University of Central Lancashire
School of Psychology and Computer Science
Department of Computing

Models, methods, and tools for developing MMOG backends on commodity clouds

Nicos Kasenides

A thesis submitted for the degree of
Doctor of Philosophy
September 2022

RESEARCH STUDENT DECLARATION FORM

Type of Award **PhD**

School **School of Psychology and Computer Science**

1. Concurrent registration for two or more academic awards

I declare that while registered as a candidate for the research degree, I have not been a registered candidate or enrolled student for another award of the University or other academic or professional institution.

2. Material submitted for another award

I declare that no material contained in the thesis has been used in any other submission for an academic award and is solely my own work.

3. Collaboration

Where a candidate's research programme is part of a collaborative project, the thesis must indicate in addition clearly the candidate's individual contribution and the extent of the collaboration. Please state below:

N/A (No collaboration)

4. Use of a Proof-reader

No proof-reading service was used in the compilation of this thesis.

Signature of Candidate

N Kasenides

Print name: Nicos Kasenides

Abstract

Online multiplayer games have grown to unprecedented scales, attracting millions of players worldwide. The revenue from this industry has already eclipsed well-established entertainment industries like music and films and is expected to continue its rapid growth in the future. Massively Multiplayer Online Games (MMOGs) have also been extensively used in research studies and education, further motivating the need to improve their development process.

The development of resource-intensive, distributed, real-time applications like MMOG backends involves a variety of challenges. Past research has primarily focused on the development and deployment of MMOG backends on dedicated infrastructures such as on-premise data centers and private clouds, which provide more flexibility but are expensive and hard to set up and maintain. A limited set of works has also focused on utilizing the Infrastructure-as-a-Service (IaaS) layer of public clouds to deploy MMOG backends. These clouds can offer various advantages like a lower barrier to entry, a larger set of resources, etc. but lack resource elasticity, standardization, and focus on development effort, from which MMOG backends can greatly benefit.

Meanwhile, other research has also focused on solving various problems related to consistency, performance, and scalability. Despite major advancements in these areas, there is no standardized development methodology to facilitate these features and assimilate the development of MMOG backends on commodity clouds. This thesis is motivated by the results of a systematic mapping study that identifies a gap in research, evident from the fact that only a handful of studies have explored the possibility of utilizing serverless environments within commodity clouds to host these types of backends. These studies are mostly vision papers and do not provide any novel contributions in terms of methods of development or detailed analyses of how such systems could be developed. Using the knowledge gathered from this mapping study, several hypotheses are proposed and a set of technical challenges is identified, guiding the development of a new methodology.

The peculiarities of MMOG backends have so far constrained their development and deployment on commodity clouds despite rapid advancements in technology. To explore whether such environments are viable options, a feasibility study is conducted with a minimalistic MMOG prototype to evaluate a limited set of public clouds in terms of hosting MMOG backends. Fol-

lowing encouraging results from this study, this thesis first motivates toward and then presents a set of models, methods, and tools with which scalable MMOG backends can be developed for and deployed on commodity clouds. These are encapsulated into a software development framework called Athlos which allows software engineers to leverage the proposed development methodology to rapidly create MMOG backend prototypes that utilize the resources of these clouds to attain scalable states and runtimes. The proposed approach is based on a dynamic model which aims to abstract the data requirements and relationships of many types of MMOGs. Based on this model, several methods are outlined that aim to solve various problems and challenges related to the development of MMOG backends, mainly in terms of performance and scalability. Using a modular software architecture, and standardization in common development areas, the proposed framework aims to improve and expedite the development process leading to higher-quality MMOG backends and a lower time to market. The models and methods proposed in this approach can be utilized through various tools during the development lifecycle.

The proposed development framework is evaluated qualitatively and quantitatively. The thesis presents three case study MMOG backend prototypes that validate the suitability of the proposed approach. These case studies also provide a proof of concept and are subsequently used to further evaluate the framework. The propositions in this thesis are assessed with respect to the performance, scalability, development effort, and code maintainability of MMOG backends developed using the Athlos framework, using a variety of methods such as small and large-scale simulations and more targeted experimental setups. The results of these experiments uncover useful information about the behavior of MMOG backends. In addition, they provide evidence that MMOG backends developed using the proposed methodology and hosted on serverless environments can: (a) support a very high number of simultaneous players under a given latency threshold, (b) elastically scale both in terms of processing power and memory capacity and (c) significantly reduce the amount of development effort. The results also show that this methodology can accelerate the development of high-performance, distributed, real-time applications like MMOG backends, while also exposing the limitations of Athlos in terms of code maintainability.

Finally, the thesis provides a reflection on the research objectives, considerations on the hypotheses and technical challenges, and outlines plans for future work in this domain.

Acknowledgements

First and foremost, I would like to express my gratitude to my supervisor, Dr. Nearchos Paspallis, who has encouraged me to pursue this Ph.D. and provided an extraordinary amount of support through the years. It has been a privilege to be under his supervision both as a B.Sc. and Ph.D. student and to have had the opportunity to hold many of our productive meetings and discussions. Dr. Paspallis has given me invaluable advice in so many aspects that I cannot even begin to enumerate and provided insights without which I would never have finished this thesis. He trusted me with the delivery of several Computing modules despite the fact that I had no teaching experience, and his encouragement to participate in many projects had a profound impact on my skills as a researcher and software engineer. Such endeavors challenged me to grow as an academic, learner, and person, but also provided much-needed economic support through the years, alleviating the financial burden through difficult times. He has set the highest example, not only as an academic and researcher but also as a person.

I am also grateful to my second supervisor, Professor Irene Polycarpou, who has supported me many times during this course. She has provided much-needed advice during my studies, helping to formulate my Ph.D. proposal and reviewing important documents. Professor Irene has also provided various employment opportunities which were crucial in sustaining my efforts through the years, for which I am very thankful.

My colleague and friend, Dr. Andrie Piki has also had a profound impact on my progress. Dr. Piki was one of the very few people outside my supervisory team to take a keen interest in my research, supporting me and providing valuable advice and emotional support. It is a privilege and an opportunity to work with such a talented educator, and I hope that her energetic spirit, positivity, and tremendous skills continue to influence my development as an academic.

I would like to attribute many additions, re-considerations, and improvements made to this thesis to my internal and external examiners, Dr Josephina Antoniou, Prof. Andreas Andreou, and Dr George Pallis. Through their positive attitude, detailed feedback, and constructive comments, they have made my Ph.D. viva an enjoyable and memorable experience.

This thesis is dedicated to my fiancée, Panayiota, who has been more patient and caring than I could ever ask for. Meeting her has been an incredible stroke of luck that unlocked my better self, and I am incredibly thankful for her encouragement, support, and love. I would have never pursued this degree with such high determination and consistency without her help, as she has provided clarity and emotional support through the most difficult of times. I cannot describe in words how much she means to me and how lucky I am to have met her.

I am thankful for the remarkable support and unconditional love of my parents, Andreas and Vicky, without which I would have never reached this far. This thesis is partly dedicated to them. My mother's persistence to make me study has once again paid off, after many arguments and heated discussions during my junior years, but also her playful reminders to study, even as a Ph.D. student. My father's passion for computers greatly influenced me in my junior years and his guidance steered me toward a career in computer science and academia. By visiting his workspace, I had the opportunity to experience what a university is long before I attended or worked for one, and without his financial support, I would never have been able to undertake any of my studies.

I would also like to thank my grandparents Nicos and Despina. They practically raised me in my junior years, nurtured my love for nature, and taught me simplicity, selflessness, modesty, and curiosity. Along with my other grandparents, Fanis and Eleni, they endured the hardships of war and poverty, but still managed to provide for their children and grandchildren. This thesis is partly dedicated to my late grandparents, for their never-ending care, diligence, and incredible resilience, in front of which my own efforts fade into insignificance.

My godmother Yiota, and my friend Marios have also supported me over the years. I would like to thank them for always being there, and for being among the very few I could count on. I am also grateful to Sebastian for his lasting friendship and support. We have spent many hours discussing various matters and playing multiplayer online games, which has helped to take my mind away from work at the end of some long days.

This project has received financial support from Google and the Cyprus Youth Organization, without which it would be impossible to develop, test, and evaluate this thesis. I would like to applaud their support for education through their programs and express my appreciation for the economic lifeline they provided to this project.

To the love of my life, Panayiota.

Table of contents

Abstract **i**

Acknowledgements **iii**

Table of contents **vii**

List of tables **xiv**

List of figures **xvi**

List of listings **xxi**

Acronyms **xxii**

1 Introduction **1**

1.1 The characteristics of MMOG backends 2

1.2 Motivation 4

1.3 Scope and Objectives 6

1.4 Contributions 11

1.5 Publications 14

1.6 Statement of Originality 15

1.7 Thesis structure 15

2 Related work **18**

2.1 Introduction . . . 18

2.2 Research questions . . . 19

2.3 Search strategy . . . 20

2.4 Criteria . . . 20

2.4.1 Inclusion criteria . . . 20

2.4.2 Exclusion criteria . . . 21

2.5 Review process and data collection . . . 21

2.6 Aspect selection . . . 22

2.7 Approach categorization . . . 23

2.7.1 Infrastructure . . . 23

2.7.2 Architecture . . . 24

2.7.3 Performance . . . 24

2.7.4 Scalability . . . 25

2.7.5 Persistence . . . 26

2.7.6 Security . . . 26

2.8 Literature review . . . 28

2.8.1 Infrastructure . . . 28

2.8.2 Architecture . . . 36

2.8.3 Performance . . . 42

2.8.4 Scalability . . . 47

2.8.5 Persistence . . . 52

2.8.6 Security . . . 55

2.8.7 Other approaches . . . 57

2.9 Analysis of the related works . . . 60

2.9.1 Infrastructure 60

2.9.2 Architecture 64

2.9.3 Performance 65

2.9.4 Scalability 66

2.9.5 Persistence 68

2.9.6 Security 69

2.10 Insights and future research directions 69

3 Feasibility study **71**

3.1 Introduction 71

3.2 Objectives 72

3.3 Experiment overview 72

3.4 Implementation 73

3.5 Approaches 76

3.6 Evaluation 77

3.7 Commodity cloud support for MMOG backends 81

3.8 Conclusions 84

4 The Athlos framework **86**

4.1 Introduction 86

4.2 Motivation 88

4.2.1 Other frameworks 88

4.2.2 Case study: Mars Pioneer 89

4.3 Model 90

4.3.1 Data types 91

4.3.2 Type extensibility 91

4.3.3 Static and dynamic models 92
4.3.4 Worlds (NX) 95
4.3.5 Terrain 100
4.3.6 Terrain identifiers (NX) 102
4.3.7 Entities (X) 102
4.3.8 Partial states (NX) 105
4.3.9 State updates (NX) 106
4.3.10 Other types 106
4.3.11 Games and rules 107
4.4 Methods 108
4.4.1 Game definitions 110
4.4.2 Infrastructure 112
4.4.3 Architecture 115
4.4.4 Persistence 124
4.4.5 Data serialization 125
4.4.6 Networking 130
4.4.7 Performance and scalability 137
4.5 Tools 160
4.5.1 The Athlos API 160
4.5.2 Project editor 161
4.5.3 Code generator 164
4.5.4 Guide 168
4.5.5 Libraries 168
4.6 Conclusions 169

5 Case studies 170

5.1 Introduction . . . 170

5.2 Case study 1: Mars Pioneer . . . 171

5.2.1 Development . . . 171

5.2.2 Impact on framework . . . 174

5.3 Case study 2: aMazeChallenge . . . 177

5.3.1 First version . . . 177

5.3.2 Development . . . 179

5.3.3 Impact on framework . . . 181

5.4 Case study 3: Minesweeper . . . 182

5.4.1 Development . . . 183

5.4.2 Impact on framework . . . 185

5.5 Conclusions . . . 185

6 Evaluation 186

6.1 Introduction . . . 186

6.2 Evaluation strategy . . . 187

6.3 Performance and runtime scalability . . . 188

6.4 State scalability . . . 203

6.4.1 Absolute state size . . . 204

6.4.2 Sub-state loading time . . . 206

6.4.3 Queries vs loading time . . . 207

6.4.4 Serialization time . . . 210

6.5 Development effort . . . 216

6.6 Code maintainability . . . 219

6.7 Tools . . . 221

6.8 Conclusions . . . 222

7 Analysis 224

7.1 Introduction . . . 224

7.2 Addressing the hypotheses . . . 225

7.2.1 Hypothesis 1 . . . 225

7.2.2 Hypothesis 2 . . . 226

7.2.3 Hypothesis 3 . . . 227

7.2.4 Hypothesis 4 . . . 228

7.2.5 Hypothesis 5 . . . 228

7.2.6 Hypothesis 6 . . . 229

7.3 Addressing the technical challenges . . . 230

7.3.1 Challenge 1 . . . 230

7.3.2 Challenge 2 . . . 231

7.3.3 Challenge 3 . . . 233

7.3.4 Challenge 4 . . . 234

7.3.5 Challenge 5 . . . 235

7.3.6 Challenge 6 . . . 235

7.3.7 Challenge 7 . . . 236

7.4 Limitations . . . 238

7.4.1 Development methodology . . . 238

7.4.2 Research methodology . . . 239

8 Conclusion 242

8.1 Contributions and content . . . 242

8.2 Impact . . . 244

8.3 Future work . . . 246

9 Appendices **249**

9.A Feasibility study data 249
9.B Model 252
9.B.1 Players (NX) 252
9.B.2 Teams (NX) 252
9.B.3 Positioning and direction (NX) 252
9.B.4 Events (X) 254
9.B.5 Actions (X) 254
9.B.6 Game sessions (NX) 255
9.B.7 World sessions (NX) 255
9.B.8 Services (X) 255
9.B.9 Requests and Responses (X) 256
9.C State API diagram 258
9.D Mars Pioneer case study code 260
9.E aMazeChallenge case study code 266
9.F Libraries 269
9.F.1 Firestorm 269
9.F.2 Objectis 271
9.F.3 World generation 273
9.G Tool evaluation 276
9.G.1 Firestorm 276
9.G.2 Objectis 278
9.G.3 ByteSurge 281

Bibliography **287**

List of tables

2.1 Different aspects identified during the review process, sorted in descending order of importance based on the number of papers mentioning them. 23

2.2 Aspects and categories used to classify approaches for developing Massively Multiplayer Online Game (MMOG) backends. 61

2.3 Comparing the studied approaches using the identified criteria (Infrastructure, Architecture, Scalability, Persistence, Performance and Security). 62

6.1 Results showing the time taken to run through each of the identified stages in the backend's request-response cycle. 194

6.2 The processing latency in terms of milliseconds, recorded for various sub-processes of a play service in a locally-hosted version of Mars Pioneer. 197

6.3 The processing latency in terms of milliseconds, recorded for various sub-processes of a play service in a locally-hosted version of Mars Pioneer. 197

6.4 Processing, network latency, and initiated backend instances in the original version of aMazeChallenge under various number of players. 201

6.5 Time taken to load a single, pre-generated, not previously loaded 16x16 chunk as the full size of the state increases. 207

6.6 Results showing how different chunk sizes affect the number of queries, average retrieval time, and average time per generation. 208

6.7 State size requirements for the non-Athlos and Athlos implementations of aMazeChallenge for various maze sizes. 211

6.8 The size of the state in bytes, as serialized by both JSON and Protocol Buffers in Mars Pioneer. 213

6.9 Results obtained from the serialization and de-serialization of objects in Mars Pioneer using JSON and Protocol Buffers, for various numbers of objects. 214

