastore. This offers strong consistency for the runtime, while also offering weak/eventual consistency for the game's backup state. For the game's state-update mechanism, MP utilizes WebSockets, which are supported by GAE Flexible, and makes small customizations to the mechanism that enable it to work more efficiently for this game. Lastly, on the client-side, we enable communication with the server by implementing the corresponding service stubs and managing the local state. We visualize the game state, mostly for debugging purposes, using basic 2D graphics and text as shown in Figure 8.

### 5.2. *aMazeChallenge (AMC)*

As a second case study, we take an existing educational programming game called aMazeChallenge and implement its backend using the Athlos framework. aMazeChallenge [51] is a turn-based, multiplayer, maze-solving game in which players code their avatar to escape a maze by writing code in a block-based language. The mazes are created in a 2D square grid world and the game uses a custom generator to create different types of mazes.

AMC is different compared to conventional multiplayer games in terms of its gameplay, as players do not have direct control over their character. Instead, the players' code is uploaded at the start of each game and executed at each game round, which determines the character's next move. Figure 9 shows the area of the game, which includes the maze, players, and various objects that can be picked. The game's runtime enables gameplay using a priority queue. When a player uploads their code, they join the back of the queue and their code is executed in order. For every game turn, the runtime executes the code of all players, applies their move, and updates the global game state.

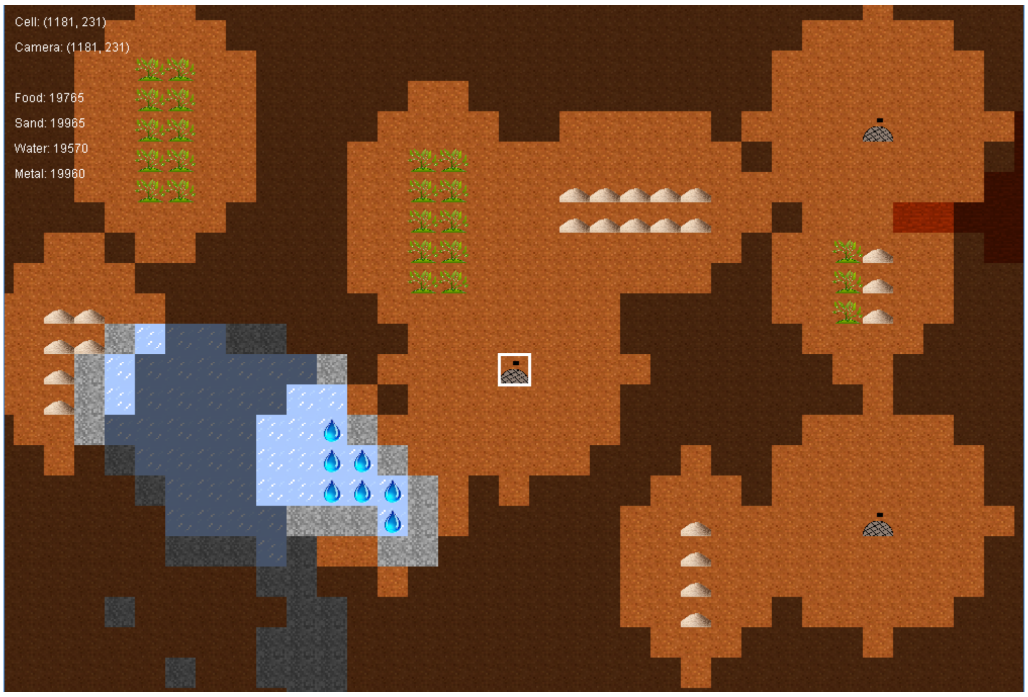


**Figure 8.** A Mars Pioneer client, presenting the state of the game to the player.

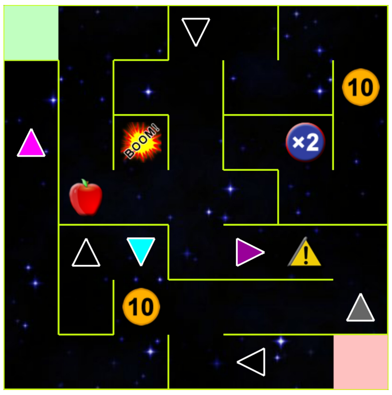


**Figure 9.** Screenshots of the aMazeChallenge client during a competition. The client presents a view of the game area to the player, which contains the maze area and the players and objects within it.

aMazeChallenge uses an Android-based client and a backend hosted on GAE Standard. To enable communication between these two components, JSON-formatted strings are sent using web services implemented in Java Servlets. When a request is received, adjustments are made to the game state and then persisted using a cloud-based version of Memcache provided with GAE. Memcache allows the game to store and access the state quickly runtime which has a positive impact on latency. To persist other information such as game sessions, player information, or scoreboards for longer periods of time, the initial implementation uses Google's Cloud Datastore. To disseminate state updates back to the players, AMC uses long-polling which allows clients to request the state of the game in 1 s intervals and then update their local states.

As an upgrade to this initial version, we implement the game using the Athlos framework. The main objectives of this case study are to (a) study the applicability of Athlos to an existing game, and (b) determine whether Athlos has the potential to satisfy the requirements of mobile and web-based games. As a first step in this case study, we created a game definition, game model, and services using the Athlos editor, which allowed us to automatically generate a big part of the project's source code. We then implemented various extensions and features to this boilerplate code, such as database connectivity, service logic, etc., most of which were simply copied from the older version and adapted to match the framework's structure. On par with Athlos, the upgraded version utilizes PB instead of JSON for serializing data, but still communicates data to Java Servlets hosted using GAE. Even though we still utilize Memcache in the new version, we depart from using the older Cloud Datastore and upgrade to the more recent Cloud Firestore which has lower latency and better support for real-time features. For the state update mechanism, we decided to keep the long polling method as the game's requirements in terms of latency are not very strict, and this method reduces dependency on state update mechanisms provided by third parties. From this experience, we gather that Athlos can satisfy the modeling requirements of existing games like aMazeChallenge. Furthermore, we observe that Athlos is compatible with technologies used to create and host mobile and web-based games.

## *5.3. Minesweeper (MS)*

Our third case study is a multiplayer version of the puzzle game Minesweeper. We selected this particular game for several reasons. Firstly, Minesweeper has a simple, straightforward set of rules that are easy to implement. In addition, it is a 2D puzzle game that does not require the use of any complex graphics, which are out of the scope of our research. Despite having these simple requirements, it still presents a challenge on how the game can be converted from a single-player, locally hosted game into a multiplayer game that can be hosted on cloud infrastructure.

In our first implementation of Minesweeper, which was created without Athlos, we enabled multiplayer gameplay using socket technology, and the transmission of messages using JSON-formatted text. To utilize this technology it was necessary to define a custom protocol for the game from scratch so that the client and server could communicate. We allowed the server to host a selection of available games, which the clients could then join and play cooperatively at the same time. The game did not persist any information on a database but rather used the server's local memory as a cache to save data. Even though this approach is not scalable, it still allowed our barebones implementation to work as an online game. Due to the bi-directional nature of socket technology, we were able to distribute state updates back to the clients when a player made an action (i.e., revealed or flagged a cell) without the use of an additional service or API. In terms of the client, we use simple 2D graphics to represent the game state, as shown in Figure 10.

