Multiplayer Game Backends: A Comparison of Commodity Cloud-Based Approaches

Nicos Kasenides, Nearchos Paspallis

Multiplayer game backends: A Comparison of commodity cloud-based approaches

Nicos Kasenides[0000−0002−1562−3839] and
Nearchos Paspallis[0000−0002−2636−7973]

University of Central Lancashire—Cyprus Campus - 12-14 University Av. 12-14
CY-7080 {nkasenides,npaspallis}@uclan.ac.uk

Abstract. The development of resource-intensive complex distributed systems such as the backend side of Massively Multiplayer Online Games (MMOGs) has shifted towards cloud-based approaches in recent years. Despite this shift, researchers and developers have mostly utilized proprietary clouds to provide services for such applications — thus leaving the area of commodity clouds largely unexplored. The use of proprietary clouds is almost always applied at the Infrastructure-as-a-Service layer, thereby enforcing restrictions on the development of MMOGs. In a previous work we focused on the characteristics of MMOGs, outlining certain factors that prohibit their deployment on commodity clouds. In this paper, we evaluate the suitability of common public cloud platforms in developing and deploying the backend side of MMOGs. In our approach, we implement a simple MMOG over three popular public cloud platforms. Then, we evaluate their performance by measuring the latency of the game over each platform as well as the maximum size of game worlds supported by each approach. Our measurements show that approaches based on the Infrastructure-as-a-Service layer perform better than those based on the Platform-as-a-Service layer — which was expected. However, our results indicate that MMOGs based on the Platform-as-a-Service layer can also perform relatively well and within the bounds of real-time latency. Coupled with accelerated development and lower maintenance costs, Platform-as-a-Service technology paves the way for further development of MMOG specific Backend-as-a-Service platforms.

Keywords: Software engineering · Distributed systems · Cloud computing · MMOG · Backend · Commodity clouds

1 Introduction

The use of commodity cloud platforms to power enterprise applications has become the default choice in recent years. Cloud computing offers numerous advantages, most notably scalability, elasticity, and cost efficiency [6]. Despite their scale, enterprise applications exhibit moderate synchronization requirements that rarely cause any significant issues with their scalability. On the other hand, resource-intensive applications such as *Multiplayer Online Games*

(MOGs) and especially *Massively Multiplayer Online Games* (MMOGs) typically limit their scale using game-imposed constraints — such as game "rooms" with specific capacities — to cope with the very high resource demands. Such games have traditionally pushed the limits of cloud computing: they have certain peculiarities and present a different set of challenges that must be tackled before they can be enabled to run on commodity clouds [16, 18, 24, 37]. In the past, the resource-intensive nature of such applications has led game providers to opt for on-premise rather than for public cloud solutions to host their game's backend [13, 38, 45]. However, trends emerging from a recent study we conducted [28] show that the use of cloud technology has become the most popular option for deploying MMOGs, and also that it is moving towards higher abstraction layers such as *Platform-as-a-Service* (PaaS).

In this paper, we assess how commercial-grade, public cloud solutions can be used to realize the backends of MOGs and MMOGs. While *Backend-as-a-Service* (BaaS) technology has already been used for secondary functionalities — such as analytics, score-keeping, etc. — we investigate how PaaS-based backends can be used for core tasks that cover the state management and core operations typically placed at the backend. We believe there is an opportunity to utilize these higher layers of cloud computing to provide inherent scalability, offer higher abstraction during development and decrease maintenance costs for game producers.

Our approach realizes a simple MOG which is implemented and tested on top of commodity cloud services at the *Infrastructure-as-a-Service* (IaaS) and PaaS layers. By implementing the game in multiple layers, we aim to identify some of the constraints, peculiarities, and challenges presented when moving from on-premise solutions to private clouds and then to public clouds. Our main objective is to allow a comparison between these approaches, based on their performance. Performance — frequently measured in terms of latency — significantly affects the *Quality of Experience* (QoE) perceived by players and can have a remarkable impact on a game provider's success in the market. We evaluate each approach by running tests that give insight into their performance and enable comparison between them. First, we compare their data stores in terms of the maximum size of game worlds supported. Secondly, we conduct simulations to measure the latency in each service and thus the performance of each approach. Our results show that approaches based on the IaaS layer perform better than those based on the PaaS layer in terms of latency — as we initially expected. However, the PaaS-based approach still performed reasonably well and within the bounds of real-time MMOG latency. Consequently, we believe that the PaaS layer can offer a viable alternative for the development of MMOGs on commodity cloud platforms which pushes the development boundaries of cloud-enabled MMOGs past the IaaS layer. Our results motivate us to explore the possibility of utilizing alternative emerging technologies to enable MMOGs at progressively higher levels of abstraction: PaaS, BaaS and *Function-as-a-Service* (FaaS). The utilization of these higher-level solutions may offer additional advantages to those offered by current cloud solutions based on IaaS. For instance, PaaS, BaaS and FaaS solu-

tions can offer significantly higher levels of abstraction — therefore facilitating faster, more efficient and more sustainable development of MMOG backends.