6.10 CK metric measurements for the two implementations of Minesweeper. 222

6.11 CK metric measurements for the two implementations of aMazeChallenge. . . . 222

9.1 Base latency data in the feasibility study experiment. 249

9.2 Maximum board size for each datastore, in cells^2 249

9.3 Latency data for the `/create` service in the feasibility study experiment. . . . 250

9.4 Latency data for the `/join` service in the feasibility study experiment. 250

9.5 Latency data for the `/list` service in the feasibility study experiment. 250

9.6 Latency data for the `/getState` service in the feasibility study experiment. . . 251

9.7 Latency data for the `/play` service in the feasibility study experiment. 251

9.8 Times taken to perform various operations using the Firestorm library and the Firestore API. 277

9.9 Results for the time taken to perform creation operations involving different numbers objects using the Jedis API, Objectis, and Objectis' Hybrid Multi-Threaded mode. 279

9.10 Results for the time taken to perform read operations involving different numbers objects using the Jedis API, Objectis, and Objectis' Hybrid Multi-Threaded mode.279

List of figures

2.1 Infrastructure approaches used as a percentage of the total papers mentioning this aspect. 61

2.2 Choice of infrastructure over time — as derived from the studied works. 63

2.3 Architecture approaches used in terms of frequency of entries. 64

2.4 Choice of software architecture over time as found from the studied works. . . . 65

2.5 Scalability types, as observed from the approaches taken in the related work entries. 66

2.6 Scalability types, as observed from the approaches used in the related work entries. 67

2.7 Approaches used in dealing with persistence in MMOG backends, in each year. . 68

3.1 Illustration of the differences between full board states and partial board states using the AoI concept. In the right figure, the player's AoI is highlighted in red color, with translucent mines being outside of the AoI and not being perceived by the player. 76

3.2 Base latency in each approach, in milliseconds (ms). 78

3.3 Maximum state size supported by each of the datastores used, in cells^2 80

3.4 Average latency for the `/play` service. 81

4.1 The different components of the model, illustrated using the Player type as an example. 94

4.2 The coordinate system used for uniform and grid-based game worlds. 96

4.3 A uniform world, from a bird's-eye view. 97

4.4 A square grid world, from a bird's-eye view. 98

4.5 A hexagonal grid world, from a bird's-eye view. 99

4.6 The interfaces supporting the world models. 99

4.7 The default world model. 100

4.8 A representation of the terrain grid, demonstrating the difference between a cell (red) and a chunk (cyan). 102

4.9 The model and relationships between cells and chunks, used to represent terrain 103

4.10 The default entity model. 105

4.11 The process of defining and generating a new project. 110

4.12 The Athlos API, with pluggable serverless components. 115

4.13 The proposed Athlos architecture. 117

4.14 A closer look at the Game API component. 120

4.15 A closer look at the Persistence API component. 121

4.16 The structure of the event mechanism component, used to schedule events. . . . 123

4.17 The processes involved in utilizing PB in the standard and Athlos approach. . . 129

4.18 The service execution pipeline. 132

4.19 A depiction of a game world's terrain divided into chunks and cells, with each having its own coordinates. 141

4.20 An illustration of the AoI concept in action, when retrieving the partial state. . 148

4.21 The process of retrieving and communicating snapshots of state updates from the backend to the client. 150

4.22 An illustration of the AoE being used to filter state updates. 153

4.23 An overview of the state update process, involving the use of the state update mechanism. 157

4.24 The Athlos project editor (prototype). 163

4.25 An overview of the generation pipeline – the processes involved in the generation of boilerplate code in MMOG projects. 164

4.26 The structure of an Athlos project. 167

5.1 Custom classes defined in Mars Pioneer. 173

5.2 The API defined for Mars Pioneer. 173

5.3 A screenshot of the Mars Pioneer client program, presenting a visualization of the game state to the client. 175

5.4 A screenshot of the aMazeChallenge client during a student competition, held at UCLan Cyprus in 2021. 178

5.5 The game API defined in the new version of aMazeChallenge. 180

5.6 The game API defined for the Minesweeper case study MMOG. 184

5.7 The GUI presented by the Minesweeper client during a simulation using a 10×10 partial state size. 184

6.1 The percentage of the total global response latency taken by different stages in the request-response cycle 194

6.2 Sub-process latency as a percentage of the total service latency in the locally-hosted version of Mars Pioneer. 198

6.3 Sub-process latency as a percentage of the total service latency in the cloud-hosted version of Mars Pioneer. 198

6.4 Total processing latency of local and cloud-hosted approaches as a function of the number of active players and the number of instances launched. 200

6.5 Processing and network latency in aMazeChallenge under varying numbers of players and numbers of instances launched by App Engine. 202

6.6 The amount of time taken to load a single 16x16 chunk as the size of the full state increases. 208

6.7 The effect of chunk size on the number of queries required to fetch a part of the game state and the time taken to generate the chunks. 209

6.8 A comparison between the serialization formats used in the non-Athlos (JSON) and Athlos (Protocol Buffers) implementations of aMazeChallenge, across a range of state sizes. 212

6.9 A comparison between the size of the state when serialized using JSON and Protocol Buffers in Mars Pioneer, across a range of object numbers. 213

6.10 A comparison between the time taken to serialize identical objects when using JSON and Protocol Buffers in Mars Pioneer, across a range of object numbers. . 215

6.11 A comparison between the time taken to de-serialize identical objects when using JSON and Protocol Buffers in Mars Pioneer, across a range of object numbers. . 215

6.12 A comparison between the Athlos and non-Athlos implementations of Minesweeper in terms of source lines of code efforted. 218

9.1 The player and team models. 253

9.2 The default event model. 254

9.3 The default game and world session models. 255

9.4 A Crow's foot diagram presenting the data model of the Athlos framework and the relationships between various types. Attributes are omitted for brevity. . . . 257

9.5 The state API, including methods from the State and World Context classes. . . 259

9.6 A simplified version of the ODM structure created by Firestorm in conjunction with the Athlos model. 270

9.7 A comparison between the Firestore API and the best and worst cases of code used by Firestorm in terms of minimum SLOC required to perform various operations. 278

9.8 A comparison of the time taken to create different numbers of objects when using the Jedis API, or the Objectis library in default or multi-threaded mode. 280

9.9 A comparison of the time taken to read different numbers of objects when using the Jedis API, or the Objectis library in default or multi-threaded mode. 280

9.10 A comparison between ByteSurge (uncompressed and compressed) vs JSON serialization times. 283

9.11 A comparison between ByteSurge (uncompressed and compressed) vs JSON deserialization times. 283

9.12 A comparison between ByteSurge (uncompressed and compressed) vs JSON size. 284

List of listings

4.1 A simplified implementation of a service for a Java-based serverless environment. 137

4.2 Modifying partial states using the 'standard' approach. 150

4.3 Using modifiables to change the partial state. 151

9.1 The implemenation of the WorldSession DAO in Mars Pioneer – MPWorldSessionDAO.java 260

9.2 Customizations made to the `getPartialStateSnapshot()` method – WorldContext.java 261

9.3 Implementation for the BuildFarm action. – BuildFarmWebSocket.java 262

9.4 Implementation of a WebSocket service stub in Mars Pioneer – SellBuildingStub.java265

9.5 The `GetState` service in aMazeChallenge – GetState.java 266

9.6 Implementation of the player entity DAO in aMazeChallenge – PlayerEntityDAO.java 267

Acronyms

ACID – Atomicity Consistency Isolation Durability

AEG – Average Exponential Growth

AES – Advanced Encryption Standard

AMC – aMazeChallenge

AoE – Area of Effect

AoI – Area of Interest

API – Application Programming Interface

AR – Augmented Reality

AWS – Amazon Web Services

BaaS – Backend as a Service

CaaS – Container as a Service

CBO – Coupling Between Objects

CCT – Cloud Computing Technology

CK – Chidamber-Kemerer (metrics)

CRUD – Create, Retrieve, Update, Delete (operations)

CSV – Comma-Separated Value

DAO – Database Access Object

DB – Database

DIT – Depth of Inheritance Tree

ECS – Elastic Container Service

ERP – Enterprise Resource Planning

FaaS – Function as a Service

FPS – First Person Shooter

FPS – Frames Per Second

GaaS – Gaming as a Service

GAE – Google App Engine

GCP – Google Cloud Platform

GPU – Graphics Processing Unit

GQL – Google Query Language

gRPC – Google Remote Procedure Call

HMT – Hybrid Multi-Threaded mode

IaaS – Infrastructure as a Service

IDE – Integrated Development Environment

IoT – Internet of Things

IP – Internet Protocol

JAR – Java Archive

JSON – JavaScript Object Notation

LBF – Load Balancing Factor

LCOM – Lack of Cohesion Methods

LOS – Line of Sight

MD – Message Digest

MMOFPS – Massively Multiplayer Online First Person Shooter

MMOG – Massively Multiplayer Online Game

MMORG – Massively Multiplayer Online Racing Game

MMORPG – Massively Multiplayer Online Role-Playing Game

MMORTS – Massively Multiplayer Online Real-Time Strategy

MMOSG – Massively Multiplayer Online Sports Game

MOBA – Multiplayer Online Battle Arena

MOG – Online Multiplayer Game

MP – Mars Pioneer

MS – Minesweeper

MSDVE – MultiServer Distributed Virtual Environment

MTBF – Mean Time Between Failures

MVE – Modifiable Virtual Environment

NOC – Number of Children

NPC – Non-Player Character

NVE – Networked Virtual Environment

NX – Non-eXtensible (data type)

ODM – Object-Document Mapping

OOP – Object Oriented Programming

ORM – Object-Relational Mapping

P2P – Peer to Peer

PaaS – Platform as a Service

PB – Protocol Buffers

PM – Provisioning Manager

PODO – Plain-Old Data Object

QoE – Quality of Experience

RDBMS – Relational Database Management System

REST – Representational State Transfer

RFC – Response for class value

ROIA – Real-Time Online Application

RPC -– Remote Procedure Calling

RPG – Real-Playing Game

RSA – Rivest-Shamir-Adleman

RTS – Real-Time Strategy

SDLC – Software Development LifeCycle

SHA – Secure Hashing Algorithm

SLOC – Source Line Of Code

SOA – Service-Oriented Architecture

TCP – Transmission Control Protocol

UDP – User Datagram Protocol

URI – Uniform Resource Identifier

VM – Virtual Machine

VR – Virtual Reality

VR-MMOG – Virtual Reality MMOG

WMC – Weight Methods per Class

X – eXtensible (data type)

XML – eXtensible Markup Language

Chapter 1

Introduction

"A problem well stated is a problem half-solved."

Charles Kettering

The ever-increasing drive towards the adoption of cloud computing technology (CCT) has led to unprecedented availability of computing power for individuals and businesses alike (Morgan & Conboy 2013, Alijani et al. 2014). Among others, the cost-effectiveness, broad variety of features, and high service availability offered by this type of technology have made it an attractive option for hosting enterprise applications that must scale to meet business demands (Buyya et al. 2018). For businesses specifically, cloud computing empowers innovations and optimizations in business processes, which are in turn delegated into the creation of new technologies, methods, and tools for developing software, such as mobile and web applications (Boillat & Legner 2014). Enterprise applications that utilize these technologies must be scalable so that they can accommodate fluctuating numbers of customers with a good Quality of Experience (QoE), while also minimizing costs. In turn, the utilization of cloud computing may help businesses of all sizes to improve their profit margin, allowing them to attain economies of scale. To achieve this, enterprise applications provide online services that are often optimized for throughput and parallel execution. This makes them easily scalable, as requests can be handled in parallel and can therefore be offloaded to multiple computing nodes simultaneously. At the same time, these services typically lack the need to access resources synchronously, which enables them

to achieve better performance by leveraging concurrent processing which ultimately translates to a better QoE for customers. The concurrent nature of these services and the business logic behind them allows for their migration to services provided in private or public clouds with relative ease (Attaran & Woods 2019).

1.1 The characteristics of MMOG backends

Resource-demanding, latency-sensitive applications such as the backends of Massively Multiplayer Online Games (MMOGs) have dramatically different characteristics compared to enterprise applications. The sequential processing nature of games imposes constraints and limitations which prohibit them from taking advantage of cloud resources effectively. While other types of applications can take advantage of parallel execution mainly due to disjoint business logic operations, MMOG backends must keep their processing in a single pipeline to enforce a game's rules within their *game loop*. This ultimately leads to decreased opportunities to offload processing on different computing nodes and thus run tasks in parallel to achieve higher performance. In the past, researchers and developers have attempted to mitigate this issue by vertically scaling cloud instances, or configuring each instance to run a specific part of the game. *Session-based* and *room-based* gameplay has allowed online games to be played at specific times, intervals, or with a specific subset of players. Improvements in these concepts have led to the division of shared worlds into *zones*, allowing for gameplay to occur within the same domain, while assigning each zone's processing requirements to a different computing node (Nae, Prodan & Fahringer 2010, Nae, Iosup & Prodan 2010). These innovations have given rise to *Multiplayer Online Battle Arenas* (MOBAs), which can support several hundred simultaneous players (Burger et al. 2016). However, popular MMOGs like World of Warcraft (WoW), face the difficult task of managing a community of millions of players, in common, persistent worlds. To solve this problem, game developers have turned to very powerful dedicated machines that could accommodate "*about 20,000 players*" in each room (Hosseini 2017).

Moreover, MMOG backends are high-performance applications that must deliver a good QoE.

This is especially important in competitive game markets as players may quickly switch to another game if they have recurring negative experiences. The main indicator of good performance in MMOG backends is their ability to provide the players with *soft real-time updates* (Buttazzo et al. 2005) of their game states while managing resource-demanding tasks quickly and efficiently at scale (Google 2021). To deliver state updates under certain latencies –and thus keep the QoE at a satisfactory level– MMOG backends must have access to sufficient raw computing power while also employing a cost-efficient architecture. Such architectures must be flexible and modular, allowing multiple, independent components to work together to provide the best possible equilibrium of cost-efficiency and performance. Ultimately, this points toward the need to have a standardized method for developing high-performance MMOG backends that can offer a good QoE, while also being cost-efficient, and easy to develop and maintain.

Apart from performing rapid updates, MMOG backends must also keep their state consistent across all clients. The need for state consistency is especially important because having an inconsistent state would mean that some players perceive the state of the game differently than others. This may cause players to behave differently, deciding to carry out different actions than those they would normally carry out if they had obtained a consistent state. As a consequence, these actions may be delayed or not executed at all, which can lead to a negative experience while playing. In fact, studies have shown that players generally prefer to lose some of their progress in a game in case of a system failure, as long as the state of the game itself remains consistent (Blackman & Waldo 2009). This highlights how important consistency is in such systems, from the player's point of view. More importantly, state inconsistencies may also lead to unexpected outcomes due to functional defects during logic execution, or even worse, catastrophic consequences that cause the entire system to crash. While such negative experiences cannot be eliminated in such large-scale systems due to the involvement of multiple components and the fact that no piece of software is perfect, they should be very rare. Otherwise, inconsistent MMOG backends run a high risk of losing some of their current players or receiving a bad reputation that may affect future players from ever playing the game.

As mentioned, services provided by MMOGs during gameplay require rapid, successive, and consistent updates to shared states. The behavior and requirements for such systems are

quite different from those in other, business-oriented applications, which mostly perform their operations in read/write data cycles. For example, an *Enterprise Resource Planning* (ERP) system may enable its users to carry out operations such as adding stock-tracking entries, carrying out data analysis, creating reports, and so on. These operations traditionally involve *create* or *read* operations in databases, whereas updates are carried out much less frequently. On the other hand, MMOG backends behave in exactly the opposite way, performing rapid, often simultaneous updates to information during gameplay but only reading new data less frequently and writing on very limited occasions – such as during the creation of a new world or player which are rather rare events (Diao et al. 2015, Diao 2017). This important difference underlines the need for different methodologies and tools to develop MMOG backends. Such types of backends can benefit from cloud-based data persistence systems which are optimized for real-time updates, rather than those traditionally used in business applications.