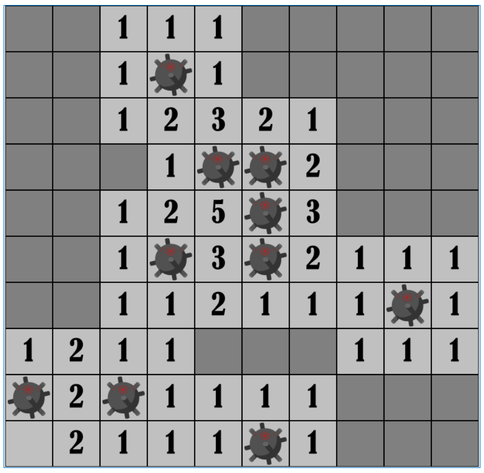


**Figure 10.** The GUI presented by the Minesweeper client.

For our study, we utilize Athlos to create a second version of Minesweeper which can be hosted on cloud IaaS infrastructure. We define the game's model and API using the project editor to create the required services. These are later implemented as gRPC services which are automatically called using remote procedure calls when a player attempts to make an action. When an action occurs, *bi-directional* gRPC streams can distribute the updated game state to all clients at very low latency, allowing for real-time gameplay. In this case we avoid using instance memory to persist game information and utilize Redis on GCP's Memorystore instead. Despite the slight decrease in performance, using Redis allows us to create a more scalable MMOG, as container orchestration platforms such as Kubernetes or Docker can be used to automatically scale the game to cope with increased demand while still having access to a shared pool of data. Consequently, this case study proves the feasibility of the dedicated/IaaS approach when utilizing Athlos and shows that it can be used to develop and deploy the backends of a variety of MMOGs.

## 6. Evaluation

The evaluation of our framework is based on two aspects. First, we conduct a performance evaluation of Athlos, which revolves around important metrics related to raw performance and scalability. Based on research questions 1–4 mentioned in Section 1, our experiments aim to reveal whether the framework can produce MMOG backends that operate under certain latency thresholds and can achieve the necessary scalability. Second, we evaluate the framework in terms of development effort and code maintainability. To this end, we attempt to explore research questions 5 and 6 by measuring the code complexity, readability, maintainability, and development effort in games that were both implemented with and without our framework.

### *6.1. Performance Evaluation*

To evaluate the performance of MMOGs developed using Athlos, we use our case study games mentioned in Section 5 and conduct a set of experiments aimed at evaluating research questions 1 through 4. Based on the related work, we use the response latency of game services to player calls as the main indicator of the performance of a backend under varying loads. This metric also encompasses other factors, such as overall processor and memory usage, network connection speed, and is also the main indicator of the players' quality of experience. In our evaluation, we distinguish between different forms of latency that occur during various phases of the data processing cycle. We define *global response latency* as the time taken for a request to be sent from a client, processed, and then a response to be received, which includes the obvious delay caused by network distance. Besides its usefulness in determining the feasibility of hosting MMOG backends on public clouds, this metric can significantly affect the evaluation of Athlos as a framework. While we still consider the global response latency as a useful metric, we also introduce a separate metric called *processing latency*, which we define as the time taken for a backend to process a request after receiving it and before sending a response. Using processing latency, we aim to evaluate the performance of the framework's code by eliminating other types of delays occurring due to network distance, load, or data serialization/deserialization overheads on the client devices.

#### 6.1.1. Global Response and Processing Latency

To measure the latency as a function of the number of active players (H1), we use a simulation based on our Athlos implementation of Mars Pioneer. We deploy the same version of the game's backend on a local machine, as well as on GCP's App Engine Flexible. From a purely theoretical standpoint, we expect that the cloud-hosted backend will suffer from higher latency at lower numbers of active players due to the relatively lower resources that instances initially employ. However, we expect this trend to reverse as the number of players increases and the locally hosted backend increasingly becomes overwhelmed while the cloud-hosted approach allocates more resources to cope with the demand. We break down processing latency measurements into five different groups, based on the following sub-processes that occur during the processing cycle:

- Session validation, which mostly entails interactions with the cache using DAOs, and verifying that a world session is valid.
- State retrieval, which firstly finds the appropriate sub-state to retrieve using algorithms and then retrieves it from the cache.
- Rule processing, which applies the game's rules for the action made by the player, based on the sub-state retrieved in the previous step.
- State modification, which modifies the game state based on the action made, by interacting with the cache or datastore.
- State sending, which distributes the updated state through the state update mechanism to the observing clients within the action's AoE.

To enable large numbers of players to play concurrently, we implement a simulation environment with bots that randomly pick actions to carry out during the game. To enable bots to start and play at the same time, we create a harness that is used to create and then run various simulation configurations. Using this harness we can adjust the number of players joining the game when each player will join, how much time they will stay in the game before leaving, the simulation phase, and the delay between moves for each individual player. While we have a variety of independent variables we can modify, we only aim to modify one of them—the total number of players—during this experiment to keep it as consistent as possible. In addition, several factors are kept in control to improve the validity of the experiment, such as device and network utilization (at or near idle), cloud data center location, database/cache type and policies used, App Engine deployment configuration, and server environment. For the local approach, we use a computer running Windows 10, with a 3rd generation Intel Core i7 processor and 16 GB of RAM, of which 10 GB are allocated for running the server. To run the simulation clients, we use a computer running Windows 10, with a 7th generation Intel Core i5 processor and 6 GB of RAM, of which 4 GB are allocated to the simulation. We also lock the experimental phase to 60 s, while we use a random delay for player moves that ranges from one to two seconds to prevent situations where all players make an action simultaneously. We conduct our experiment by creating a series of simulation configurations for varying numbers of active players ranging from 5 to 150. First, we measure the *base* global response latency over 20 calls made to an inert service for both local and cloud-based approaches to establish a controlled value for the latency of each infrastructure. Then, we run each configuration on each deployment three times, taking measurements for the sub-processes mentioned above. From these, we calculate the processing latency of each subprocess and the average processing latency of all sub-processes combined in each approach.

We found that the global response latency of the locally hosted approach was 4.9 ms, and 63.95 ms for the cloud-hosted approach. To measure the processing requirements and latency of each sub-process, we first run our experiment using the locally hosted approach. The results, shown in Table 1, firstly indicate that the total processing latency of the system grows linearly, averaging a $\times 2.07$ growth for each increase of 10 players. Secondly, we observe that the most computationally expensive operation was state sending/distribution, which consistently resulted in more latency than other services. This is justified when you consider the overheads of the serialization scheme and the state update mechanism itself. The sub-process with the second-highest latency was by far state retrieval, which was expected as this sub-process performs computationally expensive read operations from the persistence layer, as well as merging the disjoint states of cells together to form a logically connected game context. In terms of latency increase, we find that all sub-processes have a growth factor close to $\times 2$ for every increase of 10 players, with the only exception being the rule processing sub-process ($\times 1.4$). We explain this by the fact that the rule processing step is the only one that does not involve any database operations, which gradually become more expensive as the game reaches larger spatial scales. From these results, we determine that the 1000 ms latency threshold for MP, hosted using a local, dedicated machine is situated just below 40 players. Finally, we note that the simulation had to be stopped at

60 active players, as the backend started running out of resources at larger numbers and did not yield consistent results.