The rest of this paper is organized as follows: We first discuss related work in section 2. Then we describe our experimental approach and enumerate which platforms we evaluate in section 3. Our pilot implementation of a Minesweeper-themed MOG and its related architecture are discussed in section 4. Then, our evaluation and results are critically presented in in section 5. Finally, we list the conclusions and discuss future work in section 6.

2 Related work

As good performance is one of the most important features of MOGs and MMOGs [16, 38], a large number of related studies have focused on evaluating their platforms using various techniques, such as those we have discussed in a previous work [28]. In their evaluations, various authors also focus on measuring different types of metrics such as:

- Latency – [10, 16, 17, 23, 27, 32, 39, 42].
- Bandwidth – [27].
- Network distance between peers/servers – [16, 43].
- The number of players – [31].
- Messages per second – [31].
- Moves per second – [27].
- The number of connections – [42].

According to [16], "*ensuring an acceptable Quality of Experience (QoE) for all players is a fundamental requirement [for cloud-based games]*". By proposing a mathematical model for measuring the QoE in MMOGs, the authors identify the *global response delay* as the most notable metric. Global response delay — also known as latency — is highly dependent on several other parameters such as the CPU and memory capacity. Furthermore, they argue that other factors, such as network distance between the players and the servers can significantly affect latency. These authors evaluate: (i) the performance of a cloud-based MMOG in terms of latency using simulations, and (ii) the degradation of the QoE as a function of the number of allocated Virtual Machines (VMs) and the number of players, using an empirical approach. Similarly, the authors of [22] state that to maintain the quality of experience, game state updates "*must be delivered within specific time bounds*", depending on the type of MMOG. The study points to the players' *flocking* behavior in certain *hotspots* as a significant challenge because of the high bandwidth load it places on servers. DynFilter [22], a game-oriented message processing middleware based on the publisher-subscriber pattern, filters out state-update messages from entities located far away to reduce bandwidth demand. Experiments on Amazon's EC2 platform have proven that it can maintain bandwidth use within quotas while maintaining the QoE. Another approach is CloudFog [30], a system utilizing fog computing in conjunction with cloud computing to distribute intensive tasks — such as graphics

rendering — to powerful super-nodes which are located closer to the end-user. As a result of its offloading strategy and closer proximity to the players, CloudFog manages to reduce latency, bandwidth consumption and to improve user coverage.

3 Experimental approach

As argued earlier, developers and researchers alike have used a plethora of approaches to implement, deploy and evaluate the performance of backends for MMOGs. To compare various on-premise and native cloud approaches, we have implemented a version of Minesweeper [9], modified to run as a multiplayer game. While Minesweeper is a relatively simple game in terms of complexity and graphics, it still demonstrates the requirement for a backend which can be used to maintain consistence, persistence, and push updates to the clients — all while maintaining an *acceptable* performance. This section describes the general architecture of our implementation and enumerates the approaches we have used. Last, it discusses how we developed the necessary software to evaluate them.

Minesweeper is a game in which the player has to clear a rectangular board that contains hidden mines, without detonating them. It was initially created in the 1960s as a single-player game and gained wide popularity when it was included in Microsoft Windows [14]. As a result, it has seen many offshoots including ones that feature multiplayer competitive gameplay [35].

3.1 Game state

The game state of Minesweeper can be represented as a two-dimensional grid/array — also known as a *tilemap* [15] — which is a common type of game state. Examples of other popular games based on 2D grids are Pac Man, the Civilization series, and the more recent Clash of Clans. Our research focuses on this type of games because their worlds are *persistent* in the long run, meaning that their state is sustained in memory and does not cease to exist after certain conditions are met. In contrast, other types of games such as Call of Duty (first person shooter) are not persistent, which means their state is lost when certain conditions are met and the game is over.

Rather than using match-based gameplay, persistent worlds allow players to control entities that co-exist in a common world which is constantly updated — thus meeting the requirements of our target systems. Theoretically, a Minesweeper game could feature a very large game board with a large number of players accessing it simultaneously and either competing or co-operating with each other to solve the puzzle. Based on previously implemented games, we created several classes which represent the main elements of the game, such as: *BoardState*, *CellState*, *GameState*, etc.