1.2 Motivation

The popularity of Massively Multiplayer Online Games has surged in the last ten years, and this trend is expected to accelerate in the future. Popular MMOGs like World of Warcraft, Clash of Clans, and World of Tanks have attracted millions of players from all over the world, and new games are being produced at an unprecedented rate (Mordor Intelligence 2022). Even though MMOGs first became popular as MMORPGs (Massively Multiplayer Online Role-Playing Games), today's market has expanded to include almost any genre of online games – such as MMOFPS (MMO First Person Shooter), MMORTS (MMO Real-Time Strategy), and more. Playing online games is described by many as a very popular entertainment activity, which in some cases even leads to addictive behavior (Thakur et al. 2021). Despite fostering addictive behavior in a minority of cases, studies have shown that MMOGs –and online games in general– can have positive effects. Especially for young players, MMOGs can be a motivating factor in learning soft skills such as communication and teamwork, and in promoting creativity and exploration (Schultheiss 2007). MMOGs have also been used in various other contexts, as tools to teach hard skills such as language, mathematics, engineering, and programming (Az-

man & Farhana Dollsaid 2018). It is supported that students can adopt such skills more easily in a contextualized environment such as a game. For example, Minecraft: Education Edition allows students to study a diverse set of topics, including computer science lessons, topics in physics, history, and more (Minecraft 2022). Aside from serving as catalysts in the development of these skills, MMOGs have also been used by researchers to perform various types of experiments. The nature of MMOGs can make them especially useful as research tools, helping to understand how people behave in terms of social interactions, economic activity, and more (Rezvani & Khabiri 2018). It can therefore be argued that the usefulness of MMOGs transcends their original entertainment purpose, and this can be a motivating factor towards improving their development process.

Meanwhile, the MMOG industry has grown to be worth more than $55 billion and is forecast to grow by another $21 billion in the next four years (TechNavio 2022). Interestingly, the online game industry generated a revenue of $152 billion in 2019, which eclipsed the music industry's revenue by tenfold (Donkervliet et al. 2020). The continuous advancements, especially in computing power and development technologies are propelling this industry and enhancing the ways these games are developed, leading to both a higher quantity of games produced and higher-quality games. Game developers strive to enhance the gaming experience by writing code for a diverse set of platforms, and often provide their content using cloud technology. Due to these advancements, some gaming services are now provided entirely through the cloud –known as *cloud gaming*– making them more accessible by lowering hardware costs for the players. Looking into the future, the increased adoption of 5G technology in mobile devices provides new opportunities for mobile-based MMOGs, as 5G offers significantly higher speed, bandwidth, and more importantly, lower latency. The use of 5G may also enable the creation of new types of online games which utilize wearable devices, drones, and Augmented or Virtual Reality (AR/VR) technologies. There is no doubt that the industry of multiplayer online games will expand, not only in popularity and worth but also in terms of the technologies used to develop and maintain these systems.

At the moment, the vast majority of research in MMOG backends has focused on the *Infrastructure as a Service* (IaaS) layer. This is understandable, as IaaS technologies are by

nature more suitable for multiplayer game backends. The fine control, low overhead, and customization options provided by such technologies have made them the most popular option for hosting MMOG backends (Kasenides & Paspallis 2019). Traditionally, game developers have either used private clouds or dedicated servers/clusters rather than public clouds to host their MMOGs. However, such systems are expensive to purchase, set up, and maintain, and are only affordable by large game studios. On the other hand, the use of public clouds offers a lower barrier to entry for smaller game studios and startups due to significantly lower costs in purchasing hardware and requiring a smaller workforce with less technical expertise in areas other than development. Moreover, it appears that there is a slow shift towards higher-level cloud layers such as *Platform-as-a-Service* (PaaS), *Backend-as-a-Service* (BaaS), and *Function-as-a-Service* (FaaS) to host MMOG backends (Shabani et al. 2014, Google 2021). The higher abstraction provided by these layers has the potential to accelerate various software development processes, such as their implementation, deployment, and maintenance. Furthermore, these layers are by definition elastic, which means they can automatically respond to fluctuations in demand by allocating or de-allocating resources when needed, without supervision, and without the need for expertise in setting up load balancing configurations. Despite these advantages, the development of MMOG backends using higher cloud layers and commodity clouds is not standardized in any way, has remained mostly on the sidelines, and is relatively unexplored. Motivated by this gap in knowledge, this thesis aims to explore, develop, and evaluate models, methods, and tools that facilitate the development of MMOG backends running on commodity cloud platforms and at higher computing layers.

1.3 Scope and Objectives

This thesis attempts to investigate how resource-intensive and latency-sensitive applications like MMOG backends can be developed for and deployed on commodity cloud platforms. It focuses primarily on higher cloud computing layers, known as *serverless computing*, such as PaaS, BaaS, and FaaS, and aims to propose solutions to various problems that are associated with the development of MMOGs, such as scalability and state consistency. While the development

processes can also be used to develop solutions for the IaaS layer, this is only studied to a limited extent. In addition, only very minor considerations are made for the presentation layer of MMOGs, which is another field of its own that involves computer graphics, Human-Computer Interaction, and more. The focal point of this research is how information can be modeled and abstracted in such systems to enable their development for a variety of commodity cloud platforms, as well as improve their overall development process.

The large variety of game genres and the existence of a broad set of technologies and approaches in a multitude of areas related to game development make it impossible to predict how each game can best be developed. This thesis focuses on games that feature fully persistent, scalable states, and which typically have to support large numbers of concurrent players. Such games usually fall under specific genres that do not require very low latency, such as MMORPG and MMORTS. While the development methodologies and tools developed in this research can also be used for other types of low-latency game genres such as MMOFPS, MMOSG (MMO Sports Game), or MMORG (MMO Racing Game), these are only explored and evaluated to a limited extent.

Research objective 1: Assessing the state of the art in MMOG backend development

To develop an understanding of the different aspects of the game development process, the first research objective of this thesis is to gather information on how MMOG backends have been developed based on past studies. The purpose of this is to identify the various characteristics of such systems, the aspects involved during the process of development, and the challenges and opportunities arising from different development approaches. Information gathered from related works on these approaches can be systematically categorized and analyzed to compare the advantages and disadvantages of each approach, ultimately leading to a better understanding of the core principles involved in MMOG development.

Research objective 2: Examining the feasibility of using public clouds for hosting MMOG backends

The second research objective of this thesis is the assessment of various public cloud technologies and how these can be used to enable MMOG backends. While technologies like BaaS have already been used for secondary functionalities like analytics and scorekeeping, their utilization to fully power MMOG backends in terms of gameplay remains relatively unexplored. Thus, to prove the feasibility of the proposed research direction, it is necessary to explore how such technologies can be used for core tasks like state management and updates, to what extent they can be utilized, and how effective they are in providing the necessary features and performance. To achieve this objective, several experimental solutions are realized, utilizing various commodity cloud services to power an MMOG backend prototype, and their performance is evaluated.

Research objective 3: Creating a software development framework for developing scalable MMOG backends hosted on commodity clouds

The third research objective of the thesis is the development of models, methods, and tools which can be used to develop MMOG backends that run on commodity cloud platforms. These may be incorporated into a framework that can be used by game developers to create MMOG backend prototypes that leverage the resources of public clouds. The development of a common game model aims to abstract the development process by decoupling state management and game logic, which are game-specific processes, from other development processes which are common to all games – such as database management, networking, and so on. Building on this model, new methods can be proposed to standardize the development of MMOG backends that feature scalable states, utilize resources as efficiently as possible, and can attain the necessary level of performance. To allow developers to utilize these novel models and methods, various types of tools can be created. For instance, a common game development Application Programming Interface (API) can assist developers in utilizing the model and various other abstractions, while also speeding up the development process. In addition, various types of tools can enable the creation of approach-independent game definitions which leverage the proposed

methods to produce prototype MMOG backends for specific development environments. To complement these tools, utility libraries can also be developed for specific environments and cloud services. Such tools may enable MMOG backends to be rapidly developed by allowing developers to write code that utilizes these services more effectively.

Research objective 4: Evaluating the proposed methods and tools

The fourth objective of the thesis is the evaluation of the developed models, methods, and tools through the use of MMOG backend prototype case studies. These case studies can help determine the feasibility of the proposed approach while ensuring that the contributions of the thesis capture actual development needs. More importantly, these case studies can be used to evaluate the proposed approach. Firstly, the evaluation should quantify the performance of MMOG backends that are developed using the new framework, which can involve the measurement of important metrics such as latency under varying loads and configurations. Secondly, it should investigate if MMOG backends developed using the proposed framework and test whether they can achieve the necessary scalability by measuring how the state loading times are affected by changes in the game worlds. Thirdly, the evaluation should assess how the development is affected by measuring the effort required to produce MMOG backends and the quality of the produced code. To achieve these evaluation objectives, the following set of hypotheses are defined:

- **H1** MMOGs that are hosted in *serverless cloud environments* (Schleier-Smith et al. 2021) and utilize the proposed 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 the framework.
- **H2** MMOGs based on the proposed framework and hosted on serverless clouds can sustain a higher total number of active players than single-machine, non-framework approaches, under the threshold latency of 1000ms.

H3 MMOGs that utilize the proposed framework are able to feature very large and expandable game states (within the limits of the hardware resources being utilized).

H4 When using the proposed framework, the time taken to retrieve a sub-state of a game world remains constant regardless of the world's full size.

H5 The development of scalable MMOG backends using the proposed framework simplifies the development process and results to lower effort and time taken to develop an MMOG.

H6 The proposed approach produces high-quality, readable, maintainable, and reusable code.

Additionally, this research aims to address several technical challenges that are identified through a set of questions which drive the exploration and development of new models, methods, and tools:

- The use of a common model would improve the development process by enabling code reuse, better maintainability, and component modularity. However, games are vastly different from each other in terms of gameplay and modeling requirements. *Can a generic model be created and used for all types of games and game genres?*
- Serverless environments have technical limitations, such as bounded service execution time, limited bi-directional communication methods, etc., which hinder the development of MMOG backends. *How can these limitations be dealt with and what types of methods and tools must be developed to enable MMOG backends to run in these environments?*
- Especially in cloud environments, there is a tradeoff between consistency and performance, which are both requirements of MMOGs. *How can these two attributes be balanced to provide both good performance and adequate consistency to ensure a good QoE?*
- In terms of persistence, many have opted to use key-value stores or caches (Kasenides & Paspallis 2019), which are inherently scalable information storage methods. *Can these types of persistence be adapted for gaming workloads, and if so, do they need to be complemented with new methods and tools?*

- Serverless environments offer inherent elasticity which eliminates the need to create custom load balancing and resource provisioning tools. *Are these built-in tools adequate for developing and servicing MMOG backends with a good QoE?*

- Online games, despite their differences in gameplay, have many common architectural components. *Is it possible to design a framework that utilizes the same architecture for a variety of games?*

- Commodity clouds provide both IaaS and serverless services. While it may be possible to support some games on serverless environments, more demanding games may not attain the necessary performance. Conversely, IaaS products may offer less scalability, but better performance, which means that the use of each technology may depend on the game type. *Is it possible to support the development of MMOG backends on both IaaS and serverless environments with the same models, methods, and tools?*

1.4 Contributions

The primary contributions of this thesis can be traced to the research objectives discussed in the previous section. Firstly, the thesis presents a holistic view of the state of the art in the technologies, methodologies, and approaches used to enable MMOG backends. It outlines important concepts and principles that are mentioned in related works and attempts to provide insights regarding their usefulness. Furthermore, it identifies criteria that are important in the development of MMOG backends and uses these to categorize the presented methodologies and approaches. The aims of this categorization are multifold. We can determine which approaches are the most popular and why, which helps us understand how research around this topic has evolved in the past. Secondly, we can critically evaluate these approaches by comparing the advantages and disadvantages of each, allowing us to point out their strengths and weaknesses. This may be especially useful to game developers who need to decide which approach to use. Finally, by looking at these approaches in a timeline, we can determine future trends in research, allowing us to identify potential research directions. While some works have previously

attempted to systematically study some aspects of MMOG backend development, to the extent of the author's knowledge, none of them have reached the coverage and scope of what is being presented in this thesis.

Another contribution of this thesis is a study of the applicability of commodity cloud platforms and their associated services in terms of hosting MMOG backends. This contribution sheds light on the suitability of public cloud services in various aspects such as performance, persistence, and state management, and allows us to compare them through a proof-of-concept MMOG. Through this comparison, this thesis presents the benefits and risks associated with the use of services provided by three popular public clouds, identifies which services perform better under certain circumstances, and determines if any abstractions or methods can be used to unlock their full potential.

One of the main contributions of this thesis is the proposition of novel models, methods, and tools with which scalable MMOG backends can be developed to run on commodity clouds. Herein, a framework named Athlos is presented, which incorporates these models, methods, and tools, and enables developers to utilize them to develop scalable MMOG backends. The Athlos framework first allows the creation of approach-agnostic game definitions that can use a default and abstract game model. This model can be expanded to include game-specific declarations using code generation, which significantly cuts down the time and effort required to develop these applications. Furthermore, Athlos utilizes various methods that a) standardize the development process to a large extent, b) enable MMOG backends to attain the necessary performance and scalability, c) eliminate the need to spend time on common game development processes such as networking and data management, allowing the developer to focus on game-specific logic, and d) promote the evaluation of various approaches through the use of a modular architecture, in which many components can be swapped seamlessly without requiring many code updates.

These positive aspects are provided through several innovative technical contributions within this development methodology. For instance, the chunk-based state management method which is described in section 4.3.5 enables virtual worlds to reach vast scales while using commod-

ity cloud database systems. Various other abstractions, such as the State, Persistence, and Communication APIs, the serialization and state-update mechanisms, and so on, provide standardized solutions to common problems that are encountered across many types of MMOGs and therefore eliminate the cost of developing customized solutions.

Additional contributions of this thesis include the chunk-based state management method which enables virtual worlds to reach vast scales while using commodity cloud database systems. Other abstractions, such as the State, Persistence, and Communication APIs, the serialization and state-update mechanisms, and so on, provide standardized solutions to common problems that are encountered across many types of MMOGs and therefore eliminate the cost of developing customized solutions.

Another important contribution of this thesis is the evaluation of the proposed approach through a series of experiments designed to test various aspects of different MMOG backends that are produced using the framework. The results presented in this thesis show that Athlos and its associated models, methods, and tools, can be effectively coupled with various types of serverless technologies hosted in commodity clouds, enabling MMOG backends to take advantage of the proposed methods to achieve significantly larger states and serve a large number of concurrent players compared to dedicated approaches. Such improvements in performance and scalability are also illustrated by results, based on which one can observe that MMOGs developed with Athlos have the capacity to serve larger numbers of concurrent players under certain latency thresholds, particularly when coupled with serverless computing services. Ultimately, the patterns emerging from these results can be used to further study the behavior of MMOG backends and to motivate their evaluation at commercial scales in the future. Moreover, a limited code evaluation shows that Athlos allows developers to construct MMOG backends with significantly less effort than without Athlos. Through the evaluation of this framework, this thesis provides important insights into how the process of developing MMOG backends has improved and can be further improved in the future, enabling them to support increasingly larger, consistent states, and achieve a higher QoE while utilizing the serverless cloud paradigm.

1.5 Publications

The bulk of the material presented in this thesis has been previously accepted for publication in various conferences and journals. While most of these publications are directly related to this thesis, some are only indirectly related.

The majority of the work presented in Section 2 has been published in the systematic mapping study of MMOG backend architectures (Kasenides & Paspallis 2019). This study presents the state of the art in research for the development of MMOG backends and aims to identify criteria and aspects that are important for the development of these systems, unveil the challenges and opportunities arising from each approach, enable comparisons to be made between them, and ultimately reveal gaps in current knowledge.