**Table 1.** The processing latency of each sub-process, based on the number of active players in the game, using the locally hosted approach.

| **Processing Latency (ms) for the Locally Hosted Approach** | | | | | | |
|---|---|---|---|---|---|---|
| **Active Players** | **Session Validation** | **State Retrieval** | **Rule Processing** | **State Modification** | **State Send** | **Total** |
| 5 | 1.03 | 24.11 | 0.16 | 2.89 | 43.97 | 72 |
| 10 | 0.91 | 41.26 | 0.04 | 2.39 | 101.48 | 146 |
| 20 | 1.25 | 51.63 | 0.05 | 2.95 | 113.08 | 169 |
| 30 | 3.94 | 141.36 | 0.04 | 7.33 | 399.62 | 552 |
| 40 | 16.97 | 359.08 | 0.03 | 31.99 | 808.74 | 1217 |
| 50 | 39.17 | 614.30 | 0.17 | 87.93 | 1334.77 | 1137 |
| 60 | 64.20 | 942.85 | 0.02 | 133.98 | 2091.96 | 3233 |

To compare the two approaches, we also carry out the same experiment by utilizing Google App Engine Flexible to host the same backend on the cloud. We measure the latency of the same sub-processes, and show our results in Table 2. From these, we distinguish the state retrieval sub-process as the one with the highest growth in latency ($\times 2.05$ per 10 player increase). Despite expecting database-heavy sub-processes to have a greater impact on overall latency, we admit this was an unexpected result because the persistence method utilized in this approach is hosted on the cloud. Comparing this with the processing latency of the same sub-process in the locally hosted approach ($\times 1.91$), we see that the dedicated approach fared better in terms of latency increase. Despite this unanticipated result, we must state that this is the only case in which the local approach fares better. In terms of average latency growth across all sub-processes, we calculate that the cloud-based approach fares significantly better ($\times 1.41$ per 10 player increase) compared to the local-based approach ($\times 2.07$). This rate brings it much closer to an ideal constant growth in latency as the number of active players increases, compared to the dedicated backend. In terms of absolute processing latency, the cloud-hosted backend records significantly lower latencies overall, as seen in the comparison shown in Figure 11. From this, we discover that the cloud-hosted approach reaches the 1000 ms latency threshold at much higher player numbers compared to the locally hosted approach (130+). Furthermore, we argue that these results are not fully conclusive, as the cloud-hosted approach suffers from a sudden spike in latency when no more resources are available during our experiment because of budget limitations. Despite only utilizing four App Engine instances during this experiment, we have (a) demonstrated the ability of the Athlos and serverless cloud computing layers to develop and host MMOG backends, (b) shown that cloud-hosted backends have the potential to serve much higher numbers of active players for specific latency thresholds, and (c) revealed a trend in latency to active player number ratio that may be extrapolated to much higher numbers of active players when the cloud resources are available.

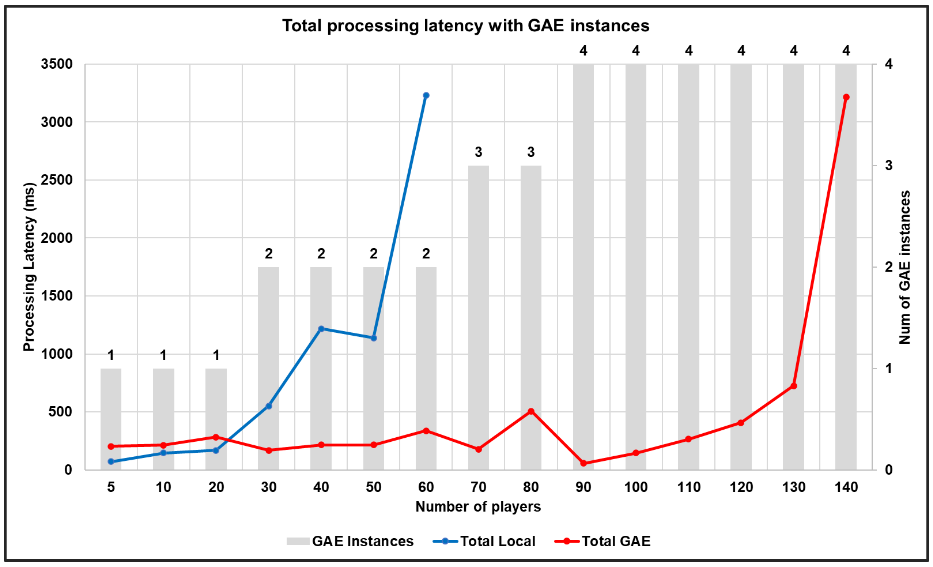


**Figure 11.** The total processing latency of local vs. cloud-hosted approaches and the number of GAE instances launched, as a function of the number of active players.

**Table 2.** The processing latency of each sub-process, based on the number of active players in the game, using the cloud-hosted approach (GAE Flexible).

| **Processing Latency (ms) for the Cloud-Hosted Approach (GAE Flexible)** | | | | | | |
|---|---|---|---|---|---|---|
| **Active Players** | **Session Validation** | **State Retrieval** | **Rule Processing** | **State Modification** | **State Send** | **Total** |
| 5 | 1.44 | 75.39 | 0.96 | 10.11 | 116.16 | 204.05 |
| 10 | 1.17 | 64.77 | 0.09 | 10.35 | 137.01 | 213.40 |
| 20 | 1.08 | 81.47 | 0.05 | 5.13 | 195.09 | 282.83 |
| 30 | 0.94 | 45.51 | 0.05 | 3.55 | 118.16 | 168.21 |
| 40 | 0.98 | 60.36 | 0.02 | 4.42 | 150.37 | 216.16 |
| 50 | 0.95 | 63.84 | 0.05 | 6.97 | 142.99 | 214.80 |
| 60 | 0.98 | 102.90 | 0.05 | 6.73 | 225.80 | 336.46 |
| 70 | 1.13 | 45.70 | 0.04 | 2.86 | 127.63 | 177.36 |
| 80 | 1.13 | 140.12 | 0.02 | 5.78 | 359.34 | 506.38 |
| 90 | 1.13 | 17.23 | 0.00 | 2.30 | 34.50 | 55.17 |
| 100 | 1.12 | 39.42 | 0.05 | 3.10 | 101.71 | 145.37 |
| 110 | 1.13 | 70.09 | 0.05 | 5.16 | 189.81 | 264.19 |
| 120 | 1.15 | 112.88 | 0.02 | 26.37 | 286.95 | 406.17 |
| 130 | 3.88 | 216.90 | 0.05 | 26.37 | 479.18 | 726.38 |
| 140 | 1.56 | 2342.08 | 0.03 | 105.50 | 765.17 | 3214.33 |

### 6.1.2. Scalability

In addition to the experiments in the previous section which were primarily focused on latency, we conduct another set of experiments that explore the ability of our framework to produce scalable MMOG backends. We inspect scalability from three different aspects: (a) runtime scalability, which was explored in terms of latency in the previous section, (b) state scalability, and (c) bandwidth consumption. We define *state scalability* as the ability of an MMOG to grow or shrink its state based on context. For instance, games may have to upscale their states when new players join the game, mostly by creating new entities for each player or by extending the world's terrain. On the other hand, games may also need to downscale their states under several circumstances, such as when players leave the game.