3.2 Actions and rules

We chose to implement Minesweeper because of the relative simplicity of its components, such as its *actions* and *rules*. In terms of actions, the player can either *reveal*, *flag* or *unflag* a selected cell on the board. We have also identified and implemented the following rules: (a) A player can make a single move on a single cell per turn, (b) when an empty cell is revealed, the game displays the number of mines in adjacent cells, (c) if the revealed cell has no adjacent mines, all adjacent cells without a mine are also revealed recursively, (d) when a cell containing a hidden mine is revealed, a penalty is applied, and (e) the game ends when there are no hidden mines left.

3.3 Evaluation strategy

Our evaluation strategy is to develop nearly identical multiplayer Minesweeper games, targeting a set of popular public cloud platforms and use them to compare their performance in terms of latency — the most notable performance metric according to the related work. We specifically target the following platforms which are widely considered to be the most popular/widely used:

- Amazon Web Services: EC2 and DynamoDB (IaaS).
- Microsoft Azure: VM and CosmosDB (IaaS).
- Google Cloud Platform: App Engine and Cloud Datastore (PaaS).

We have chosen these platforms because they allow a meaningful comparison between the services of three major, commodity cloud providers. For this experiment, we kept the same code base for every project but we modified the rules to allow for longer simulations. For instance, we have introduced a score element and award players with points when they reveal an empty cell, and deduct points when they erroneously reveal a cell with a hidden mine. Consequently, instead of ending the game when all mines are flagged (win) or when the player reveals a mine (loss), we consider a game as finished when all cells have been revealed. Our simulations can therefore run for longer periods and allow for a bigger range of tests to take place.

4 Implementation

All projects were developed using Java 8 and are based on the client-server architecture, which is the preferred choice for most MMOG systems [23, 27, 29]. We identify several architectural components necessary to build an MMOG system: a client application, a server/backend application, a data store, and a state update mechanism.

Our client applications have been kept completely identical throughout the three approaches. The objectives of the client are: (a) to allow visualization of the game state, (b) initiate simulations with multiple players, and (c) gather data regarding performance and save it in a local files.

The purpose of the backend is to provide access to services of the game so that the clients can perform in-game actions. The functionality of the backend is exposed through a set of commands that can be accessed through an *Application Programming Interface* (API). When a client issues a request, the backend resolves it, executes the logic that enforces the game's rules, and performs the necessary actions by updating the state stored in the data store.

The data store component is used to persistently store the game worlds/states as well as information about the players and their game sessions. Lastly, the state-update mechanism component is responsible for updating the client's view of the game state, once an update to the game state has occurred, or periodically when latency allows for it. Figure 1 summarizes the general architecture used in our implementations.

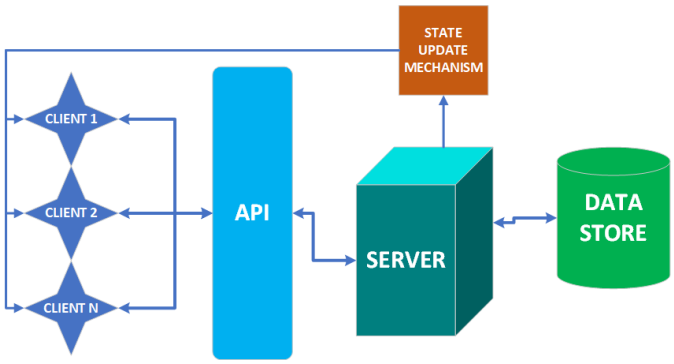

Fig. 1. General architecture for all approaches.

To allow communication between the clients and the servers, we provide a command interface that allows clients to issue commands to servers as requests, and servers to respond with the corresponding data after retrieving the state from their data store. We use the following minimal interface which contains five core functions. `/createGame` creates a new Minesweeper game, `/join` allows a player to join a game, `/list` lists all the available games, `/getState` allows players who have joined a game to get its state, and `/play` allows players in a game to perform an action on a selected cell.

All of our implementations utilize the *Area of Interest* (AoI) [7, 20, 23, 40] concept to reduce the bandwidth required to communicate the game's state. We use a class called *BoardState* to store the state of each game's board, which is an abstract class that can store game cells. We use *FullBoardState* and *PartialBoardState*, which are both derived from BoardState, to distinguish states which contain the *full* game state from those containing a specific, *partial* part of the game state. While the server utilizes the former for storage and computation, each client can only see a part of that full state — they receive a partial state

based on their location in the game. The location of each client's partial state can be moved by issuing a move command to the `/play` service and specifying the new desired location within the game board. To support game state updates for the clients we use Ably, a real-time WebSocket infrastructure [1] that enables a publish-subscribe mechanism supporting the concept of AoI.