The work discussed in section 3 has previously been published in a paper which explores if, and to what extent various commodity cloud-based approaches can be utilized to enable MMOG backends (Kasenides & Paspallis 2020).

The material discussed in Sections 4 and 6 has been published in a journal article that describes the models, methods, and tools that can be used to enable MMOG backends on commodity cloud platforms, incorporated in a framework called Athlos (Kasenides & Paspallis 2022).

Some of the contributions of this thesis are also utilized in indirectly related studies. The first of these studies is a paper that presents aMazeChallenge, a multiplayer educational programming game that discusses how such games can be developed to use the resources of public cloud platforms (Kasenides & Paspallis 2021). A second paper presents an architecture that can be used to develop multiplayer educational games like aMazeChallenge by leveraging mobile technologies and commodity clouds (Paspallis et al. 2022). Both of these studies utilize several contributions of this thesis to enable gameplay for the educational game being presented.

1.6 Statement of Originality

I hereby declare that the material and work submitted for this thesis is my own work and that, to the best of my knowledge, it does not contain material from other resources, except the ones that are being mentioned and acknowledged. I also declare that none of this material has been previously submitted for a degree at any university.

1.7 Thesis structure

This chapter introduced the research topic by establishing a context and background, and by explaining the general concepts related to the area of study. Various problems associated with the development of MMOG backends and the characteristics and peculiarities of MMOG backends are discussed. Finally, it defines the scope and research objectives of the thesis, along with several hypotheses and questions to be explored in the following chapters. The rest of the thesis is organized as follows:

Chapter 2 presents a review of the state of the art in MMOG backend development and identifies important aspects related to their deployment on commodity clouds. These aspects are used as criteria to perform a systematic mapping study of past research works, to analyze the different approaches, identify their advantages and limitations, and ultimately assimilate the opportunities and challenges arising from each approach. Through this analysis, trends in research are established through patterns in the methods utilized by researchers and developers. The discovery of these trends further motivates this thesis by identifying gaps in the research and potential directions that have not been explored by previous works.

Chapter 3 then reports an exploration of three popular commodity clouds in terms of supporting MMOG backends. It first identifies the facilities provided by these clouds and describes the implementation of a simple prototype MMOG backend. The prototype is then deployed and used to evaluate the performance and suitability of the studied clouds to host such applications. Through this feasibility study, various challenges and limitations of these platforms are

uncovered, posing additional questions and further motivating the creation of a new software development methodology.

Chapter 4 proposes new models, methods, and tools which can be used to develop scalable MMOG backends on commodity clouds. This is first motivated by discussing the limitations of existing methods and tools, followed by the presentation of a conceptual MMOG case study that aids the development of various elements, such as a common model and architecture for MMOG backends. The models and methods described in this chapter are incorporated in a software development framework called Athlos which utilizes also includes guidelines and specifications that aim to standardize the development of MMOG backends hosted on commodity clouds – ultimately leading to a more efficient software engineering process.

This framework is empirically evaluated in chapter 5, by developing and showcasing three prototype MMOG backends. In this chapter, the suitability of the proposed approach is investigated through the experience of developing these prototypes. Some of the technical questions posed in chapters 1 and 3 are also explored, helping to establish a proof-of-concept for the proposed development methodology and address its limitations.

Chapter 6 introduces the research methodology which involves a quantitative evaluation of the proposed development approach. This is mainly centered around the hypotheses, questions listed in section 1, and other isolated experiments conducted to explore the usefulness of specific methods or tools or to obtain information about the behavior of MMOG backends. The proposed methods and tools are evaluated in terms of performance and scalability, the development effort required to produce MMOG backends, and the quality of the code generated by the framework. Results obtained from numerous experiments show that MMOG backends can sustain larger player bases below certain latency thresholds, can allow game states to be scaled beyond the normal capabilities of the underlying technologies, and be managed efficiently when developed using the proposed framework rather than generic approaches. The results also suggest that the proposed framework may significantly reduce development effort, even though the quality of the produced code may not be optimal.

Chapter 7 discusses the results obtained from the evaluation with respect to the hypotheses, and

attempts to answer the questions outlined in sections 1.3 and 3.7. This chapter also identifies and discusses the limitations of the development and research methodologies used.

This thesis concludes with chapter 8, which presents a summary of its contributions and contents. This is followed by a general discussion on the impact of this research with respect to the broader areas of study as well as practical domains. Finally, the thesis closes with a discussion of future work and possible ways to address known limitations and further explore the identified research problems.

Chapter 2

Related work

> "*If I have seen further it is by standing on the shoulders of giants.*"
>
> Sir Isaac Newton

2.1 Introduction

The first online games were introduced in the 1970s and, despite being revolutionary for their time, featured only basic rules, controls, graphics, and mechanics, and communicated their states over serial cable (Thompson 2004). These games later evolved to support gameplay over the forefather of the modern Internet, the ARPANET, allowing players to play in primitive versions of virtual worlds. Online games started gaining traction in the 1990s when access to the Internet became more widespread. During this decade, arcade games such as Sega's OutRunners started supporting simultaneous online gameplay for up to eight players. During the late half of the 1990s, MMORPGs exploded in popularity due to the rise of the personal computer, with popular titles like Ultima Online, EverQuest, and more, which allowed more advanced features such as dedicated online services for gameplay, e-commerce, and media sharing (Bartle 2009). With the introduction of online gaming on consoles such as Sony's PlayStation 2 and Microsoft's Xbox in the 2000s, the popularity of online games skyrocketed to new levels. During this time, gaming networks facilitated additional features and were consolidated into integrated

platforms. The rise of cloud computing in the mid-2010s started a new era in online games with researchers scrambling to find ways to utilize this technology to host MMOGs that feature ever-larger virtual worlds and numbers of players. This trend continues to this day, and it would be safe to assume that the pace of research will have to keep up with the latest technological advancements.

This section discusses the state of the art in the development and deployment of MMOG backends, starting from the late 2000s when MMOGs became increasingly popular. Firstly, it investigates the approaches described in various studies and identifies important aspects that researchers have focused on when developing MMOG backends. These aspects are used as criteria, and a categorization is made out of several selected studies that are considered landmarks in supporting a specific approach. The different approaches in each criterion can be compared to expose their advantages and disadvantages, as well as the opportunities and challenges that are encountered in each approach. Using this analysis, research trends can be revealed by spotting emerging patterns in the use of various approaches and technologies. These trends can point towards gaps in research, and potential directions that justify further exploration, based on the guidelines defined by Keele et al. (2007).

2.2 Research questions

To explore the state of the art in the area of MMOG development and deployment, this thesis aims to address the following research questions:

1. What are the main challenges in developing MMOG backends?
2. Which criteria/aspects are the most relevant to categorize studies into groups?
3. What are the research trends over time, in terms of the approaches used for MMOG backends?
4. Are there indications of alternative, promising research directions for realizing MMOG backends?

2.3 Search strategy

The search for related work was done online, using services provided by digital libraries, and was partitioned into three stages. In the first stage, the Google Scholar search engine was used to search for keywords and term combinations that are related to this topic. Examples of these keywords include *MMOG*, *Online*, *Games*, *Virtual world*, *Infrastructure*, *Architecture*, *Cloud*, *Distributed*, *Computing*, *Dedicated*, and many more. This stage discovered a wide set of related works. In addition, digital libraries that yielded the most relevant results were noted for use in the second stage. During the second stage, the search was expanded to specific digital libraries, including *ACM*, *Springer*, *IEEE*, *ScienceDirect*, and *Elsevier*. The same keywords as those used in stage 1 were used to search for relevant literature from these libraries, leading to additional related works being discovered. In the final stage, the search was further expanded to include relevant publications that were referenced from the original list of papers found in stages 1 and 2.

2.4 Criteria

The pool of available studies resulting from this search is relatively large, and many of these studies are directly linked with regards to the approach being used. Therefore, certain studies must be prioritized over others, firstly to avoid studying irrelevant work, and secondly, to keep the number of studies to a manageable number. To achieve this, certain inclusion and exclusion criteria are pre-defined, to help ensure that the selection process remains as objective as possible.

2.4.1 Inclusion criteria

The following criteria are used to determine if a study is relevant and should be included:

1. The paper is related to MMOGs and/or other similar large-scale, distributed systems.

2. The paper relates to cloud technology which could be applied to developing MMOG backends.

3. The paper touches on at least one of the identified research questions, either directly or indirectly.

2.4.2 Exclusion criteria

To exclude irrelevant or repetitive studies, the following criteria are used:

1. The paper is not related to software engineering, software architecture, or software technology.

2. The paper does not provide details for any of the topics of interest.

3. The paper is not relevant to any of the research questions.

4. The paper was not published in a peer-reviewed journal or conference proceedings.

5. The full paper is not available.

2.5 Review process and data collection

To select only the most significant publications and determine their usefulness based on the criteria mentioned above, the abstracts of all studies were reviewed. Each study was then assigned a level of significance based on the criteria, with papers meeting multiple criteria being assigned a higher significance, as they are inherently more useful for answering the research questions. To extract more information from the selected studies, the following questions were used as a guide to express their relevance to the research questions:

1. Do the authors identify any challenges in their methodologies? Are these relevant to the study?

2. Is the paper focused on a certain area of MMOG development/deployment? How are the described methods evaluated?

3. What is the approach utilized by the authors to implement the backend of their MMOG? Do they use any specific tools? In what context was their study conducted? When was it published?

4. When data from all studies are collected and analyzed, is there any emerging correlation between time and the methods used? Are there any *gaps* that have not been explored so far?

To answer these questions the full body of the selected papers was reviewed to identify the approaches, technologies, and evaluation methods used. The total number of resources considered during the review process was 176, with each publication being reviewed by reading its abstract and assigned a relevance score based on the inclusion and exclusion criteria. This round of review excluded a large number of low-importance papers. In the second review round, the full text of each resource was read to re-evaluate its relevance and importance. From this round, more papers were excluded, leaving the total number of resources under consideration to 50. Similar studies published by the same authors or featuring directly related work are grouped to limit the number of entries and to keep the review process fair. The final number of entries is 41.

2.6 Aspect selection

From the data collected during the review of the state of the art, several non-functional aspects can be identified as important for the development of MMOGs and their deployment on cloud platforms. Based on the frequency of studies mentioning these aspects, table 2.1 presents these criteria in descending order of importance:

It is worth noting that a related study (Chu 2008) also found results that are consistent with those presented in table 2.1.

Aspect	Frequency (number of papers)
Infrastructure	13
Architecture	12
Performance	7
Scalability	6
Persistence	4
Security	4

Table 2.1: Different aspects identified during the review process, sorted in descending order of importance based on the number of papers mentioning them.

2.7 Approach categorization

The aspects identified above point towards broad areas that must be considered when developing MMOG backends. Each of these aspects acts category in its own right, however, each paper must be further categorized to identify the specific approaches used in these aspects.

2.7.1 Infrastructure

The infrastructure aspect identifies the type of infrastructure being used to host the MMOG backend in each study. In the context of this thesis, *infrastructure* is defined as *the set of technology components –physical or otherwise– that are used to enable MMOG backend services.* Infrastructure approaches can be broken into four categories:

- **Dedicated (D)**: Use of network facilities that are specifically purposed to enable one type of application and require direct management at the hardware level.
- **Private clouds (PrC)**: Use of proprietary clouds that are built and maintained privately and can offer higher availability, scalability, etc.
- **Public clouds (PuC)**: Use of public clouds which are owned by a third party. Services are leased to the game provider for a given price model. These are further categorized as IaaS and Serverless (SL). The serverless approach may encompass layers like PaaS, BaaS, and FaaS.

- **Hybrid clouds (HC)**: Hybrid clouds use a combination of private and public cloud services to enable MMOG backends.
- **Unknown (U)**: Unknown—that is, no information could be identified regarding this aspect.

2.7.2 Architecture

The architecture aspect identifies the communication architecture used in each included study. An architecture *defines how the components of a system are configured to interact and communicate with each other to partition and carry out their workload.* These are categorized as:

- **Client-Server (CS)**: This approach offloads most of the workload on a powerful, central server that receives requests, performs processing, and provides responses to other computers, which act as clients.
- **Peer-to-peer (P2P)**: P2P is a de-centralized approach that splits the workload among equipotent peers in a network and uses algorithms to synchronize processing. Each peer performs a part of the workload and can send data to other peers.
- **Hybrid (H)**: A hybrid approach utilizes both the client-server and peer-to-peer architectures at different levels in the architecture.
- **Unknown (U)**: Unknown—that is, no information could be identified regarding this aspect.

2.7.3 Performance

The performance aspect *explores the importance of performance and how each study conducted a performance evaluation for their proposed approaches.* The related works use different methods and tools for their performance evaluation, which are shown below:

Methods:

- **Simulation (S)**: The authors have used computer simulations to conduct their experiments and evaluate the performance of their solutions.
- **Modelling (M)**: The authors provide mathematical or computational models and utilize them to predict the performance of their solution.
- **Unknown (U)**: Unknown—that is, no information could be identified regarding this aspect.

Tools:

- **Pre-existing MMOGs (P)**: Studies that have been identified to use pre-existing MMOGs to carry out their evaluation.
- **Other (O)**: Use of other types of applications or case studies to carry out an evaluation.

2.7.4 Scalability

Scalability *is the measure of how capable and efficient a system is at being able to serve a changing number of concurrent users and state sizes.* This aspect is used to measure the capability of a system in terms of utilizing resources, providing versatility, and cost-efficiency. Related works are categorized into the following groups:

- **Not scalable (NS)**: This category describes systems that always use the same amount of resources, and thus have hard limits on the scale they can support before performance becomes severely degraded.
- **Manually Scalable (MS)**: These systems can respond to changing workloads but depend on supervision from a system administrator—and usually the manual procurement and installation of hardware.

- **Elastic—or automatically scalable—(E)**: Such solutions respond to changing workloads by automatically allocating resources without any supervision.
- **Unknown (U)**: Unknown—that is, no information could be identified regarding this aspect.

2.7.5 Persistence

Persistence is defined as *the ability of an MMOG backend to persist information, either temporarily or permanently, to facilitate gameplay and its related game services.* A wide variety of persistence systems are used in the related papers, which can be categorized into three main groups:

- **Relational Databases (R)**: Relational Database Management Systems (RDBMSs) store data as rows and columns in a table and use the Structured Query Language (SQL) to describe relationships between them.
- **Non-relational datastores (N)**: Non-relational database systems which have no schema and rely on collections and documents to organize data instead of using tables.
- **Unknown (U)**: Unknown—that is, no information could be identified regarding this aspect.

2.7.6 Security

Finally, the security aspect is used to categorize the level of security provided in each approach discussed in the related works. This aspect involves *developing countermeasures against threats to ensure data confidentiality, state integrity, and service availability.* A handful of studies have used security practices, which are categorized as:

- **Loose security (L)**: Defines practices to secure a solution that are not controlled at the architecture level.

- **Tight security (T)**: Defines security practices that are controlled at the architecture level.

- **Unknown (U)**: Unknown—that is, no information could be identified regarding this aspect.

Apart from these non-functional aspects, several studies also identify functional features of MMOGs. A white paper by Google (2018) incorporates several auxiliary features that are typical in-game platforms, such as *matchmaking and lobbies*, *leaderboards*, *social feeds, chat, and presence systems*, *team formation*, *user and team profiles*, *analytics*, and more. Others also mention the use of developer portals, which can be used by developers to "*interact with the game platform*" (Shaikh et al. 2006). Such portals can be used to check the status of a backend, view resource utilization, deploy game resources, manage information, set up server policies and configurations, and issue patches or content downloads. The latter is identified as an issue due to the high frequency and size of patches being issued for each game, as well as the *flash-crowd* behavior of players when these patches are released. To mediate this problem, Shaikh et al. (2006) propose the use of peer-to-peer architectures to "*deliver content to users quickly while preserving bandwidth at [...] the content servers*".