Our first scalability experiment measures the maximum game state possible when utilizing our framework and a cloud-based NoSQL datastore. Using Mars Pioneer (cloud-based, using Athlos) and Minesweeper for control and comparison (cloud-based, not using Athlos), we measure how many chunks can be generated and stored for a single world, allowing us to observe how many chunks can be loaded before the system is overwhelmed. Then, we measure the size of the state of a single terrain cell to make comparisons between the size of the state in each approach. To achieve this, we use an algorithm designed to request parts of the terrain at various positions in the game world. After the chunks are created or the persistence option used reaches its limitations, we measure the maximum number of chunks each approach and game can support. In the first stage of this experiment, we measure the absolute size of each terrain cell for both games. For Mars Pioneer, this stands at 16 bytes for each cell, whereas for Minesweeper the size is only 5 bytes. Naturally, the state of MP cells is significantly larger than MS, because the game features more complex gameplay. Using our terrain generation algorithm, we obtain part of the terrain at various locations in both games which allows us to measure the maximum possible state in cells for both approaches. For Mars Pioneer, which utilizes Athlos and Google's Firestore for persistence, the algorithm managed to generate 110,240 cells. On the other hand, Minesweeper, which did not utilize Athlos, was able to generate only 52,441 cells despite having a far smaller cell size. More importantly, we report that the cell generation process in Mars Pioneer was stopped because of Firestore quota limitations and not because of space, or framework limitations. Therefore, we argue that the generation may have continued unhindered to even larger states if more resources were available, and claim that MMOGs utilizing Athlos are indeed able to feature very large, expandable game states within the confines of the resources available.

As an extension of this experiment, we explore how the Athlos framework creates and saves a `TerrainIdentifier` entity with each chunk that is being created. Terrain identifiers are used to index chunks, allowing them to be easily identified, queried, and retrieved without having to retrieve the terrain state itself—which is a very costly operation. Despite their usefulness, terrain identifiers add an absolute overhead of 7.25 bytes per cell which must be taken into consideration in our evaluation. Secondly, they require an additional write operation in the corresponding datastore. For the Mars Pioneer backend, generated cells were grouped in 6890 chunks, which means that the same number of identifiers had to be created. This effectively doubles the number of items being written to the datastore during this experiment to 13,780. For massive states like these, using terrain identifiers has a positive impact on performance, especially given the limitations of several key-value datastores or caches in terms of query support and filtering. On the other hand, it is possible to avoid using terrain identifiers to avoid these overheads for games that are not expected to reach such large terrain scales.

Guided by the capabilities of Athlos in terms of scalability, we conduct another experiment that aims to explore how the time taken to retrieve a sub-state of a world changes with respect to the world's size. Using our implementation of Mars Pioneer, we measure the time taken to load a *pre-generated*, not previously loaded, $16 \times 16$ chunk, under different full state sizes ranging from 1 to 1024 chunks. In this experiment, we keep several factors in control, including the number of App Engine instances running, data center location, as well as game-specific items such as the size of a cell's state. The time taken is recorded in terms of milliseconds from the moment the chunk is requested *by the backend* until it is retrieved. To maximize the validity of our results, we run our experiment three times for each different full state size and record the averages, which are presented in Figure 12.

Based on the results obtained, we observe that as the size of the full state grows exponentially, the time taken to load a single chunk has a constant order of growth, and averages a time of 33.72 ms. Therefore, our hypothesis, which is based on the assumption that Athlos cells and chunks use maps and hash values to access the terrain in constant time seems to be confirmed—the time taken to retrieve a subset of the state remains constant regardless of the full state's size. Consequently, we argue that Athlos supports the development of MMOG backends that feature expandable and scalable terrain states that can be accessed in constant time regardless of their full size.

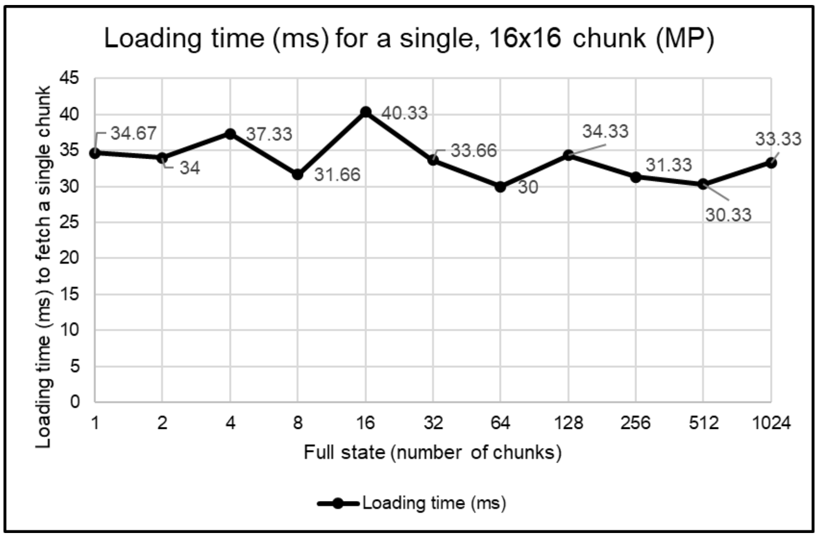


**Figure 12.** The amount of time taken to load a single 16 × 16 chunk, in milliseconds, as the full size of the full state changes.

Furthermore, we study the effects of chunk sizes on the number of queries needed to fetch the entire state and the latency of generating and retrieving it. Our goal is to determine which chunk size achieves the best performance vs. query number balance, as there is a trade-off between the two. Initially, we use Mars Pioneer and request 1000 individual cells at positions (0,0) to (0,999) inclusive. We measure the effect of various chunk sizes on the number of queries required to fetch the game state mentioned, as well as the latency of generating and then retrieving the state for use in the backend. We vary the chunk sizes ranging from 4 × 4 (16 cells) to 64 × 64 (4096 cells) while keeping the cell state, and other underlying factors in control. The results, shown in Figure 13, compare the number of queries needed to fetch the state against the time taken to generate these cells. We notice that as the chunk size increases, the number of queries required to fetch the state is reduced, whereas the opposite occurs for the time taken to generate the necessary chunks. Based on the data, we discover that the size 16 × 16 is an equilibrium between the number of queries and the generation latency. In other words, this specific size happens to offer an ideal condition where both the number of queries and the generation latency are at relatively favorable values. As a consequence of this, MMOGs utilizing this size for their chunks may benefit from both a relatively low number of queries and a decreased generation latency. Guided by these results, we set the 16 × 16 size as the default terrain size in every Athlos project, but still allow developers to change the size of chunks within the range 4–64. The reasoning behind this is that some games may benefit from having either a low or high chunk size. For instance, games that only have to load terrain once, or very few times during gameplay, may benefit from a large chunk size (i.e., 32 or 64) to reduce the number of queries. On the other hand, games that need to refresh terrain continuously and at very low latencies will certainly benefit from lower chunk sizes (i.e., 4 or 8).