Client-side Our client application uses Java Swing forms to visualize the Minesweeper board for testing purposes. To carry out simulations, we have created a simple Minesweeper solver — with no GUI — that tries to solve the game by opening cells sequentially – i.e. moving rightwards and then downwards as the game progresses. Naturally this is a sub-optimal approach to play Minesweeper, but our study focuses on the performance of the backend in terms of serving player requests rather than the efficiency of solving the game.

We run our simulations by initializing the client programs with information such as the width and height of the game board, the number of players in the game, the size of their partial states and the delay between the execution of moves by each player. Our simulations instantiate a new thread for each player in the game, with each player going through a series of steps that simulate real player actions in a multiplayer game. Firstly, our bot players request a list of all available games from the server by calling the `/list` service. Upon acquiring this list, they always select the first available game and try to join it by calling `/join` and specifying their name. The names of players are automatically set when created (e.g. $Player_1$, $Player_2$, ... $Player_N$). After successfully joining the game, a player requests the initial partial game state using `/getState`. Upon receiving the initial game state, the players run the solver and submit their moves — *reveal* or *flag/unflag* — using the `/play` call. When all cells in their visible area have been revealed, the players try to shift their position rightwards and then downwards by calling `/play` and issuing a *move* command.

To allow multiplayer gameplay, we define an additional entity called *Session*, which couples a certain player to a specific game that the player has joined. Using sessions, we track the locations and actions of players, award or deduct points for their actions, and choose which players to send game state updates to, based on their proximity to those updates.

Our client records the time the request is sent and the response is received for each of these calls, using timestamps. The time taken for the round trip of the request-response (latency) is found in terms of milliseconds, by subtracting the two timestamps. The recorded values are stored in memory and saved in a comma-separated value (CSV) file, as shown in table 1.

Amazon Web Services backend Our Amazon Web Services (AWS) project is hosted by an Amazon EC2 t2.micro instance, running on Linux Ubuntu 18 and features 1 vCPU, 1GB of RAM and "*low to moderate*" network performance [3]. Our server does not store any game data but instead utilizes an instance of DynamoDB with a provisioned capacity within the free tier [4]. DynamoDB is

players	endpoint	latency(ms)
2	GAME_LIST	1414
2	JOIN	310
2	STATE_GET	141
2	PLAY	335
...	...	...

Table 1. The format used for the simulation results file.

Amazon's NoSQL data store which stores data in tables that contain items consisting of key-value pairs. Amazon claims that DynamoDB has "*single-digit millisecond performance at any scale*" [4]. We have implemented our project using Java Servlets running on Apache Tomcat 9 [21], with each Servlet implementing an endpoint of the API. Client-server communication occurs through the HTTP protocol, with each client issuing requests to the server and the server carrying out the request and responding with the necessary information. To retrieve data from DynamoDB we use DynamoDBMapper [5], a library which maps client-side classes to DynamoDB tables using code annotations. As with all platform setups, we use Ably [1] for state updates. Ably allows state update messages to be sent from the server through a channel in real time. Clients subscribe to a channel once they have an active game session and listen for state updates from the server. For these experiments, we used the free package of Ably [2].

Microsoft Azure backend Our Microsoft Azure backend project is powered by a B1S-type virtual machine running on Linux Ubuntu 14 and featuring 1 vCPU and 1GB of RAM[33]. As in the AWS project, we achieve client-server communication using HTTP and Java Servlets powered by Apache Tomcat 9. The two projects are almost identical (i.e. we use the same endpoints, algorithms, etc.). The only difference is the code which utilizes the data store since we opt to use an Azure product in this approach. To store data, we use Azure's CosmosDB [34], a NoSQL data store that saves data in documents as key-value pairs. In terms of performance, Microsoft also claims that projects utilizing CosmosDB can "*take advantage of fast, single-digit-millisecond data access*" [34]. Just as in our AWS project, we realize server-to-client state updates using Ably's free package, and identical code in our state update function.

Google App Engine backend Our third and last approach is based on a server-less PaaS infrastructure. We use *Google App Engine* (GAE) [25], a fully-managed platform that allows application development without the need to deal with server configuration — i.e. realizing *serverless* computing [11]. App Engine allows applications to scale *seamlessly* and without developer supervision, which is a major advantage over the other approaches. The serverless architecture of this approach lets App Engine manage the server resources — we only had to create an App Engine instance and select our environment (Java 8). App Engine

Java projects utilize Jetty 9 [19], an HTTP server that is similar to Apache Tomcat, which also enables clients to communicate with the server using Java Servlets. Our web-based API is kept identical to the other two native cloud approaches, with the only difference being the code utilizing the data store. In this approach, we use Google's NoSQL solution, the Cloud Datastore [26]. Google claims that its Cloud Datastore "*scales seamlessly [...] with your data allowing applications to maintain high performance as they receive more traffic*" [26]. The Cloud Datastore saves data in documents called *Entities* that contain key-value associations. To easily interact with the Cloud Datastore we utilize a Java library called Objectify [41], which allows us to annotate classes as entities and easily perform CRUD operations. Just like with the other approaches, we have implemented game state updates using Ably. Figure 2 shows the selection of architectural components for all three approaches.