2.8 Literature review

Due to the wide range of approaches used in this domain, the aspects presented in table 2.2 are used to divide the related work into categories. In this section, these categories are used to present the state of the art concerning each aspect.

2.8.1 Infrastructure

Dedicated infrastructure

Traditionally, game developers have used dedicated infrastructure to deploy MMOG backends. Using this approach, it is necessary to install, configure, and maintain a private infrastructure that is dedicated to just running a specific game (Shaikh et al. 2006). For the developers of MMOGs, this type of infrastructure presents several challenges. For example, the developers of *World of Warcraft (WoW)*, a popular MMORPG, had to design, operate, and maintain a large infrastructure with hundreds of servers which presented them with several problems (Nae et al. 2011). Firstly, such infrastructures are relatively static and cannot be easily scaled without manual changes to the hardware. In addition, finding the best possible hosting solution and configurations is problematic and often requires lengthy experimentation processes. This is further exacerbated by the fact that risks must be taken to make the necessary investments to support this infrastructure, both in terms of equipment and manpower (Shaikh et al. 2004).

Dedicated infrastructures typically consist of four basic components that allow the system to carry out the operations needed to run an MMOG backend (Chu 2008). A dedicated infrastructure may be composed of one or more *game server(s)*, which are responsible for executing game logic and player actions. These servers may be connected to share information for different parts of the game world. Some of these servers may be secure to protect sensitive processes, while others may be publicly accessible to enable communication with the clients. To complement this, a *database server* is used to persistently save information in a database. Players can interact with the game using *game client* programs which run on the player's machine. Chu

(2008) mentions the use of a *Web application server*, which integrates the functionality of clients with the game servers to allow online actions and gameplay to take place. This infrastructure is designed to allow different machines in the network to carry out a specific set of tasks, based on their specialization. For instance, database transactions and queries are handled by the database server, while game logic, action validation, and so on, are carried out by the game server. Game servers that are publicly accessible allow the clients to interface with the game's functions while restricting access to the secure servers.

A more recent study by Barri et al. (2016) identifies that dedicated infrastructure allows full customization and provides the necessary performance for an MMOG backend even though it has several limitations, the most prominent of which are its limited scalability and availability. The authors propose improving this type of infrastructure by utilizing dedicated clusters of servers to enable MMOG backends so that scalability issues can be overcome. This also has the potential of making the system more redundant and enabling a higher level of availability. While this brings this type of infrastructure closer to being a cloud, it still lacks several characteristics to be included within this category.

Private clouds

The use of dedicated infrastructures in MMOG backends has receded in the last ten years, giving way to private/proprietary clouds. As dedicated server clusters and private clouds have some common characteristics, we use the following attributes to distinguish between them (LeadingEdgeTech.co.uk 2019):

- High availability: Cloud systems offer high availability, usually 99% or higher.
- Elasticity: In cloud-based systems, there are enough resources and provisioning policies for the system to scale up or down, following the on-demand model.
- Virtualization: Access to the infrastructure is not done physically/directly but through a virtualization medium.

- Automation: A cloud system automates a number of administrative processes such as storage configuration, security policies, and more.

A study by Nae, Iosup & Prodan (2010) uses a private cloud infrastructure to host an MMOG backend and proposes a dynamic resource provisioning solution that aims to solve the problem of resource over-provisioning and "*low-cost market joining problems*" that are usually encountered in dedicated infrastructures. To achieve this, they serve game operators simultaneously using geographically-distributed data centers that offer computing resources (such as CPU, memory, etc.). These resources can be requested by the game operators, and depending on each data center's policy requests can be either served immediately or queued. In either case, resources can then be allocated to each game operator. To achieve this, the authors use virtualization and create Virtual Machine (VM) images with the required software, which is a relatively time-efficient process as these can be easily created and "*deployed on all supported platforms*". The authors support this type of infrastructure, and argue that "*compute clouds provide generic functionality for on-demand hosting and provisioning of resources*" and that "*cloud computing [has] the potential to eliminate the scalability barriers in MMOG hosting through scaling by credit card*". Finally, through the encouraging results of their study, they attest that "*the advantages of virtualization are rather important when using heterogeneous computing resources*".

The on-demand model is further supported by Shaikh et al. (2004), who describe an online game platform that also follows this paradigm. Based on open standards and off-the-shelf software, the platform uses resource virtualization through a server cluster that is configured to host MMOG backends. Their platform is designed with a layered architecture that ranges from low-level infrastructure to application-level services. At the infrastructure level, it consists of clusters of shared game servers, database servers, *content delivery servers*, and so on, which can be expanded to support larger numbers of players or to deploy games on multiple hosts. At this layer, the system implements non-game-specific functions that manage and monitor the network, provision server resources, and more. These can be extended to contain code from different software stacks if necessary. Perhaps one of the most important novelties presented in this article, and for this type of infrastructure in general, is the *Provisioning Manager*

(PM). The PM is a service that can automatically manage and provision game servers by dynamically collecting performance and availability metrics. This can allow the system to respond to changes in demand through the implementation of performance models which enable it to make real-time decisions on when and how resources must be allocated or de-allocated. Through the PM, game developers can use these load prediction models to create different resource allocation configurations to leverage the advantages of resource virtualization in a standardized environment.

Another study by Dhib, Zangar, Tabbane & Boussetta (2016) shows how private cloud infrastructure can be used to test MMOG backends to ensure they meet the necessary Quality of Experience (QoE). In their infrastructure, they distributed physical servers over multiple data centers, with each one hosting several VMs. Each VM runs its own game server, which is responsible for managing "*a limited area*" of a game's world. Using this approach, the authors are able to reduce the load on each server. In addition, using a pricing model similar to Amazon's EC2, they can calculate the potential costs for the allocated resources. This is quite useful, as it can allow game developers to experiment with different parameters to determine which configurations are more economical. Through several experiments, it is shown that by using virtualization in a private cloud, "*the cost per player decreases smoothly when more players get connected*" while the QoE remains at a "*minimal threshold*".

Through the innovations and insights presented in these studies, we can take for granted that private clouds not only provide a feasible way to host MMOG backends, but also present multiple advantages for game developers, the games they create, and ultimately, the players.

Public clouds

Public clouds, also known as *commercial* or *commodity* clouds provide infrastructure that is owned by a *cloud provider*. This infrastructure is used to make various types of cloud services available to consumers based on pre-defined pricing units and models. Amazon, Google, and Microsoft are some of the most popular organizations that host public cloud platforms, which include a large variety of features and services.

These clouds "*aim to realize economies of scale and increased utilization by sharing resources or services as available through technologies such as virtualization and multitenancy*" (Mishra et al. 2014). In the context of MMOG backends, Mishra et al. (2014) argue that public clouds can be utilized by using techniques to offload computationally-intensive tasks from the client devices to cloud-based servers. One of the advantages introduced by this type of infrastructure is the ability to provide resources at much lower costs compared to other types of clouds or dedicated infrastructure. Najaran & Krasic (2010) further claim that the cost per player is significantly lower: "*just a few cents per player per hour*". MMOG backends can be enabled on public clouds through multiple architectures and can support any type of virtual world. For instance, Najaran & Krasic (2010) have utilized Amazon's EC2 infrastructure with a peer-to-peer (P2P) architecture to host a fast-paced, FPS-style game. Using this approach, they were able to distribute the load "*amongst multiple nodes*" during gameplay, and therefore achieve better performance. To evaluate their approach, they create simulated players that randomly walk around a virtual world. Through their experiment, they were able to measure the performance of both the client machines, as well as Amazon's EC2 instances, for which they report very encouraging results. In addition, they were able to measure the system's ability to scale, for which they concluded that it is able to scale "*an order of magnitude more players than the state of the art FPS game servers currently support*".

The use of public clouds brings several advantages for the development of MMOG backends but also introduces some problems. A barrier when working with public clouds to enable MMOG backends is the fact that the resources provided are located in a cloud provider's data center. These data centers are rarely located close to users, which results in "*large communication latency in the network infrastructure*" (Mishra et al. 2014). A large latency is prohibitive, especially for some fast-paced MMOGs, which means that this issue must be addressed –either at the resource level or software level– before these types of games can be hosted on such infrastructure. Several solutions which are based on Fog and Edge Computing technologies have emerged to combat this problem. For instance, Cloudlets, which are discussed by Satyanarayanan et al. (2009), can alleviate this problem by bringing computational resources closer to the player to achieve low-latency, real-time gameplay.

Although latency is perhaps the most significant problem area when it comes to hosting MMOG backends on public clouds, some have also explored the trade-off between latency and resource allocation cost. The problem of resource provisioning –or rather over or under-provisioning– is also important as it can determine how economical games will be in the long run. To address this problem, Dhib, Zangar, Tabbane & Boussetta (2016) propose an architecture that uses a multi-layered architecture on top of public cloud infrastructure to host an MMOG backend. Their approach is to utilize a dynamic resource allocation strategy that only allocates resources when necessary in order to minimize the cost while also keeping the latency below a certain threshold. The experiments in this study have shown that this approach can maintain the trade-off and be more economical than other strategies such as over-provisioning or under-provisioning. This research is further enhanced by the development of a new model that captures the intrinsic trade-off between response delays and the corresponding costs. Dhib et al. (2017) have also proposed a VM placement algorithm that estimates the equilibrium of delay and allocation cost. This algorithm is evaluated and compared against others, like the Random, Greedy, and Minimum allocation cost algorithms. Based on the results, it is evident that the new algorithm is more effective in maintaining the balance between latency and resource allocation cost. Through these two studies, the authors have made significant contributions to our understanding of how public cloud infrastructure can be used to host soft real-time applications like MMOG backends.

While previously mentioned studies have utilized the IaaS layer to power their backends, some have also used higher computing layers. Zahariev (2009) mentions the use of Google's App Engine (GAE), which allows the development of backends on Google's scalable, serverless infrastructure. By using GAE, developers can upload applications to a public cloud, in which it is deployed immediately as "*there are no servers to maintain and no administrators needed*". Backends hosted on the PaaS layer can enjoy several advantages, such as automatic scaling and load balancing, a variety of persistent storage options, a set of APIs for authentication, emails, analytics, and much more. These features integrate well and offer a stable, well-rounded environment for developing MMOG backends. In addition, they include support for a variety of *Integrated Development Environments* (IDEs), which allow local testing of GAE instances.

Moreover, products like GAE and Google's Firebase (Google 2021) offer access to a free tier of services that allows developers to host applications for free up to a certain resource quota. The free tier, coupled with locally-available tools for testing allows game developers to experiment with their code without worrying about the costs associated with hosting. Comparing the serverless and IaaS layers, Zahariev (2009) concludes that even though the "*resulting system will be more extensible*" in IaaS, it will "*take more time to build*", whereas development on PaaS is more constrained but offers a much more efficient development process.

The use of the public cloud and PaaS layer is further supported by Shabani et al. (2014), who discuss the use of various products such as GAE and Google's Datastore to create large-scale distributed systems. The authors mention that GAE can be used to "*to serve 'real-time dynamic' applications which are simultaneously accessed by many users*", which aligns with the objectives of an MMOG's backend. Complementing the survey conducted by Zahariev (2009), the authors mention that PaaS products like GAE can support multiple programming languages. An exploration of these tools reveals that GAE and Firebase can support a multitude of development environments, including Java, Python, Go, PHP, .NET, Ruby, and Node.js, which makes it possible for developers to create MMOG backends on their favorite technology stack. Furthermore, Donkervliet et al. (2020) supports that serverless Modifiable Virtual Environments (MVEs) can have independent services scheduled in parallel, managed by the cloud operator, leading to high elasticity. The addition or removal of these services is fine-grained, which leads to "*good elasticity properties*". These services are also isolated from each other, making it easier for game operators to consolidate them. This isolation further provides a possibility to create a *modular system*, in which it is easier to create different APIs for different aspects of MMOGs. This makes it easier to create games that a) separate game logic from the technology or infrastructure used, and b) feature support for player-created content such as mods, which is a major driving force behind popular games like Minecraft (Donkervliet et al. 2020).

Hybrid clouds

A hybrid cloud is a combination of private and public clouds. The purpose of hybrid clouds is to harness the advantages of both types of clouds while negating their disadvantages. Nae et al. (2009) show how hybrid clouds can help power online game servers by describing an infrastructure where smaller, less expensive, privately-owned data centers host MMOG backends, while additional public cloud servers are available, offering access to virtualized resources. This works by having game hosts pool resources from both private and public data centers to serve multiple games at the same time. This adds an additional virtualization overhead to the standard cloud approach, which is necessary to enable resources from both private and public clouds to be dynamically provisioned through an integrated set of policies. The authors propose a new resource provisioning model for their infrastructure, which they evaluate based on various metrics related to virtualization overhead. They measure the time taken to instantiate VMs and allocate resources and find that virtualization policies in hybrid clouds are very important to achieve good performance.

Furthermore, Negrão et al. (2016) describe significant contributions in the area of hybrid clouds. The authors use a hybrid cloud solution in combination with a system that breaks down high-level tasks into smaller sub-tasks that can be offloaded to public cloud resources. The system identifies overload events on the servers and breaks down larger tasks that would normally be executed on an overloaded server. These tasks are then offloaded into other machines to alleviate overloading and improve performance. These sub-tasks are categorized into two groups. Firstly, the important sub-tasks that have strong timing constraints or that require game data to be kept in-house are categorized as *core tasks*. These are kept within the private cloud. On the other hand, less sensitive tasks with more lenient time requirements are categorized as *background tasks*, and can therefore be offloaded to public cloud resources that are acquired temporarily to relieve the load from the private cloud servers. Background tasks are designed to be game-independent, which means that no game state computations are necessary. This approach is ideal for situations where there is temporary overload and for the execution of tasks in unreliable environments. The authors empirically evaluate this novel system and show that

it achieves "*modest*" frame rates for up to 1,500 clients. They argue that this approach "*gives application programmers more freedom*" while still allowing applications to benefit from the advantages of task partitioning.

2.8.2 Architecture

In terms of network architecture, developers and researchers have utilized a variety of approaches that can be categorized into three main groups.

Client-server

By far, the most frequently used architecture type in MMOG backends is the client-server model. This model provides a distributed structure that partitions workload in a system among resource providers, called *servers*, and resource consumers, called *clients*. In this architecture, machines use messages in a pre-defined language, also known as a *protocol*, to communicate. Typically, a client will need access to a resource, which it requests from a server through the network. The server is responsible for receiving the request, authenticating it, finding the necessary resource, and responding to the client. Client devices do not directly communicate with each other and rely on the server to relay messages about the state of an application.

The centralized nature of this architecture makes it useful for MMOG backends. Researchers have utilized this architecture to support large numbers of concurrent players without sacrificing efficiency or security (Assiotis & Tzanov 2005), and there are popular examples of existing games –like *Quake* and *Doom*– that utilize this architecture. However, challenges arise when dealing with a large amount of traffic generated by simultaneous gameplay from many players. The large amount of information communicated between players' clients and the server(s) requires a large bandwidth to support gameplay. Secondly, the large virtual worlds hosted by MMOGs require "*huge computational power*" (Assiotis & Tzanov 2005) to simulate, which means that at some point the state of the game must be divided among multiple server nodes. In their study, Assiotis & Tzanov (2005) separate large worlds into smaller, more manageable

regions that can be hosted on different computing nodes. By using this technique, the authors manage to spread out the computational requirements of the game among multiple machines. Despite its usefulness in that regard, this concept presents new challenges:

- Players are not always interested in receiving updates about certain areas of the map – especially if these areas are far away.
- When two players are near the border between two parts of the world, they still need to *see* and *interact* with each other – however this is not trivial when these are hosted on separate nodes.
- Regardless of the synchronization scheme, there is a possibility the game state will be invalid for events that occur near borders and affect players on both sides.