To measure the effects of our framework on the bandwidth consumption of MMOGs, we use our two implementations of aMazeChallenge. To find the bandwidth requirements of each version, we measure the size of the game state in terms of bytes. One of the main differences between these two versions of aMazeChallenge is the method used to serialize the game state. For the initial (non-Athlos) version, JSON was used to serialize the state, whereas the newer (Athlos) version, uses Protocol Buffers. As game state communication is the most frequently occurring process in an MMOG, measuring the size of the state should be a direct indicator of its bandwidth requirements. To measure the state size, we create mazes of different sizes in both versions of the game, ranging from 5 × 5 to 30 × 30 cells. We keep several properties in control, such as the fact that there are no pickable objects or players within the state, and that the mazes must be of the same type. Firstly, we measure the state size of the non-Athlos implementation by recording the byte size of the JSON-formatted string produced by a `Grid` object. For the Athlos-based implementation, we serialize a Grid protocol buffer object into bytes and record their size. From the results, shown in Figure 14, we find out that the Athlos implementation has a significantly lower state size for all maze sizes and therefore lower bandwidth requirements. In comparison with the non-Athlos approach that utilizes JSON, we see that Protocol Buffers can serialize the game state in as little as 7% of the size used by the JSON format in the 5 × 5 maze. While the efficiency of protocol buffers is unparalleled, we observe a decreasing trend as the size of the mazes grows larger. For instance, in the 30 × 30 maze, the Athlos approach used 58% of the size of the original version. We argue that this example is in fact a favorable condition for the JSON version, as the size of the state only grows at a linear pace. Nevertheless, as shown from the bytes per cell columns in Figure 14, there are no circumstances under which PB would be less efficient than JSON. Based on our results, we believe that MMOGs developed using Athlos—and its extensions and abstractions in terms of utilizing Protocol Buffers—can greatly benefit from reduced bandwidth requirements and ultimately better economy over time.

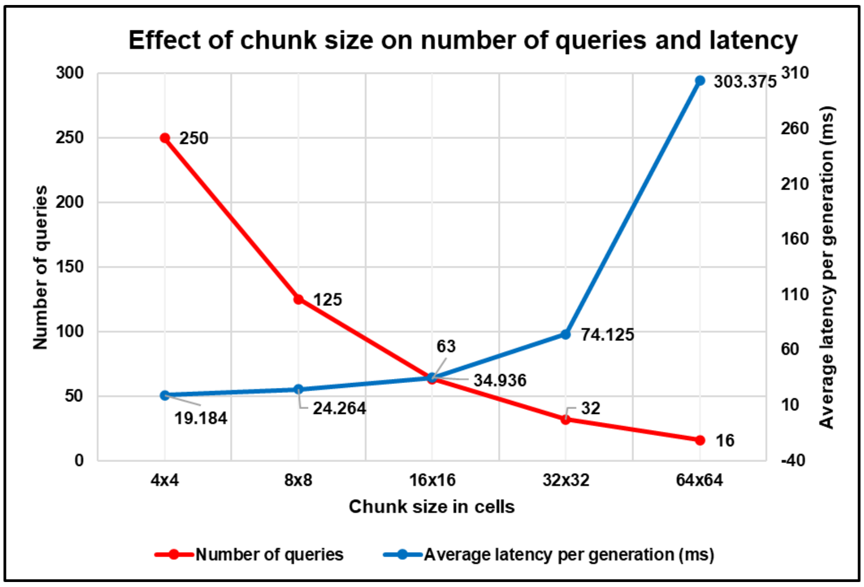


**Figure 13.** The effect of chunk size on the number of queries required to fetch the game state vs. the time taken to generate the chunks.

## 6.2. Development Effort and Code Maintainability

The second aspect of our evaluation deals with the effort needed to create MMOG backends in terms of software engineering, and the quality and maintainability of the code produced by our framework. To study the effects of using Athlos on the effort required to develop a scalable MMOG, we measure the Lines of Code (LOC) created in our implementation of Minesweeper. For comparison, we also count the LOC in a previous implementation of the game that did not utilize our framework. We separate the lines counted in two categories: those which were automatically *generated*, and those that had to be written manually (i.e., *efforted*) by the developer. In addition, we only include source code files—thus omitting any project and configuration files, Athlos definitions, or other resources. Any source code that is not related to the game's implementation, such as the code written for simulation and testing purposes does not count towards the line count. Finally, we also omit source code produced by the Protocol Buffers compiler, even though this code is actively used in the game implementations. We argue that the omission of these files makes the comparison more fair, as Protocol Buffers is considered a separate library that could be used in an approach that does not utilize Athlos. As seen from our results in Figure 15, the total LOC in the Athlos-based version is 3628 and 2355 in the non-Athlos version. We attribute this difference in total LOC to the fact that Athlos implementations include a more diverse set of functionality than that required by a specific game, and especially one that is as simple as Minesweeper. However, when separating the LOC into the two aforementioned categories, we observe that the lines generated in the Athlos implementation greatly exceed those that were efforted. In fact, comparing just the efforted LOC between the two projects reveals that the Athlos implementation required only a third (32.6%) of the lines of code to be written by the developer compared to the non-Athlos implementation. Based on this comparison, we deduce that even though the absolute project size in terms of LOC may be significantly bigger when utilizing Athlos, the relative effort required to produce it may be far lower than when Athlos is not used.

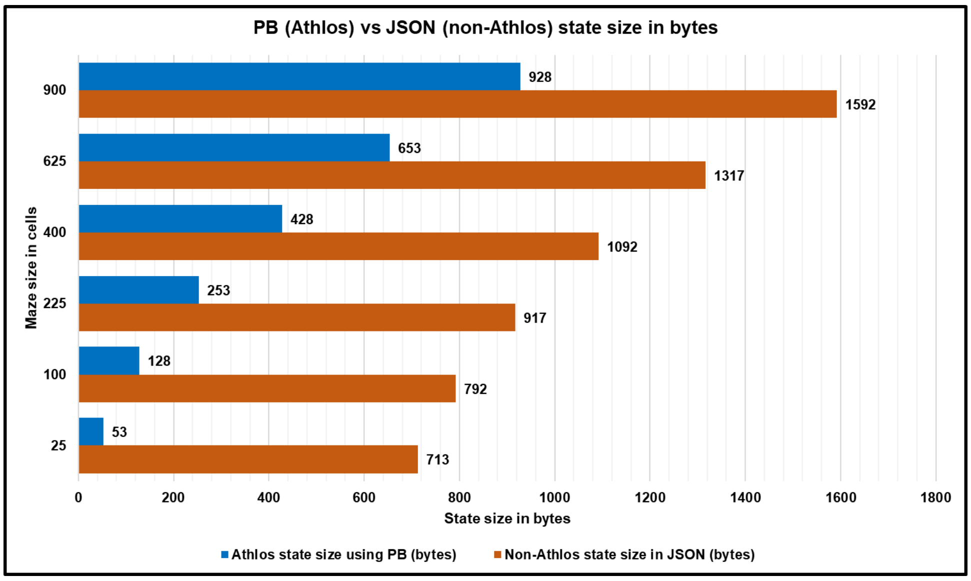


**Figure 14.** A comparison between JSON (non-Athlos) and Protocol Buffers (using Athlos) in terms of the number of bytes required to serialize the game state.

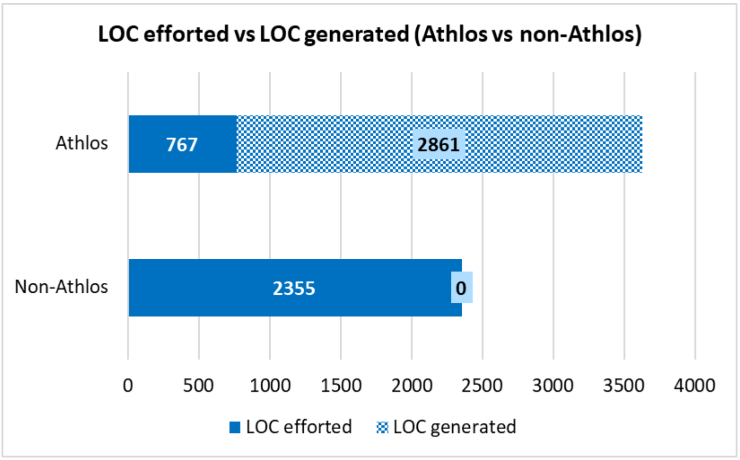


**Figure 15.** A comparison between the LOC generated and the LOC efforted for the Athlos and non-Athlos implementations of Minesweeper.