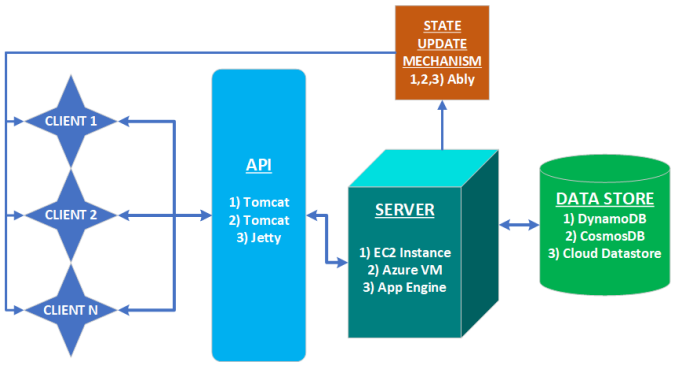

Fig. 2. Architectures used for all three platforms: (1) Amazon Web Services, (2) Microsoft Azure, (3) Google Cloud Platform.

5 Evaluation

Our evaluation is driven by the performance aspect of MMOGs. We evaluate each approach independently, while maintaining identical secondary components such as game-solving algorithms and game logic. We focus on performance because we believe it is the most important performance metric, as also indicated in several related works [10, 16, 17, 23, 27, 32, 39, 42].

In our data collection experiments we aimed to keep secondary factors in control as follows:

– We aimed to keep the network conditions as similar as possible by running the experiments within the same wired network. We also (a) monitored the network, verifying that it was not being utilized by other programs at the

time and (b) ran the experiments at similar times and days of the week to avoid different network conditions.
- We kept the client device conditions as similar as possible by running all simulations on the same computer while it was initially idle.
- We used comparable data center locations (Eastern United States) for all experiments.
- We used NoSQL data stores for all cloud approaches to allow a comparison between them. We use this type of persistence because it can be easily scaled and appears to better match the needs of MMOG backends [8, 12, 44].
- We created virtual machines with similar specifications to keep the backend processing power as comparable as possible.
- We conducted our experiments based on a set of identical commands and made sure that the parameters and logic of those calls stayed the same throughout all simulations.

To establish a base latency for each approach, we created a Servlet that performs no operations and returns an empty result. The purpose of the *BaseServlet* is to allow us to establish a minimum latency for each approach. Given that our code is kept the same for game logic, this helps compare the latency between calls that utilize the backend extensively and those that do not — thus determining the latency caused by our backend implementations. Secondly, we measured the latency of each endpoint in our API by running our simulations and obtaining the data from a local file, as shown in table 1. In each case, we ran simulations 10 times for each endpoint, including the base latency endpoint and took averages from these results. We ran the base latency test first, which yielded an average of 97.2ms for Google's App Engine, 144.2ms for Microsoft Azure and 167.3ms for the Amazon Web Services approach.

Before testing each of the actual backend services, we performed several tests to establish the maximum size of the game board state possible in each approach. In our approach, the size of game state is limited by the size of the *unit* element used in the corresponding data store. While there exist ways to circumvent these limitations to create bigger game sizes, we kept our implementations free of these modifications for three reasons: First, state modeling is beyond the scope of this paper. Second, a workaround implemented on a specific platform may not necessarily work on all platforms – thus making it harder to compare our results. Third, we aimed to keep our implementations as simple and consistent as possible.

We conducted these tests by creating square-sized boards where the width is the same as the height. Initially, we attempted to create games of size 100x100 – if that game size could be created we incremented the size by 50% and tried again. When the game could not be created anymore because of platform-enforced limitations, we reduced the size by 25% and tried again until we found the exact size of game boards supported by each approach. Our experiments showed that Microsoft's CosmosDB supports a game state up to 229x229 cells, Google's Cloud Datastore takes the middle ground, with up to 158x158 game boards and Amazon's DynamoDB supports a maximum size of up to 98x98 board states for

Minesweeper. These hard limitations are subject to change from game to game and are dependent on each platform.