Firstly, the authors introduce a concept called the *Area of Interest* (AoI). They define this area as an area spanning outwards from a player's position for a certain distance, within which they are able to receive event updates occurring in the game world. Outside of this area, it is assumed that players would naturally not be interested to receive updates about events, as they may be too far away to perceive them. The AoI of each player or entity may depend on specific game mechanics. For instance, players carrying a sniper rifle will naturally have a larger AoI than those carrying a pistol, representing the real-life range of their equipment. Consequently, players can be subscribed only to a limited subset of the full game state, which drastically reduces bandwidth requirements.

Secondly, the authors identify four scenarios where players' actions near border areas must be handled:

- A player standing near the border of two regions hosted by different servers needs to be able to receive event updates within their AoI from both servers.
- A player may suddenly move to an area handled by a different server (this is usually known as *teleporting* in games).

- An event originating in one server may end up in a region covered by another server. A typical example of this is shooting a rocket that travels from one area to another before exploding.
- An event that occurs near the border may affect multiple regions, hosted on different servers. An example of this is a bomb exploding at a border, affecting players in adjacent regions.

To solve these problems, the authors discuss the potential of subscribing players to both servers/regions when they are in proximity of a border, and how different nodes may cooperate to share and update their states when a border event occurs. Based on the results of their evaluation, the authors have managed to improve the efficiency and performance of the client-server architecture using techniques to address potential problems in hosting MMOG backends with large virtual worlds.

Nae, Iosup & Prodan (2010) further explore how MMOGs can "*operate as client-server architectures*". They describe the game server as a component that simulates a world through computational and data operations and by receiving commands from the clients. Once these commands are executed, the server computes the *global state* of the game world. This represents the positions and interactions between entities in the world. When this process is completed, the server sends responses that contain the updated state back to the clients, which are responsible for presenting this information to the player by rendering graphics. It is strongly argued that to keep players engaged, a good game experience is very important Nae et al. (2011). In addition, game experience has a direct impact on a game's monetary success and the income of the game operators. Therefore, it is important that architectures can support large numbers of simultaneous players efficiently. To achieve a good experience when dealing with such demands, the authors present three *parallelization techniques* used with the client-server architecture:

- **Zoning**: This technique partitions the game world into areas that are "*handled independently by separate machines*". This technique is particularly useful in slow-paced games such as MMORPGs.

- **Replication**: A technique that parallelizes game sessions with large numbers of players gathering in certain hot spots. Each server computes the state of a number of *active entities* that are based on it, while it synchronizes the state of other *shadow entities* that are based on different machines. This technique is primarily used in fast-paced games such as FPS games.

- **Instancing**: "*Distributes the session load by starting multiple parallel instances of highly populated zones*" (Nae, Iosup & Prodan 2010). These zones are independent of each other.

Peer to peer

A lesser-used type of architecture in MMOG backends is the peer-to-peer (P2P) architecture. P2P partitions and fully distributes the workload among equipotent and equally privileged peers, making each peer a participant in the game's processing pipeline. This works by utilizing a portion of each peer's resources for use by other peers when required. Thus, in this architecture, a node can be considered both a client and a server simultaneously.

GauthierDickey et al. (2004) extensively discuss the use of P2P architecture to host a fully distributed MMOG backend, arguing that P2P can introduce several advantages for online games. Firstly, it reduces the delay for messages and eliminates localized congestion, as traffic is dispersed on many machines instead of a single server node. Secondly, it allows players to launch their games without investing in expensive hardware required to create powerful game servers. Thirdly, it enables games to overcome several bottlenecks of centralized computation, and finally, it is more resilient and available as it does not have a single point of failure. Kavalionak et al. (2015) have also studied P2P architectures in the context of MMOG backends, and identified further potential advantages. For instance, they argue that P2P systems are inherently scalable, as the number of available resources grows with the number of players joining the game. Such systems are also more robust, as the architecture can "*self-repair*" when a peer fails.

GauthierDickey et al. (2004) further discuss how different aspects of online games may work using P2P. For instance, data consistency must be guaranteed using mutual exclusion, as there may be copies of the same data on multiple peers. In terms of storage and data specifically, there are two types of approaches. In the *unstructured* P2P approach, clients can transfer data to each other directly, while in the *structured* approach, a distributed hash table must be maintained to convert resource names into network addresses. In the latter, special algorithms are required to maintain routing tables in each peer and ensure consistency. In terms of processing, the authors mention the use of distributed scheduling techniques to allow peers that need more processing power to leverage resources from other peers which have lower demands.

Due to the necessity of utilizing these techniques to ensure consistency and smoothness in performance, it can be inferred that P2P is relatively more complex to develop and operate compared to the client-server architecture. A more recent study, Mildner et al. (2017) focuses on the performance of the P2P architecture for MMOGs. The authors propose a P2P-based Networked Virtual Environment (NVE) for an MMOFPS game. They attempt to minimize the overhead for connection management by utilizing a *publish-subscribe* mechanism that avoids inconsistencies, instead of using sender-oriented message distribution. In addition, their new Geocast algorithm sends messages to users relative to their positions. By using an NVE system and a pre-existing game called PlanetΠ4 in a simulation environment, they obtain results that indicate that their approach offers a scalable and consistent overlay that limits the number of connections made to the network and therefore improves performance in crowding/flocking scenarios. Despite these improvements in terms of performance, the decentralized nature of the P2P architecture leaves it vulnerable to state manipulation (also known in games as *cheating*). This will be further discussed in section 2.8.6.

Hybrid architecture

Meanwhile, others have proposed novel approaches that utilize both client-server and P2P in hybrid architectures. For instance, one of the first studies that explore the use of P2P systems in conjunction with cloud computing by Kavalionak et al. (2015) proposes the use

of a hybrid architecture for an MMOG backend. In this approach, the architecture consists of two components: the *positional action manager*, which is responsible for managing the positions of entities, and the *state action manager*, which allows the storage of entity states without transferring them across nodes. The authors' main aim is to "*exploit and combine*" the advantages of both client-server and P2P architectures. However, this brings several challenges in itself. Firstly, the use of both architectures raises the complexity of development and makes it necessary to partition the virtual environment in some way. The authors use *spatial partitioning* to divide the world into regions that are distributed to peers, with the most 'resourceful' peer in a region becoming its manager.

Different approaches to partitioning have also been devised in other studies. For instance, Shaikh et al. (2006) use a central server in combination with a pool of peers. In this approach, the central server is responsible for hosting the MMOG and distributing the game state to other peers once full capacity is reached. Furthermore, using *functional partitioning*, important functions can be delegated to peers. Jardine & Zappala (2008) have successfully used functional partitioning by first categorizing the types of moves/actions within an MMOG into *positional moves*, in which a player changes their position within the game world, and *state-changing moves*, which have a direct effect on the game's state. Positional moves are composed of abstract data that relates to the position of the player and do not contain any player or entity-specific information, which allows them to be delegated to non-reliable peers. Conversely, state-changing moves contain entity-specific information and must be processed in a central server. Thus, functional partitioning allows the delegation of only a subset of the complete set of events to other nodes through the P2P approach. The system works by allowing a central server to appoint peers as regional servers that can be used to handle positional moves for a specific region of the world. The hosted MMOG will still be able to function and remain consistent even if an unreliable peer leaves the network.

A known problem of the P2P architecture is the state manipulation that can occur due to its decentralized nature. Matsumoto & Okabe (2017) study the features of MMOGs and investigate the types of cheats seen in games, their frequency, and detectability. To combat cheating in these architectures, the authors propose a collusion-resilient hybrid P2P framework that utilizes

various techniques such as *data scrambling*. After evaluating their framework and comparing it with other similar approaches, the authors conclude that it offers more effective protection against cheating and especially peer collusion, even though it did not fully protect against other types of cheating.

Meanwhile, Zhang et al. (2017) further identify the challenges of hosting Virtual Reality MMOGs (VR-MMOGs): large scales, stringent latency, and high bandwidth. Their study focuses on improving the performance of hybrid game architectures to achieve a better, more efficient distribution of workload. They assign *local view updates* to be handled by *edge clouds* in order to achieve faster response times, whereas high-bandwidth, *global state updates* are assigned to centralized clouds. The authors use a service placement algorithm that dynamically places services on edge clouds while players move across the game world. The evaluation of this approach through simulations points towards a "*viable solution for supporting VR-MMOGS*".

The use of edge computing technology with the P2P architecture has also been explored by Plumb et al. (2018*b*). The authors devise AvatarFog, a system that enables the formation of hybrid P2P clusters. The clusters can be formed using game design principles to decide the network topology instead of assigning a physical structure like the client-server architecture. The focus of this research is the improvement of latency for actions that take place between the players by paying more attention to the interactions that occur between them within the game world rather than the physical connections between their clients. This approach places players into groups based on their gameplay behavior and interactions rather than their physical positions in the world. Through a custom simulation, the authors evaluate the performance of their framework and conclude that AvatarFog "*improves the latency and server resources of the traditional server and client model*".

2.8.3 Performance

Perhaps one of the most critical aspects determining the success of an MMOG is its performance (Nae et al. 2011, Dhib, Boussetta, Zangar & Tabbane 2016). Researchers have used many techniques to *improve* the performance of MMOG backends, as well as provide meaningful ways

to *measure* it. In popular culture, the performance of resource-intensive, fast-paced games is often measured in *Frames Per Second* (FPS, not to be confused with First Person Shooter games), which is a metric used to measure the *refresh rate* in a game's loop (Janzen & Teather 2014). The refresh rate determines how many times the game world can be rendered in each second, with a higher refresh rate corresponding to a better QoE. However, this only provides very minor insights into the performance of the backend, as it is mostly linked to the client-side performance that mainly involves graphics rendering. For the backend, other metrics can be used to measure performance, with *latency* being by far the most important (GauthierDickey et al. 2004, Jardine & Zappala 2008, Burger et al. 2016, Dhib, Boussetta, Zangar & Tabbane 2016, Dhib, Zangar, Tabbane & Boussetta 2016). The list below enumerates some of the most commonly-used metrics used to measure the performance of MMOG backends.

- **Latency** – Burger et al. (2016), Dhib, Boussetta, Zangar & Tabbane (2016), Dhib, Zangar, Tabbane & Boussetta (2016), GauthierDickey et al. (2004), Jardine & Zappala (2008), Meiländer & Gorlatch (2018), Najaran & Krasic (2010), Plumb et al. (2018*a*).
- **Bandwidth** – Jardine & Zappala (2008).
- **Network distance between peers/servers** – Dhib, Boussetta, Zangar & Tabbane (2016), Plumb & Stutsman (2018).
- **Number of players** – Lu et al. (2006).
- **Messages per second** – Lu et al. (2006).
- **Moves per second** – Jardine & Zappala (2008).
- **Number of connections** – Plumb et al. (2018*a*).

The latency metric is not only mentioned frequently in relevant works but is also considered the main benchmark of performance. Game genres are often associated with certain latency expectations. Based on the related works, the following genres have different latency requirements:

- **First Person Shooter (FPS)**: 50 ms - 250 ms (Nae, Iosup & Prodan 2010, Shea et al. 2013, GauthierDickey et al. 2004).
- **Real-Time Strategy (RTS)**: 500ms - 1000 ms (Shea et al. 2013, GauthierDickey et al. 2004).
- **Role-Playing Games (RPG)**: 1000ms - 2000ms (Nae, Iosup & Prodan 2010, Shea et al. 2013).

Perhaps the reason why latency is regarded as so important is because it has a direct effect not only on performance but also on the QoE. For cloud-based MMOGs, "*ensuring an acceptable Quality of Experience (QoE) for all players is a fundamental requirement*" (Dhib, Boussetta, Zangar & Tabbane 2016). To measure the QoE in MMOGs, Dhib, Boussetta, Zangar & Tabbane (2016) propose a mathematical model and identify the *global response delay* (latency) as the most "*notable*" metric. Furthermore, they support that this metric is significantly affected by other parameters such as CPU and memory capacity, as well as the network distance between the players and the servers. They propose the use of their model to express the QoE as a function of the network-based and processing-based delays. Through their dynamic VM allocation strategy, the authors attempt to minimize the cost per player and ensure that the QoE remains above a "*minimal threshold*". To evaluate their model, the authors measure the performance of a cloud-based MMOG in terms of latency using simulations. In addition, they measure how much the QoE is degraded as a function of the number of allocated VMs and the number of players. Results from these experiments show that the approach achieved a "*high player satisfaction*", and maintained 99% of the QoE.

At the same time, other studies attempt to improve the performance of cloud-based MMOGs using various approaches. For instance, Lin & Shen (2015*b*) proposes a lightweight system called CloudFog that adds a layer to cloud technology through the use of "*supernode*" machines that are located between the players and the cloud. In this system, the game state is computed on the cloud, while the supernodes are used to carry out other intensive tasks such as video rendering and data streaming. In their work, the authors have also identified various challenges

that hinder the success of games, including *latency*, *network connection quality*, *user coverage*, and *bandwidth cost*. Through simulations, they evaluate CloudFog and compare it against other similar systems based on latency, playback continuity, and user coverage. From their results, they conclude that CloudFog's supernodes approach reduces latency, bandwidth consumption, and cost while having an overall positive impact on QoE and user coverage.

The performance of an MMOG can also be directly affected by the behavior of players in the game. Gascon-Samson et al. (2015) identify *flocking*, the gathering of many players at specific locations in the game world, as a challenge to achieving good performance. While techniques such as zoning are useful for dividing game areas among computing nodes, they have an important limitation. Flocking can induce high-performance requirements on single computing nodes that handle a specific game area, while other nodes remain underutilized. This may significantly affect the QoE in one game area, while others remain unaffected. Hence, Gascon-Samson et al. (2015) propose the use of DynFilter, a message processing middleware that filters out state update messages from entities that are located away from a certain position, to avoid unnecessary updates and therefore reduce bandwidth requirements. Based on the publish-subscribe update pattern, this system can maintain bandwidth use within specified quotas and thus deliver a satisfactory QoE.

Arguably, the performance of MMOGs can also be impacted by inconsistencies and errors. Yusen et al. (2016) argue that MultiServer Distributed Virtual Environments (MSDVEs) can suffer from *saturation*, which ultimately leads to high bandwidth and resource demands. To solve this problem, they devise a new metric that uses time-space inconsistency to measure unfairness in an MSDVE. In addition, they propose a fairness-aware update scheme that allows updates to be issued to different clients simultaneously, with the aim of reducing inconsistencies. On top of this, they come up with a new algorithm called FairLMH, which allows the minimization of inconsistencies in MSDVEs. Through simulations, the authors have proven that their approach leads to better fairness compared to other similar algorithms, in multiple scenarios.

Furthermore, Assiotis & Tzanov (2005) state that systems "*should recover the entire state of*

the world it represents as it was prior to the crash very quickly and as transparently as possible". By having to recover their state from errors frequently, MMOGs may find their performance reduced, and the QoE degraded. This suggests that performance can also be impacted by a) the frequency of errors, b) the system's ability to recover to a valid state, and c) the time taken for this recovery to take place. To solve these problems, the authors suggest the use of mirroring and replication to prevent the players from being "*locked out*" of the game.

More generally, Baker et al. (2011) propose the use of a *network simulator* to detect bugs in cloud-based systems. The simulator can be used to explore all the possible orderings and delays of communication between the simulated nodes, in order to detect bugs and reproduce the circumstances that lead to a bug. The simulator works by using a seed, which means that it is capable of producing the same behavior for a system, and thus allows an examination of the conditions that lead to problematic circumstances. While an exhaustive search of all the possible states is impossible, especially for large-scale systems, the authors claim that the simulator can explore "*more than is practical by other means*".