Furthermore, we evaluate the quality of the code produced in several games that use our framework using Chidamber–Kemerer (CK) metrics [52]. CK metrics can be utilized to explore the quality of software in terms of object-oriented design. To measure the quality of code, we use two of the games we produced using the Athlos framework—Minesweeper and aMazeChallenge. We compare the values of the CK metrics measured for these Athlos-based MMOGs with their non-Athlos counterparts, which act as controls, in order to enable a comparison between these approaches. To help us take these measurements, we utilized a plugin called MetricsReloaded, which is available in JetBrains-based IDEs such as IntelliJ IDEA and Android Studio. The plugin automatically measures CK metrics and then produces reports which can be exported or analyzed directly. From these measurements, we exclude several files, such as source code written for simulation and testing purposes, as well as files produced by Protocol Buffers. For our evaluation, we measure the following CK metrics, which are inversely proportional to the code quality (i.e., a lower value is considered advantageous):

- Coupling Between Objects (CBO), which calculates the number of classes/interfaces that each class is coupled with.
- Depth of Inheritance Tree (DIT), which is the maximum distance from the given class to the root node of the inheritance tree.
- Lack of Cohesion Methods (LCOM), which counts the degree of cohesiveness in a class based on the number of methods that are disjoint.
- Number of Children (NOC) metric, which is the number of sub-classes associated with the class.
- Response for Class value (RFC), which is a set of methods that can be called in response to a message that is received by an instance of the class.
- Weight Methods per Class (WMC) metric, which is the sum of the complexity of all methods in a given class.

Our results, shown in Tables 3 and 4, compare the Athlos and non-Athlos based approaches using CK metrics to measure the quality of code. In Minesweeper, we observe that the non-Athlos approach is clearly ahead in terms of code quality compared to our framework's implementation. Especially in terms of LCOM, RFC, and WMC, the non-Athlos implementation appears to have a higher degree of cohesiveness in terms of the class methods, make fewer procedure calls per message received in each class, and have reduced complexity compared to the framework approach. On the other hand, the results from comparing the aMazeChallenge implementations are rather inconclusive, as both approaches seem to be advantageous in some regards, but not so in others. The Athlos-based implementation has better DIT, LCOM, and NOC properties, which means that the class hierarchy is slightly simpler, and there is better cohesiveness in terms of the class methods. On the other hand, the code in this approach has worse cyclomatic complexity (WMC), produces more procedure calls per class message (RFC), and is more intertwined (CBO).

**Table 3.** A comparison between two Minesweeper implementations (Athlos vs. non-Athlos) in terms of the quality of the code produced, based on CK metrics (lower is better).

| | **Minesweeper Athlos** | **Minesweeper Non-Athlos** |
|---|---|---|
| **CBO** | 8.95 | 7.27 |
| **DIT** | 1.33 | 1.91 |
| **LCOM** | 3.14 | 1.95 |
| **NOC** | 0.00 | 0.26 |
| **RFC** | 22.05 | 13.98 |
| **WMC** | 13.74 | 10.17 |

**Table 4.** A comparison between two aMazeChallenge implementations (Athlos vs. non-Athlos) in terms of the quality of the code produced, based on CK metrics (lower is better).

| | **aMazeChallenge Athlos** | **aMazeChallenge Non-Athlos** |
|---|---|---|
| **CBO** | 8.91 | 6.47 |
| **DIT** | 2.20 | 2.33 |
| **LCOM** | 2.23 | 2.36 |
| **NOC** | 0.01 | 0.08 |
| **RFC** | 25.39 | 17.45 |
| **WMC** | 14.04 | 11.92 |

These results provide some guidance for improvement, despite some limitations. Based on the data, we acknowledge that the framework could be improved so that the code produced in game implementations is more efficient, less complex, more reusable and maintainable, and ultimately, of higher quality. Limitations that may have skewed the results of this experiment are the fact that during the development of these case studies there were no provisions or considerations made regarding code quality. Secondly, the code produced by the framework is designed to provide developers with convenience in accessing various aspects of the architecture, rather than being aimed at improving quality metrics. Nevertheless, the results indicate several points where the framework could be improved to produce more readable, and more efficient code.

## 7. Conclusions and Future Work

In this paper, we described the models, methods, and tools with which scalable MMOG backends can be developed for and deployed on commodity cloud platforms. To enable developers to utilize our approach, we have presented Athlos, a framework that enables the rapid development and deployment of scalable MMOG backend prototypes. Even though our study primarily focuses on a specific set of games and deployment options—mostly within serverless computing layers—our approach also covers a variety of other games and deployment options. Through Athlos, we aim to enhance the development process for this type of application by abstracting key concepts that are similar across games, such as networking, serialization, service implementation, persistence, and so on. Furthermore, through our model, we attempt to conceptualize the main items that are common in a variety—if not all—games, and therefore reduce the development effort by allowing code reuse and automatic code generation. Developers can take advantage of our framework's modular architecture to design MMOG backends that can work with a variety of interchangeable components. By using this modular approach, we foster experimentation with different technologies, which empowers game developers to develop their games for a variety of platforms and use the most optimal tools and technologies for their games. We hope that this leads to more innovative solutions in terms of development processes, and subsequently, higher-quality MMOG backends. To showcase the use of our framework, and to provide proof-of-concept in terms of its usability, we discuss several case studies that provide examples of how the framework can be utilized to quickly prototype backends for certain types of MMOGs.

We evaluate our framework using a set of experiments primarily focused on performance and development effort. In terms of performance, we use cloud-hosted and locally hosted versions of our case study MMOGs to study the effects of utilizing our framework and commodity clouds on the latency as backends scale to larger numbers of players. We found that by utilizing Athlos in conjunction with a commodity cloud, we were able to significantly improve the runtime scalability of an MMOG backend. This enables it to support a significantly larger number of players under a given latency threshold. Although our experiment is limited in terms of the cloud resources available, it shows a clear trend that may continue in cases where such resources are available. Therefore, we believe that our results can be extrapolated to larger, industrial scales, and are a clear indicator of the advantages brought forth by Athlos and serverless computing.

In addition, we conduct a state scalability and bandwidth usage evaluation of our framework by attempting to measure (a) the maximum possible state of a world when utilizing our framework, (b) the loading time of chunks compared to the full state of the game, (c) the effects of chunk size on the number of queries and latency, and (d) the bandwidth usage of games developed using our framework. Throughout our experiments, we compare the same backends developed with and without Athlos and compare the results. From these, we learn that Athlos can support extremely large game states when used in conjunction with cloud-based, NoSQL datastores. In fact, our experience with this investigation suggests that the limiting factor is resource availability. In terms of loading time, our results show that the time taken to load a piece of the entire state remains constant regardless of the entire state's size. Coupled with the ability to upscale worlds to massive sizes, this allows MMOGs that are constructed using Athlos to retain the same performance regardless of world size. Our explorations with chunk sizes also reveal that there may be a variety of configurations that can be beneficial for different types of games. Based on our results, we have modified our framework to allow developers to change the chunk size on a per-game basis. This allows different types of MMOGs to be either more economical or have better performance, depending on the requirements. Furthermore, by evaluating the bandwidth requirements of Athlos, we find that MMOGs developed using our framework may benefit from reduced bandwidth consumption due to the use of Protocol Buffers, especially when compared to other, commonly used serialization methods.