Upon establishing the maximum state size for each approach, we used the minimum of those values as input to evaluate the performance of the `/createGame` service. We performed HTTP GET requests by specifying the administrator password, the maximum number of players allowed, the game size and the difficulty of the game. We kept the parameters of all these calls constant throughout all experiments by using a game size of 98 — the minimum out of the three approaches — setting the difficulty to *Easy* and the maximum number of players to 10. Our results from ten calls indicate an average latency of 332.7ms for AWS, 346.3ms for Azure and 496.6ms for App Engine.

Our next test focused on the `/list` service, which returns a list of all available games. It is important to indicate that the list service only retrieves information about a game (such as its ID, width and height) but not its actual state. To conduct this test, we created two games in each of the three data stores and called the list service. Our results from ten rounds of simulations show that Azure took 555ms to respond, AWS took 568.5ms, while App Engine took 1153.2ms.

To test the `/join`, `/getState` and `/play` services, we ran ten simulations with the following configuration: a game size of 10x10 with two players, difficulty set to easy and a partial state of 5x5 for each player. AWS performed best in joining the game, with a latency of 201.8ms, compared to Azure's 234.8ms and App Engine's 554.6ms. AWS also performed marginally better when retrieving the initial state of the game at 176.3ms, while App Engine took 176.7ms and Azure 245.4ms. When calling the play service, Azure performed marginally better with 175.8ms while AWS took just a bit longer at 176.9ms. App Engine took an average of 201.2ms to respond to play requests. Table 2 summarizes our latency test results.

Approach	Base latency	Create game	List	Join	Get State	Play
Amazon EC2	167.3	332.7	568.5	201.8	176.3	176.9
Microsoft Azure	144.2	346.3	555	234.8	245.4	175.8
Google App Engine	97.2	496.6	1153.2	554.6	176.8	201.2

Table 2. A summary of average latencies for all approaches for various API calls. All time measurements are in milliseconds.

Data gathered during the evaluation shows that IaaS-based approaches generally performed better than their PaaS counterpart, which was expected because of the larger overhead of computational layers being present in the PaaS approach. From the latency results of our game service calls, we observe that AWS IaaS approach performed better in three of those services (create game, join and get state), while Azure performed better in two (list and play).

In contrast, App Engine performed significantly better — about 33% faster — compared to the other two approaches in the base latency test, something

that may reveal a higher latency caused by Google's Cloud Datastore which was utilized in the Minesweeper services but not in the base latency test.

In the create game service, the AWS approach performed slightly better (4%) compared to the Azure approach. The opposite applies for the list service, where Azure performs marginally better (3%) compared to AWS. The difference between the two is more significant when joining the game, with AWS scoring a 15% improvement compared to Azure. In these services, App Engine scores relatively poorly compared to the two IaaS approaches.

In the get state service, App Engine performs better and almost ties AWS's better-performing service – the two have a negligible difference (1%) with Azure falling behind by a relatively large margin (-28%).

The most significant test is conducted on the play service which we regard as the most important of all services because it is the one which is most frequently called by the players during a game. This means it can impact the performance of the MMOG most significantly. In this test, AWS and Azure performed within 1% difference of each other, with Azure performing better by about 1ms. App Engine also scores a relatively low latency (201.2ms) but ends up performing about 13% worse than Azure.

By combining this relatively good performance with (1) inherent elasticity, (2) code abstraction and (3) the elimination of infrastructural management from developers, the PaaS layer appears to offer a viable alternative development approach for cloud-enabled games — one that many game developers could benefit from in the future.

6 Conclusions and future work

In this paper, we selected a set of public cloud platforms to assess the suitability of public clouds for developing MMOG backends. To do this, we implemented a modified version of Minesweeper in each of three selected approaches. Our implementation extended a traditionally single-player game to run as an MMOG on the infrastructure of three major cloud providers: Amazon Web Services, Microsoft Azure and Google App Engine. Our findings suggest that MMOGs can be engineered to run on high-level commodity cloud platforms, something that game providers have generally avoided so far. We compare the performance of our game's services in terms of latency, using simulations. Our results show that the two IaaS approaches (AWS and Azure) have performed better than the PaaS approach (App Engine), which is what we initially expected. Based on related work, the expected latency of real-time MMOGs is near or below 250ms [23, 36, 46]. From our results, we conclude that the PaaS layer offers acceptable performance which indicates that it could provide a suitable environment for realizing MMOG backends — at least for game types that do not require very low latency. While not as good as in IaaS, the performance of our PaaS approach puts it within the limits even of real-time MMOGs. With its extra benefits — easier, faster and more economical development — we argue that PaaS is becoming a competitive option for realizing MMOG backends.