Through various approaches, authors have attempted to reduce or divide the number of resources needed to process the states of MMOGs, and thus improve the QoE. At the architectural level, Jardine & Zappala (2008) have utilized a hybrid client-server/P2P network to enable an MMOG backend, while also using a client-server architecture for comparison. They run a series of experiments that consist of fifty automated players (bots), programmed to move toward game objectives as quickly as possible. Through their simulations, the authors compare the performance of their hybrid architecture against the traditional client-server architecture, and claim that their approach can "*save considerable bandwidth for the central server*". In addition, "*latency can be kept low*" as long as there are enough peers capable of acting as regional servers. Similarly, Negrão et al. (2016) have used a hybrid cloud solution to enable an MMOG backend, and compare the performance of their system when all servers are "*state partitioned*" against a version that uses mixed task servers and state partitioned servers. Through simulations with varying numbers of bots, they find that the difference in performance in their approach leads to higher frame rates and lower bandwidth consumption. Along with others, El Rhalibi & Al-Jumeily (2017) utilize a hybrid architecture in combination with a dynamic AoI

management solution that aims to minimize delay and network traffic. The authors simulate a game environment with varying numbers of peers, with scenarios utilizing both client-server and hybrid architectures. The results from these simulations show that AoI management techniques can produce lower latency and network traffic, especially when used in conjunction with hybrid architectures, compared to using client-server architectures without AoI management.

2.8.4 Scalability

Another fundamental characteristic of the systems powering MMOG backends is the ability to scale up or down to accommodate a fluctuating number of players or expanding/contracting game states. The MMOG acronym itself – Massively Multiplayer – suggests the need for scalability. MMOG backends should not only be scalable but preferably automatically scalable – or elastic – so that game developers can focus on game semantics rather than manually scaling infrastructure. These types of applications are unlike other distributed applications that tend to be "*embarrassingly parallel [and] optimized for throughput*" Blackman & Waldo (2009). Rather, MMOG backends are typically described by their ability to scale to incorporate massive game worlds and service thousands, or hundreds of thousands of players (Blackman & Waldo 2009). The effects of a system's ability to scale reach beyond the game's mechanics and features, as they can determine the revenue generated by an MMOG, and ultimately, the game operator's business strategy. This section discusses the related works in terms of scalability from various vantage points, highlights the importance of scalability, and how it can lead to the creation of more resource-efficient and profitable games.

Consistency

In the context of MMOG backends, consistency 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*" Lu et al. (2006). When a system becomes scalable, consistency arises as one of the major challenges. When the number of players –and therefore actions– in a system increases, the

number of servers to support the game also increases to cope with the demand. Naturally, it becomes increasingly difficult to avoid inconsistencies and poor performance without managing these issues directly. Lu et al. (2006) propose that the solution to the problem of consistency is the use of *localized gameplay*, in which game areas can be broken down into smaller, more manageable parts. Manageable consistency through localized gameplay is divided into two groups:

- **Geographic localization**: In this approach, the world is divided into regions at initialization. For instance, a room inside a building can be considered a separate geographic region that can be accessed when a player enters the room. This concept is similar to *zoning* mentioned by Nae et al. (2011).
- **Behavioral localization**: The division of the world into further sub-divisions based on the interaction patterns of players. For example, a difference in the size of the AoI, discussed by Assiotis & Tzanov (2005) and Nae et al. (2011), may alter a player's ability to influence the state of the game.

In geographic localization, game worlds are reduced into smaller parts using three rules: a) players cannot interact across duplicated worlds, b) players cannot interact across different regions, and c) players should be able to interact *intricately* with nearby players or players that they specifically target. Based on the varying levels of consistency described by these rules, the types of interactions possible can also be broken down into two different groups. Firstly, *view* interactions are interactions where the player can simply observe other players and their actions and thus require *weaker* consistency. On the other hand, *intricate* interactions take place when a player directly interacts with another player, which typically requires *stronger* consistency. In these intricate interactions, it is important to utilize event *synchronization* and *ordering* systems to maintain consistency.

Unfortunately, maintaining such strong consistency is detrimental to the performance of MMOGs, because it means that events cannot be executed in parallel. This is a more general problem in the software that is also noticed in other types of applications. A significant contribution

in this area is presented by Chuang et al. (2013). The authors introduce EventWave, which is an event-driven programming model that "*allows developers to design elastic programs with inelastic semantics*" Chuang et al. (2013). EventWave achieves this by first allowing logical nodes to execute multiple events in parallel, and secondly by allowing the distribution of a single logical node to multiple physical nodes, while at the same time guaranteeing atomic events. In a distributed system using EventWave events can be executed in parallel provided that they do not access the same state. This can be leveraged to improve performance by first identifying the events that are state-independent of each other. This is complemented by a technique known as *context mapping*, which maps event contexts to computing nodes. Ultimately, EventWave enables developers to reason about their program's execution without considering scalability, and thus focus on program logic rather than scaling.

Load balancing

Load balancing is described as the ability to "*efficiently distribute an application's processing requirements across a number of servers*" (Lu et al. 2006). Based on Lu et al. (2006), load balancing strategies can be categorized into two groups. In *player-based* load balancing, the players are directed to different servers as they join a game, whereas in *interaction-based* load balancing, servers manage the allocation of resources based on the players' interaction patterns. The former is advantageous when servers have to be added or removed without affecting game-play. When players are allocated in duplicated worlds, it is necessary to balance the load in order to support interactions between them. The messages exchanged between these servers may take a significant portion of the available bandwidth, thus making the use of AoI and other concepts very important. Conversely, interaction-based load balancing is ideal in scenarios where flocking/crowding/hotspot behavior –i.e. a high concentration of players in a specific area– is expected. To avoid the system from being completely overwhelmed at a specific node, this strategy can be used to associate player actions and therefore host them on a server that handles a specific interaction pattern.

Meiländer & Gorlatch (2018) investigate load balancing from the perspective of performance. They propose a generic scalability model for Real-Time Online Applications (ROIAs), which "*monitors the application's performance at runtime and predicts the benefit-cost ratio of load-balancing decisions*". The system weighs the benefits of different load-balancing actions against their perceived overheads, mainly in terms of time and resources, and deduces a ratio that can then be used to recommend whether or not to distribute workload, and how often this should be done. In this approach, the authors use a computation metric in terms of CPU usage and a communication metric in terms of network usage to evaluate the quality of their model. Using a multiplayer shooter game simulation, they prove their model's ability to offer a higher efficiency of load-balancing actions in clouds.

Extending our understanding of load balancing in the context of MMOG backends, Farlow & Trahan (2018) describe the problem as NP-complete, and state that load balancing is "*subject to constraints such as player satisfaction and maximum server computational capacity*". The authors develop heuristics that can monitor load balancing in a system, and which can be utilized to "*bring an unbalanced system back into balance*". Due to the large overhead of load-balancing operations, the authors use breakpoints, during which these operations can take place. They define the *Load Balancing Factor* (LBF) which helps determine the difference between the highest and lowest loaded servers. Using LBF, the authors can determine if a load-balancing action needs to be executed during a breakpoint. If this is the case, their system can add, shed, or rejoin zones to balance the load. These contributions are bundled in a system called BreakpointLB, which is evaluated through experimental simulations. The results of these simulations show that the system is capable of bringing unbalanced systems into load balance while performing more efficiently when compared to other approaches.

Load prediction and resource provision

Predicting resource usage and deciding when to allocate resources is another problem faced by game developers. For typical software applications, resources are made available to a system by utilizing dedicated infrastructures which include hundreds of servers (Nae, Iosup & Prodan

2010). However, this method is not sufficient to create efficient MMOG backends because resources are allocated statically, which leads to over-provisioning or under-provisioning. In turn, this can lead to financial inefficiencies, making it difficult to join, keep up, or innovate in an already competitive market. It is therefore no surprise that various studies have focused on the load prediction and resource provisioning aspect of MMOG backends.

Nae, Iosup & Prodan (2010) have proposed a solution for a "*dynamic [resource] provisioning method in which the amount of resources is first predicted and then obtained dynamically*". This approach takes advantage of analytical load models for CPU, memory, and network resources and takes into account the number of players and the types of interactions occurring between them. In addition, the world is divided into smaller areas, which improves the accuracy of these real-time load prediction models. The experiments conducted in this study show that the proposed approach can "*reduce MMOG operation costs*". The authors also evaluate a neural network predictor, finding that it offers "*the best resource provisioning*" results out of every strategy that they evaluate. Moreover, using load prediction the latency of the system can be reduced, and the authors argue that such systems can be virtualized, and therefore be used to service multiple MMOG backends simultaneously – highlighting the importance of using dynamic resource provisioning over static infrastructures.

Another innovation is the Provisioning Manager (PM), initially discussed in section 2.8.1. Shaikh et al. (2006) use the PM within a prototype implementation of a platform that can be used to host online games. In this approach, the PM plays a critical role. Firstly, it enables the system to measure resource requirements by collecting performance and availability metrics from different devices in the infrastructure. Using these measurements it can decide if resources must be allocated or de-allocated, and then responds to these changes by adding or removing servers from the system. In a similar way to the previous study, this is achieved mainly by measuring CPU and memory utilization, as well as bandwidth consumption. The PM takes this a step further, allowing the system to provide resources for auxiliary services such as game content distribution or service deployment. In conjunction with other sources (Nae et al. 2011, Negrão et al. 2016, Nae, Iosup & Prodan 2010), Shaikh et al. (2006) also identify the advantages of cloud computing and how these can be used to provide on-demand

resources to MMOG backends.

In another study, Ghobaei-Arani et al. (2019) address the problem of resource provisioning by proposing a new resource provisioning framework that works on cloud infrastructure. Similar to other approaches, they use a load prediction service that anticipates the distribution of entities within a game world using trace data provided by the ANFIS prediction model. This approach splits an MMOG into various tiers. The *gateway tier* is responsible for functioning as a bridge between the client and the game layer. The *cellular layer* is responsible for processing commands from the players, and finally, the *database tier* is responsible for storing data. The authors utilize a fuzzy decision tree algorithm to estimate the number of resources that should be allocated to each one of these tiers. To evaluate their approach, the authors use RunEscape to generate a real workload, as well as simulations for a synthetic workload. From these experiments, the authors discover that their approach outperforms others in terms of accuracy and performance.

2.8.5 Persistence

Like most online software applications, MMOG backends need to persist information after it is processed for various functions, including online gameplay, or general data storage. To enable persistence in online games, a large variety of storage systems are being used. This section discusses the different approaches used to enable persistence in MMOG backends, the tools used, and various important concepts relevant to data persistence.

The peculiarities of developing MMOG backends are also observable in the data persistence layer. Compared to other types of software, MMOG backend developers must place more attention on achieving lower latency, rather than higher throughput (Blackman & Waldo 2009). In addition, MMOGs typically require a high ratio of updates to writes or reads, unlike most business applications. When dealing with data corruption or loss of information, studies have shown that players are more willing to tolerate the loss of data as long as the recovered game state remains consistent, which is in contrast to other types of applications. Blackman & Waldo (2009) discuss these issues within Project Darkstar, "*an infrastructure for building online game*

worlds". The authors propose the use of *write caching*, an approach that locally caches data on each node if the data is only relevant to that node. When other nodes require access to this data, the node flushes it to a central server where it can be accessed. Using this approach can lead to lower latency because most of the data is utilized by the same node. This approach also removes the need for redundancy or backups, as the nodes do not store any global data, and it makes it easier to remove nodes from the system. Thus, the system can remain scalable while maintaining consistency.

Researchers and developers have used a variety of database types to enable persistence in MMOG backends, including SQL databases, NoSQL datastores, and caching systems, on and off the cloud. For example, Google's Spanner is a "*highly available global SQL database*" that manages data replication and transactions at large scales, making it ideal for use in MMOG backends. Brewer's theorem (Brewer 2017) (also known as the CAP Theorem) states that systems can only fully attain two of the following three properties: *Consistency*, *Availability*, and *Partition tolerance*. This means that databases distributed across many nodes –including those used in MMOG backends– cannot be both fully consistent and available at the same time. According to Brewer (2017), either one of these must be sacrificed to some extent to achieve the necessary scalability: "*Relaxing consistency [allows] the system to remain highly available whereas making consistency a priority means that the system will not be [fully] available*" (Brewer 2017). Vogels (2009) has suggested a concept known as *eventual consistency* to work around the CAP theorem. Eventual consistency, which is a form of *weak consistency*, guarantees that all accesses to an object will return the last updated value, given that no updates are made to the object. Depending on the system's load and latency requirements, this presents a specific *inconsistency window* during which consistency failures may arise. This inconsistency has to be tolerated, because it results to a performance improvement, especially under highly concurrent conditions, and enables the partitioning of data that would otherwise be impossible.

To use a persistence system on the cloud, the approach used must possess certain features that enable it to effectively utilize *cloud economics*: scalability, elasticity, fault tolerance, self-manageability, and ability to run on commodity hardware. Agrawal et al. (2011) argue that

Relational Database systems (RDBMSs) are not optimized for use in the cloud as they were designed for enterprise infrastructure. In addition, the large costs associated with deploying these systems for large-scale applications make them a less attractive option. Instead, the authors recommend the use of a newer generation of distributed key-value data stores. This type of system is known as a NoSQL datastore and has been successfully adopted mostly because it is compatible with cloud-based systems as it can scale easily. In terms of functionality, this approach lacks various features compared to the RDBMS approach, such as a strict schema, support for complex queries and transactions, and more. On the other hand, NoSQL datastores like Bigtable (Chang et al. 2008) and Cassandra (Diao et al. 2015) are easily scalable but have fewer features, limited APIs, and loose consistency, which complicates development.

Baker et al. (2011) proposes the use of the Megastore, which "*[blends] the scalability of NoSQL datastores with the convenience and functionality of a traditional RDBMS*". Using Megastore can provide low latency through its fully serializable ACID semantics (Atomicity, Consistency, Isolation, Durability). In addition, it uses synchronous replication to achieve high availability and strong consistency at the same time. It combines the ability of RDBMS to define schemas for the data model, while also allowing items to be created as entities that contain sets of properties in key-value pairs. One of the latest incarnations of Megastore is Google's Cloud Datastore (Shabani et al. 2014), which is a highly scalable, highly available distributed data storage system that can be utilized within the Google Cloud Platform. Applications developed using Google's App Engine can utilize the Cloud Datastore to create web applications or services which enjoy the scalability of NoSQL systems. In addition, Google's Query Language (GQL) can be used to write and execute queries on the data, similar to those executed in RDBMS. While GQL has a very similar syntax to the broadly-used SQL, it has some limitations such as the lack of complex join queries.

Recent research by Diao et al. (2015) studies how strong consistency can be provided for cases where systems have to be both highly available and scalable. The authors implement a lightweight mechanism that detects failures in a system and reacts where needed. Firstly, MMOG data is classified into four categories: *account data*, *game data*, *state data*, and *log data*. Subsequently, the authors use an approach that processes the modifications of state data

in real-time using an "*in-memory database*" (i.e. cache). These changes are synchronously propagated to other players, and the data is backed up to a disk database *periodically* to allow its recovery in case of failure. According Diao et al. (2015), popular MMOGs like World of Warcraft and Second Life use RDBMSes, like MySQL and Microsoft SQL server. They argue that this approach is outdated and that such systems can be replaced with more suitable non-relational systems like Cassandra, which can provide support for bulk writes and updates, and handle read operations more rarely, which is the predominant scenario in MMOGs. However, Cassandra's weak support for strong consistency prompts the authors to explore a solution using eventual consistency. In further research (Diao 2017), the authors also investigate the benefits of cloud data management solutions, while identifying any potential shortcomings in the context of MMOGs. After an analysis of the requirements, the authors categorize MMOGs into groups and propose the use of multiple data management systems simultaneously to provide a diverse set of features. For instance, data that requires strong consistency and security, like account data, can be managed by an RDBMS, while data requiring scalability and performance (i.e. game/state data and logs) can be stored using a cloud-based persistence system.