To evaluate the development effort required to create a game with Athlos, we measure the lines of code generated against the lines of code written manually (efforted). We compare results from an Athlos based approach against an approach that does not utilize Athlos. As the data suggests, even though Athlos-based projects have a significantly bigger size in terms of total lines of code, the vast majority of that code is automatically generated by our project generator. As a result, the development effort is significantly reduced when utilizing our framework, to the point where our approach only requires a small fraction of the lines to be efforted compared to other approaches. Finally, we evaluate two of our case studies in terms of code maintainability. By using Chidamber–Kemerer metrics, we assess the quality of the produced code. Our results show that the code in Athlos-based projects has either the same, or slightly worse quality when compared to manually coded projects, which suggests that there is room for improvement in terms of the automatically generated code.

### *Future Work*

In the future, we aim to improve the structure of the code generated by Athlos. We expect that this will improve the readability, maintainability, and overall quality of the code in Athlos-based projects. We also aim to create a domain-specific game definition language for Athlos, which can be used to create game definitions more quickly and perhaps within specific development environments. In addition, such a language may be used in conjunction with our existing development tools, enhancing their functionality or expediting the game definition process where needed. In terms of evaluating our framework, we aim to extend several experiments to higher scales once more resources are available. For instance, we would like to explore how serverless backends respond to even higher numbers of players (i.e., in the order of thousands) when used in conjunction with our framework. Additionally, we aim to evaluate our framework qualitatively, via feedback from developers who utilize Athlos. Such an experiment could be used to measure further quantities, such as the time taken for the participants to complete individual tasks, the number, and type of issues faced during the development process, the perceived amount of effort exerted, etc. Finally, we plan to explore a *hybrid* architecture approach, in which the features of dedicated or IaaS approaches can be interweaved with those of serverless computing, to allow developers to leverage the advantages of both approaches. By further studying and understanding the processes involved in the development of scalable MMOG backends, and by devising other practical solutions, we aim to further expedite and simplify their development processes.

**Author Contributions:** N.K. has contributed to the paper's related work, methodology, experimental design, evaluation, and write-up. N.P. has contributed to the research problem's definition, methodology, and the reviewing process. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Data Availability Statement:** The raw data published in this study can be shared upon request by contacting the authors at nkasenides@uclan.ac.uk.

**Acknowledgments:** We thank Google for offering USD 1000 worth of credits for their Google Cloud Platform. Their contribution helped us during the development and evaluation of our framework.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Morgan, L.; Conboy, K. Key factors impacting cloud computing adoption. *Computer* **2013**, *46*, 97–99. [CrossRef]
2. Buyya, R.; Srirama, S.N.; Casale, G.; Calheiros, R.; Simmhan, Y.; Varghese, B.; Gelenbe, E.; Javadi, B.; Vaquero, L.M.; Netto, M.A.; et al. A manifesto for future generation cloud computing: Research directions for the next decade. *ACM Comput. Surv. (CSUR)* **2018**, *51*, 1–38. [CrossRef]
3. Boillat, T.; Legner, C. Why do companies migrate towards cloud enterprise systems? A post-implementation perspective. In Proceedings of the 2014 IEEE 16th Conference on Business Informatics, Geneva, Switzerland, 14–17 July 2014; Volume1, pp. 102–109.
4. Chuang, W.C.; Sang, B.; Yoo, S.; Gu, R.; Kulkarni, M.; Killian, C. Eventwave: Programming model and runtime support for tightly-coupled elastic cloud applications. In Proceedings of the 4th Annual Symposium on Cloud Computing, Santa Clara, CA, USA, 1–3 October 2013; pp. 1–16.
5. Nae, V.; Prodan, R.; Fahringer, T. Cost-efficient hosting and load balancing of massively multiplayer online games. In Proceedings of the 2010 11th IEEE/ACM International Conference on Grid Computing, Brussels, Belgium, 25–28 October 2010; pp. 9–16.
6. Nae, V.; Iosup, A.; Prodan, R. Dynamic resource provisioning in massively multiplayer online games. *IEEE Trans. Parallel Distrib. Syst.* **2010**, *22*, 380–395. [CrossRef]
7. Burger, V.; Pajo, J.F.; Sanchez, O.R.; Seufert, M.; Schwartz, C.; Wamser, F.; Davoli, F.; Tran-Gia, P. Load dynamics of a multiplayer online battle arena and simulative assessment of edge server placements. In Proceedings of the 7th International Conference on Multimedia Systems, Klagenfurt, Austria, 10–13 May 2016; pp. 1–9.
8. Ducheneaut, N.; Yee, N.; Nickell, E.; Moore, R.J. Building an MMO With Mass Appeal: A Look at Gameplay in World of Warcraft. *Games Cult.* **2006**, *1*, 281–317. [CrossRef]
9. Hosseini, M. A Survey of Bandwidth and Latency Enhancement Approaches for Mobile Cloud Game Multicasting. *arXiv* **2017**, arXiv:1707.00238.
10. Barri, I.; Roig, C.; Giné, F. Distributing game instances in a hybrid client-server/P2P system to support MMORPG playability. *Multimed. Tools Appl.* **2016**, *75*, 2005–2029. [CrossRef]
11. Tsipis, A.; Komianos, V.; Oikonomou, K. A Cloud Gaming Architecture Leveraging Fog for Dynamic Load Balancing in Cluster-Based MMOs. In Proceedings of the 2019 4th South-East Europe Design Automation, Computer Engineering, Computer Networks and Social Media Conference (SEEDA-CECNSM), Piraeus, Greece, 20–22 September 2019; pp. 1–6.
12. Kasenides, N.; Paspallis, N. A Systematic Mapping Study of MMOG Backend Architectures. *Information* **2019**, *10*, 264. [CrossRef]
13. Google. Firebase for games | Supercharge Your Games with Firebase. Available online: https://firebase.google.com/games. (accessed on 25 January 2022)
14. Shabani, I.; Kovaçi, A.; Dika, A. Possibilities offered by Google App Engine for developing distributed applications using datastore. In Proceedings of the 2014 Sixth International Conference on Computational Intelligence, Communication Systems and Networks, Tetova, Macedonia, 27–29 May 2014; pp. 113–118.
15. Dhib, E.; Boussetta, K.; Zangar, N.; Tabbane, N. Modeling Cloud gaming experience for Massively Multiplayer Online Games. In Proceedings of the Consumer Communications & Networking Conference (CCNC), Las Vegas, NV, USA, 9–12 January 2016; pp. 381–386.
16. Dhib, E.; Zangar, N.; Tabbane, N.; Boussetta, K. Resources allocation trade-off between cost and delay over a distributed Cloud infrastructure. In Proceedings of the 2016 7th International Conference on Sciences of Electronics, Technologies of Information and Telecommunications (SETIT), Hammamet, Tunisia, 18–20 December 2016; pp. 486–490.