For the future, we aim to improve our understanding of developing and deploying MMOGs on public clouds by studying related models, methods and tools. Our priority is to complement our work by exploring another important aspect of MMOGs — scalability — through the evaluation of models that allow varying sizes of game worlds. Due to technical limitations, our approach was limited to ten players, which is not representative of MMOGs but rather of MOGs. Furthermore, Minesweeper is a turn-based game and is therefore different from more popular types of online games. It does not fall into popular categories such as First Person Shooter (FPS), Real-Time Strategy (RTS) and so on, which is not the focus of this research. Lastly, our approach utilizes various data stores. Some of these may be optimized towards read operations, while others may be optimized towards write operations — perhaps skewing the latencies scored by some approaches. Despite that, we argue that our work is a good starting point and showcases the possibilities that lie ahead, especially when more work is done with respect to scalability. Such an advancement will allow MMOGs running on public clouds to handle workloads with far larger numbers of players and game world sizes than what we have used in the present study.

References

1. Ably: Ably realtime. https://www.ably.io/ (2019), last accessed: 2019-12-10
2. Ably: Pricing — ably realtime. https://www.ably.io/pricing (2019), last accessed: 2019-12-10
3. Amazon Web Services: Amazon ec2 instance types. https://aws.amazon.com/ec2/instance-types/ (2019), last accessed: 2019-12-10
4. Amazon Web Services: Dynamodb - overview. https://aws.amazon.com/dynamodb/pricing/provisioned (2019), last accessed: 2019-12-10
5. Amazon Web Services: Dynamodbmapper - amazon dynamodb. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DynamoDBMapper.html (2019), last accessed: 2019-12-10
6. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al.: A view of cloud computing. Communications of the ACM **53**(4), 50–58 (2010)
7. Assiotis, M., Tzanov, V.: A distributed architecture for massive multiplayer online role-playing games. In: NetGames' 06 Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games, Article. No. 4 in 2005 (2005)
8. Baker, J., Bond, C., Corbett, J.C., Furman, J., Khorlin, A., Larson, J., Leon, J.M., Li, Y., Lloyd, A., Yushprakh, V.: Megastore: Providing scalable, highly available storage for interactive services. In: Proceedings of the Conference on Innovative Data system Research (CIDR). pp. 223–234 (2011)
9. Becker, K.: Teaching with games: the minesweeper and asteroids experience. Journal of Computing Sciences in Colleges **17**(2), 23–33 (2001)
10. 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. p. 17. ACM (2016)
11. Castro, P., Ishakian, V., Muthusamy, V., Slominski, A.: The rise of serverless computing. Communications of the ACM **62**(12), 44–54 (Nov 2019). https://doi.org/10.1145/3368454

12. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS) **26**(2), 4 (2008)
13. Chu, H.S.: Building a simple yet powerful mmo game architecture. Verkkoarkkitehtuuri. Part (2008)
14. Cobbett, R.: The most successful game ever: a history of minesweeper (May 2009), https://www.techradar.com/news/gaming/the-most-successful-game-ever-a-history-of-minesweeper-596504, last accessed: 2019-12-12
15. Coleman, R., Roebke, S., Grayson, L.: Gedi: a game engine for teaching videogame design and programming. Journal of Computing Sciences in Colleges **21**(2), 72–82 (2005)
16. Dhib, E., Boussetta, K., Zangar, N., Tabbane, N.: Modeling cloud gaming experience for massively multiplayer online games. In: Consumer Communications & Networking Conference (CCNC), 2016 13th IEEE Annual. pp. 381–386. IEEE (2016)
17. Dhib, E., Zangar, N., Tabbane, N., Boussetta, K.: Resources allocation trade-off between cost and delay over a distributed cloud infrastructure. In: 2016 7th International Conference on Sciences of Electronics, Technologies of Information and Telecommunications (SETIT). pp. 486–490. IEEE (2016)
18. Ducheneaut, N., Yee, N., Nickell, E., Moore, R.J.: Building an mmo with mass appeal: A look at gameplay in world of warcraft. Games and Culture **1**(4), 281–317 (2006). https://doi.org/10.1177/1555412006292613
19. Eclipse: Jetty - servlet engine and http server. https://www.eclipse.org/jetty/ (2019), last accessed: 2019-12-10
20. El Rhalibi, A., Al-Jumeily, D.: Dynamic area of interest management for massively multiplayer online games using opnet. In: 2017 10th International Conference on Developments in eSystems Engineering (DeSE). pp. 50–55. IEEE (2017)
21. Foundation, A.: Apache tomcat. http://tomcat.apache.org/ (2019), last accessed: 2019-12-10
22. Gascon-Samson, J., Kienzle, J., Kemme, B.: Dynfilter: Limiting bandwidth of online games using adaptive pub/sub message filtering. In: Proceedings of the 2015 International Workshop on Network and Systems Support for Games. p. 2. IEEE Press (2015)
23. GauthierDickey, C., Zappala, D., Lo, V.: Distributed architectures for massively multiplayer online games. In: ACM NetGames Workshop. Citeseer (2004)
24. Ghobaei-Arani, M., Khorsand, R., Ramezanpour, M.: An autonomous resource provisioning framework for massively multiplayer online games in cloud environment. Journal of Network and Computer Applications (06 2019). https://doi.org/10.1016/j.jnca.2019.06.002
25. Google Cloud: App engine - google cloud. https://cloud.google.com/appengine/ (2019), last accessed: 2019-12-10
26. Google Cloud: Datastore - nosql schemaless database. https://cloud.google.com/datastore/ (2019), last accessed: 2019-12-10
27. 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. pp. 60–65. ACM (2008)
28. Kasenides, N., Paspallis, N.: A systematic mapping study of mmog backend architectures. Information (Switzerland) **10**, 264 (08 2019). https://doi.org/10.3390/info10090264

29. Kavalionak, H., Carlini, E., Ricci, L., Montresor, A., Coppola, M.: Integrating peer-to-peer and cloud computing for massively multiuser online games. Peer-to-Peer Networking and Applications **8**(2), 301–319 (2015)
30. Lin, Y., Shen, H.: Cloud fog: Towards high quality of experience in cloud gaming. In: 2015 44th International Conference on Parallel Processing. pp. 500–509. IEEE (2015)
31. Lu, F., Parkin, S., Morgan, G.: Load balancing for massively multiplayer online games. In: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games. p. 1. ACM (2006)
32. Meiländer, D., Gorlatch, S.: Modeling the scalability of real-time online interactive applications on clouds. Future Generation Computer Systems **86**, 1019–1031 (2018)
33. Microsoft Azure: Introducing b-series, our burstable vm size. https://azure.microsoft.com/en-au/blog/introducing-b-series-our-new-burstable-vm-size/ (2019), last accessed: 2019-12-10
34. Microsoft Azure: Introduction to azure cosmosdb. https://docs.microsoft.com/en-us/azure/cosmos-db/introduction (2019), last accessed: 2019-12-10
35. minesweeper.io: Minesweeper.io. https://minesweeper.io/ (2019), last accessed: 2019-12-10
36. Nae, V., Iosup, A., Prodan, R.: Dynamic resource provisioning in massively multiplayer online games. IEEE Transactions on Parallel and Distributed Systems **22**(3), 380–395 (2011)
37. 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. p. 9. IEEE Press (2009)
38. Nae, V., Prodan, R., Iosup, A.: Massively multiplayer online game hosting on cloud resources. Cloud Computing: Principles and Paradigms pp. 491–509 (2011)
39. Najaran, M.T., Krasic, C.: Scaling online games with adaptive interest management in the cloud. In: Network and Systems Support for Games (NetGames), 2010 9th Annual Workshop on. pp. 1–6. IEEE (2010)
40. Negrão, A.P., Veiga, L., Ferreira, P.: Task based load balancing for cloud aware massively multiplayer online games. In: Network Computing and Applications (NCA), 2016 IEEE 15th International Symposium on. pp. 48–51. IEEE (2016)
41. Objectify: Objectify. https://github.com/objectify/objectify (2019), last accessed: 2019-12-10
42. Plumb, J., Kasera, S., Stutsman, R.: Hybrid network clusters using common gameplay for massively multiplayer online games. pp. 1–10 (08 2018). https://doi.org/10.1145/3235765.3235785
43. Plumb, J.N., Stutsman, R.: Exploiting google's edge network for massively multiplayer online games. In: 2018 IEEE 2nd International Conference on Fog and Edge Computing (ICFEC). pp. 1–8. IEEE (2018)
44. Shabani, I., Kovaçi, A., Dika, A.: Possibilities offered by google app engine for developing distributed applications using datastore. In: Computational Intelligence, Communication Systems and Networks (CICSyN), 2014 Sixth International Conference on. pp. 113–118. IEEE (2014)
45. Shaikh, A., Sahu, S., Rosu, M.C., Shea, M., Saha, D.: On demand platform for online games. IBM Systems Journal **45**(1), 7–19 (2006)
46. Shea, R., Liu, J., Ngai, E.C.H., Cui, Y.: Cloud gaming: architecture and performance. IEEE network **27**(4), 16–21 (2013)