17. GauthierDickey, C.; Zappala, D.; Lo, V. Distributed Architectures for massively multiplayer online games. In Proceedings of the ACM NetGames Workshop, Portland, OR, USA, 30 August 2004.
18. Jardine, J.; Zappala, D. A hybrid architecture for massively multiplayer online games. In Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games, Worcester, MA, USA, 21–22 October 2008; pp. 60–65.
19. Plumb, J.; Kasera, S.; Stutsman, R. Hybrid network clusters using common gameplay for massively multiplayer online games. In Proceedings of the 13th International Conference on the Foundations of Digital Games, Malmo, Sweden, 7–10 August 2018; pp. 1–10. [CrossRef]
20. Nae, V.; Prodan, R.; Iosup, A. Massively multiplayer online game hosting on cloud resources. *Cloud Comput. Princ. Paradig.* **2011**, 491–509.
21. Nae, V.; Prodan, R.; Fahringer, T.; Iosup, A. The impact of virtualization on the performance of massively multiplayer online games. In Proceedings of the 8th Annual Workshop on Network and Systems Support for Games, Paris, France, 23–24 November 2009; p. 9.
22. Assiotis, M.; Tzanov, V. A distributed architecture for MMORPG. In Proceedings of the 5th ACM SIGCOMM Workshop on Network and System Support for Games, Singapore, 30–31 October 2006; p. 4.
23. Negrão, A.P.; Veiga, L.; Ferreira, P. Task based load balancing for cloud aware massively Multiplayer Online Games. In Proceedings of the 2016 IEEE 15th International Symposium on Network Computing and Applications (NCA), Cambridge, MA, USA, 31 October–2 November 2016; pp. 48–51.
24. El Rhalibi, A.; Al-Jumeily, D. Dynamic Area of Interest Management for Massively Multiplayer Online Games Using OPNET. In Proceedings of the 2017 10th International Conference on Developments in eSystems Engineering (DeSE), Paris, France, 14–16 June 2017; pp. 50–55.
25. Kavalionak, H.; Carlini, E.; Ricci, L.; Montresor, A.; Coppola, M. Integrating peer-to-peer and cloud computing for massively multiuser online games. *Peer-Peer Netw. Appl.* **2015**, *8*, 301–319. [CrossRef]
26. Kim, J.H. P2P Systems based on Cloud Computing for Scalability of MMOG. *J. Inst. Internet Broadcast. Commun.* **2021**, *21*, 1–8.
27. Chu, H.S. Building a Simple Yet Powerful Mmo Game Architecture, Part 2: Gaming and Web Integration. 2008. Available online: http://www.360doc.com/content/09/0115/20/28217_2341386.shtml (accessed on 25 January 2020).
28. Shaikh, A.; Sahu, S.; Rosu, M.C.; Shea, M.; Saha, D. On demand platform for online games. *IBM Syst. J.* **2006**, *45*, 7–19. [CrossRef]
29. Doddavula, S.K.; Gawande, A.W. Adopting cloud computing: Enterprise private clouds. *Setlabs Brief.* **2009**, *7*, 11–18.
30. Mishra, D.; El Zarki, M.; Erbad, A.; Hsu, C.H.; Venkatasubramanian, N. Clouds+ games: A multifaceted approach. *IEEE Internet Comput.* **2014**, *18*, 20–27. [CrossRef]
31. Satyanarayanan, M.; Bahl, V.; Caceres, R.; Davies, N. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Comput.* **2009**, *8*, 14–23. [CrossRef]
32. Najaran, M.T.; Krasic, C. Scaling online games with adaptive interest management in the cloud. In Proceedings of the 2010 9th Annual Workshop on IEEE Network and Systems Support for Games (NetGames), Taipei, Taiwan, 16–17 November 2010; pp. 1–6.
33. Zahariev, A. *Google App Engine*; Helsinki University of Technology: Espoo, Finland, 2009; pp. 1–5.
34. Lu, F.; Parkin, S.; Morgan, G. Load balancing for massively multiplayer online games. In Proceedings of the 5th ACM SIGCOMM Workshop on Network and System Support for Games, Singapore, 30–31 October 2006; p. 1.
35. Brewer, E. Spanner, Truetime and the Cap Theorem. 2017. Available online: https://research.google/pubs/pub45855/ (accessed on 25 January 2020).
36. Vogels, W. Eventually consistent. *Commun. ACM* **2009**, *52*, 40–44. [CrossRef]
37. Blackman, T.; Waldo, J. *Scalable Data Storage in Project Darkstar*; Sun Microsystems, Inc.: Mountain View, CA, USA, 2009.
38. Photon Engine. Photon Unity 3D Networking Framework SDKs and Game Backend. 2022. Available online: https://www.photonengine.com/PUN (accessed on 3 March 2022).
39. Google. Stadia: Take Game Development Further Than You thought Possible. 2019. Available online: https://stadia.dev/about. (accessed on 6 March 2022).
40. Freiknecht, J.; Effelsberg, W. A survey on the procedural generation of virtual worlds. *Multimodal Technol. Interact.* **2017**, *1*, 27. [CrossRef]
41. Kasenides, N.; Paspallis, N. Multiplayer game backends: A Comparison of commodity cloud-based approaches. In *European Conference on Service-Oriented and Cloud Computing*; Springer: Cham, Switzerland, 2020; pp. 41–55.
42. Plumb, J.N.; Stutsman, R. Exploiting Google's Edge Network for Massively Multiplayer Online Games. In Proceedings of the 2018 IEEE 2nd International Conference on Fog and Edge Computing (ICFEC), Washington, DC, USA, 1–3 May 2018; pp. 1–8.
43. Rossi, F.; Cardellini, V.; Presti, F.L.; Nardelli, M. Geo-distributed efficient deployment of containers with Kubernetes. *Comput. Commun.* **2020**, *159*, 161–174. [CrossRef]
44. Lundgren, J. Kubernetes for Game Development: Evaluation of the Container-Orchestration Software. 2021. Available online: https://www.diva-portal.org/smash/get/diva2:1562637/FULLTEXT01.pdf (accessed on 25 January 2022).
45. Kurniawan, B. *Java for the Web with Servlets, JSP, and EJB*; Sams Publishing: Clay Township, IN, USA, 2002.
46. Briceno, L.D.; Siegel, H.J.; Maciejewski, A.A.; Hong, Y.; Lock, B.; Panaccione, C.; Wedyan, F.; Teli, M.N.; Zhang, C. Resource allocation in a client/server system for massive multi-player online games. *IEEE Trans. Comput.* **2013**, *63*, 3127–3142. [CrossRef]
47. Feng, J.; Li, J. Google protocol buffers research and application in online game. In Proceedings of the IEEE Conference Anthology, China, 1–8 January 2013, pp. 1–4. [CrossRef]

48. Wang, X.; Zhao, H.; Zhu, J. GRPC: A communication cooperation mechanism in distributed systems. *ACM SIGOPS Oper. Syst. Rev.* **1993**, *27*, 75–86. [CrossRef]
49. Jia, Z.; Witchel, E. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices Extended Abstract. Avaiable online: https://dl.acm.org/doi/abs/10.1145/3445814.3446701 (accessed on 25 January 2022).
50. Google. Usage and Limits, Firebase Documentation. 2022. Available online: https://firebase.google.com/docs/firestore/quotas (accessed on 3 March 2022).
51. Kasenides, N.; Paspallis, N. aMazeChallenge: An Interactive Multiplayer Game for Learning to Code. In Proceedings of the 29th International Conference on Information Systems Development (ISD2021), Valencia, Spain, 8–10 September 2021.
52. Kumar, R.; Kaur, G. Comparing complexity in accordance with object oriented metrics. *Int. J. Comput. Appl.* **2011**, *15*, 42–45. [CrossRef]