2.8.6 Security

Security is an often overlooked aspect when it comes to developing games, as it may not be as critical to their development. However, modern games and the platforms that support them often include additional features and related data, rather than just game worlds and simulated entities. These can include features like micro-transactions, content management, infrastructure management, and more, which sometimes require the use and storage of sensitive data, such as financial information, personal data, and so on. This means that MMOG developers are required to take a serious stance regarding security to avoid legal problems or financial losses. This section explores the importance of security, potential threats, and potential mitigating countermeasures.

Shaikh et al. (2006) identify a security issue in the context of their on-demand platform for MMOGs. The authors argue that while it may be desirable for games with low resource

requirements to be hosted on the same server, this creates a security problem because games must not be able to corrupt each other's state. Thus, "*sufficient protection*" is required to ensure that games cannot access each other's state.

Secondly, the same authors argue that while the P2P architecture can allow MMOGs to enjoy higher performance and lower bandwidth use, its decentralized nature can create many security problems. Specifically for content distribution on P2P networks, they argue that "*Players must allow [...] untrusted machines to connect to their own machines*", which can expose them to malicious actions. More severe threats stemming from the use of P2P architectures are also discussed by Kavalionak et al. (2015), which state that in P2P architectures, the lack of central authority "*hinders security and anti-cheating enforcement*". When clients have "*heterogeneous constraints on computational, storage and communication capabilities*", they can be vulnerable to exploitation in various ways. GauthierDickey et al. (2004) also identify cheating as a severe problem that "*plagues modern games*", and agree that especially in P2P architectures, this problem can arise from data manipulation. The authors classify the types of cheats employed by malicious players as:

- **Fixed-delay cheat**, where a fixed amount of delay is purposefully added into each packet.
- **Timestamp cheat**, where timestamps are changed to alter when events occur.
- **Suppressed update cheat**, where updates are purposefully not sent to other players.
- **Inconsistency cheat**, where different updates are sent to different players.

The authors discuss solutions to these types of cheats, including *Lockstep*, which can guard against cheats by dividing a game's time into rounds. During each round, player clients send a cryptographic hash of their move to other players, and this can ensure that actions cannot be modified. While it protects against cheating, this approach presents a drawback: unacceptably high latency due to the calculation overheads of hashing – ultimately beating the purpose of the means. Alternative methods like *Asynchronous synchronization* mentioned by Baughman & Levine (2001), and the *Sliding pipeline control* (Jamin et al. 2003) are also problematic due to high overheads and incomplete protection against all types of cheats.

Perhaps the solution to guarding against malicious state manipulation can be found by utilizing hybrid architectures like the one presented by Jardine & Zappala (2008). In this approach, critical processing events occur on a central server, while non-critical events are offloaded to other nodes in a P2P network. The authors claim that the ability to cheat "*is significantly limited*" in this architecture because the state is controlled exclusively by the central server. A possible attack in such a scenario is when a regional (P2P-based) server intentionally drops or delays state updates (suppressed update cheat). The solution discussed by the authors copes with this by having clients monitor regional server updates for latency and packet loss and then reporting any issues to the central server. If three consecutive updates are missed, the regional server is considered insecure and may be replaced or removed. This provides "*additional protection against poor performance or failure*" (Jardine & Zappala 2008).

Another possible security issue would be when "*a player acting as a regional server [joins] its own region*". This must be forbidden, as the player hosting that region may be able to access its entire state and view how other players behave before their actions are executed. In this case, the central server will replace nodes that attempt to host their own region. Regional servers may also attempt to "*collude with other players in [their] region*". Jardine & Zappala (2008) tackle this problem by implementing an auditing mechanism that checks each state-changing move for legitimacy by using logs to verify that a player had enough time to execute the move. Players may also "*receive an unfair advantage by joining many regions at the same time*", which is possible if their computer is powerful enough. In that case, the central server, which controls all regional server assignments, will check if the player moving between two regions can be removed from one region and added to another, thus limiting the number of regions a player can actively host.

2.8.7 Other approaches

In the previous section, the aspects of MMOG backend development were discussed, and various approaches were presented for each of these aspects. However, some other, alternative approaches exist which are not related to these aspects and could be described as *separate*

trends. While there are many similarities, a few differences between these approaches are their architecture, as well as their monetary model.

It is well established that the deployment and maintenance of large data centers are cost-prohibitive. This makes it very hard for smaller game studios to employ such technologies. In addition, the network distance between these large data centers is often large, which in the context of MMOGs is problematic as it causes high latency. Lin & Shen (2015*a*) have proposed a lightweight system called CloudFog, which uses powerful super-nodes acting as intermediary nodes between large data centers and client devices. In this approach, performance-intensive tasks like game state computations, geometry calculations, and so on, are executed on the cloud. Once these tasks are completed, they are sent to the intermediary super-nodes, which are responsible for rendering the game world as it would be displayed on the client device. Finally, the super-nodes stream the rendered view to the players as video. This concept is broadly known as *Cloud Gaming*, a type of online gaming that offers Gaming as a Service (GaaS). A major advantage of Cloud Gaming is that users can enjoy playing resource-intensive games without spending large amounts of money to acquire expensive hardware. Furthermore, latency is reduced as the intermediary nodes are naturally closer to the client devices, compared to the larger distances of cloud-based servers. This also helps to reduce bandwidth costs, as the streaming of video is done from the super-nodes rather than on the cloud. However, a disadvantage of this approach is the requirement for a stable, high-speed Internet connection, which cannot be guaranteed for all contexts – i.e. for mobile games.

The use of edge technology is not only associated with Cloud Gaming. Others like Burger et al. (2016) have also considered the pitfalls of large network distances and the broad geographical distribution of players on performance. To minimize latency, Burger et al. (2016) argue that game servers must move closer to the players, and thus towards the edge of the network. The authors analyze game histories using statistics from the Steam platform and develop a model with which they can predict player locations and match durations. They use this model to evaluate the migration of Dota 2 (a popular MMOG) matches toward the edge of the network through an event-based simulation framework. In this study, the authors focus on how server placement impacts the QoE in MMOGs. The authors discovered that deploying edge servers

reduces the distance from the servers to the player by half. This means that performance is improved, because latency becomes lower, and the requirements imposed on the server are reduced. The authors deduce that higher numbers of edge servers with small capacities may be more beneficial compared to having a few, powerful dedicated servers, despite the higher operational overload.

Others have explored the use of P2P networks in combination with edge computing. A study by Plumb & Stutsman (2018) argues that "*Google's Edge Network changes everything we have concluded about peer-to-peer networks over the past decade*". The use of Edge Network enables the inclusion of trusted peers within untrusted clusters, which in turn allows developers to utilize P2P algorithms in a secure environment. In their study, the authors explore the possible uses of this approach in the context of MMOGs backends. By first gathering ping data and population maps from locations in the United States, the authors run a simulation to compare existing solutions like the *Traditional topology* and *Edge topology* with their own *Optimized edge network*. The analysis of their results presents improvements in performance compared to other approaches as latency is reduced, in addition to maintaining security within their P2P network.

Google's Stadia (Google 2019) is also an important milestone in the development of cloud gaming services. Stadia takes an extreme approach that also employs thin-client computing, and is built to stream games at high resolutions and frame rates. This service requires a stable, high-speed Internet connection, but no expensive gaming hardware on the client-side, such as a powerful Graphics Processing Unit (GPU) or processor. In Stadia, game state computations and graphics rendering occur on cloud-based hardware, and the view of the game is streamed to the player using "*YouTube-like functionality*" Google (2019). This approach offers "*tremendous scale*", but only provides a very limited set of development tools. While Stadia is built to support multiplayer games, it is not designed to enable MMOGs, even though this architecture appears to be conducive to these types of games. Alternatively, other approaches like Apple's Arcade (Apple 2019) are more focused on saving the states of games on the cloud so that players can seamlessly switch devices, but fall short of providing online gameplay.

2.9 Analysis of the related works

The involvement of multiple aspects in the creation of MMOG backends complicates their development, especially given the fact that each of these aspects can be regarded as an area of computer science on its own. Due to this diversity, it is expected that most game developers will lack the knowledge or skills necessary to make analyses and decisions regarding these aspects (Boroń et al. 2020). Moreover, cloud computing helps bring down operational costs for such large distributed real-time systems, but the functionality of MMOG backends still heavily relies on developing custom-tailored backends using specific technologies. In this section important related works, which are considered milestones due to major contributions in their respective areas, are presented and categorized based on each aspect. The purpose of this process is a) to enhance the understanding of which approaches are most popular and why, and b) to enable an analysis of the past, present, and future trends for each aspect, and for all aspects collectively.

Table 2.2 summarizes the aspects initially introduced in section 2.6 and lists the categories used for classifying the approaches used in each of these aspects. In table 2.3, the relevant studies are listed in chronological order, showing how each of them handles the identified aspects. For analytical purposes, the table includes related studies (i.e. of the same or similar authors, or using an identical approach) in unified entries.

2.9.1 Infrastructure

A selected total of 50 highly-relevant papers were organized into 42 entries. The results, shown in figure 2.1, show that the majority of the entries (53%) employ, or discuss the employment of private cloud infrastructure. This is followed by the dedicated approach at 23%. Public cloud approaches like IaaS (15%) and Serverless (6%) are at roughly the same level as the dedicated approach in terms of use, having a combined frequency of 21%. The least popular infrastructure approach is hybrid clouds with only 3% of entries. It is worth observing that public cloud IaaS solutions are significantly more popular than serverless options. From these results, it is also evident that the use of either private, public, or hybrid clouds now eclipses

Aspect	Categories
Infrastructure	D = Dedicated PrC = Private Cloud HC = Hybrid Cloud PuC = Public Cloud – IaaS or Serverless (SL) U = Unknown
Architecture	CS = Client-Server P2P = Peer-to-peer H = Hybrid architecture U = Unknown
Performance	Evaluation approach: S = Simulation M = Modelling U = Unknown Tools: P = Pre-existing MMOGs O = Other types of applications
Scalability	NS = Not scalable MS = Manually scalable E = Elastic (automatically scalable) U = Unknown
Persistence	R = Relational Databases N = NoSQL U = Unknown
Security	L = Loose security (not controlled by architecture) T = Tight security (controlled by architecture) U = Unknown

Table 2.2: Aspects and categories used to classify approaches for developing Massively Multiplayer Online Game (MMOG) backends.

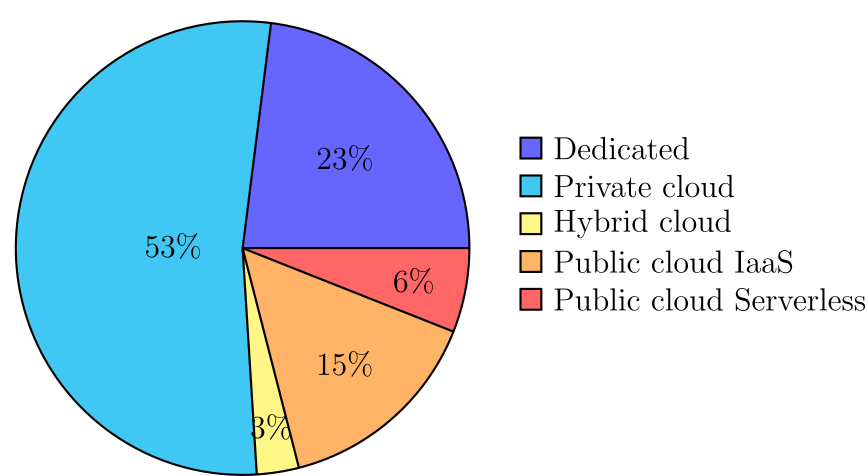


Figure 2.1: Infrastructure approaches used as a percentage of the total papers mentioning this aspect.

Table 2.3: Comparing the studied approaches using the identified criteria (Infrastructure, Architecture, Scalability, Persistence, Performance and Security).

Approach	Infrastructure	Architecture	Scalability	Persistence	Performance	Security
GauthierDickey et al. (2004)	D	P2P	NS	U	U	L
Assiotis & Tzanov (2005)	D	CS	NS	U	M & P	L
Shaikh et al. (2006), Shaikh et al. (2004)	PrC	U	E	R	M, S & P	U
Lu et al. (2006)	D	CS	MS	U	S & M	U
Jardine & Zappala (2008)	U	H	MS	U	S & P	T
Chu (2008)	D	CS	MS	R	U	T
Kienzle et al. (2009), Zhang et al. (2008)	U	CS	MS	R	O	U
Blackman & Waldo (2009)	D	U	MS	R	U	U
Nae et al. (2011), Nae, Iosup & Prodan (2010)	PrC	CS	E	U	S, M & P	U
Chang et al. (2008), Baker et al. (2011)	PrC	U	E	N	S & O	U
Weng & Wang (2012)	PrC	CS	E	U	M & P	U
Chuang et al. (2013)	PrC	U	E	N	S & P, O	U
Carter et al. (2013)	U	H	MS	U	S	U
Shen et al. (2013), Iosup et al. (2014)	PuC-IaaS	CS	E	U	U	U
Shabani et al. (2014)	PuC-SL	U	E	N	U	U
Deng et al. (2014)	PrC-IaaS	CS	E	U	S	U
Kavalionak et al. (2015)	PrC	P2P	E	U	U	U
Lin & Shen (2015*a*), Lin & Shen (2015*b*)	PrC	P2P	MS	U	S, M & O	U
Diao et al. (2015)	U	U	E	N	S	U
Gascon-Samson et al. (2015)	PrC	CS	MS	U	S,P	U
Shen et al. (2015)	D	CS	MS	U	S,P	U
Dhib, Boussetta, Zangar & Tabbane (2016)	PrC	CS	E	U	S	U
Negrão et al. (2016)	HC	CS	MS	U	S	U
Burger et al. (2016)	PrC	U	MS	U	S, P & O	U
Dhib, Zangar, Tabbane & Boussetta (2016)	PuC-IaaS	CS	MS	U	P	U
Yusen et al. (2016)	U	CS	MS	U	S	U
Basiri & Rasoolzadegan (2016)	PrC	U	U	U	S	U
Apel & Schau (2016)	D	CS	U	R	M	U
Matsumoto & Okabe (2017)	U	P2P	MS	U	M	L
Diao (2017)	PrC	P2P	MS	N	S,P	U
Dhib et al. (2017)	PuC-IaaS	U	MS	U	S	U
Mildner et al. (2017)	D	P2P	MS	U	S,P	U
Zhang et al. (2017)	PrC	H	U	U	S	U
Google (2018)	PuC-IaaS PuC-SL	U	E	U	U	U
Plumb & Stutsman (2018)	PrC	P2P	MS	U	S	L
Meiländer & Gorlatch (2018)	PrC	U	E	U	S & M,P	U
Plumb et al. (2018*b*)	U	H	MS	U	S	U
Farlow & Trahan (2018)	U	CS	MS	U	S	U
Ghobaei-Arani et al. (2019)	PuC-IaaS	CS	E	U	M & P	U
Tsipis et al. (2019)	PrC	H	E	U	S	U
Boroń et al. (2020)	U	P2P	U	U	U	U
Donkervliet et al. (2020), Eickhoff et al. (2021), Donkervliet et al. (2021)	PuC-SL	CS	E	N	U	U

the hosting of MMOG backends on dedicated infrastructure, as cloud approaches combine for a total of 77%.

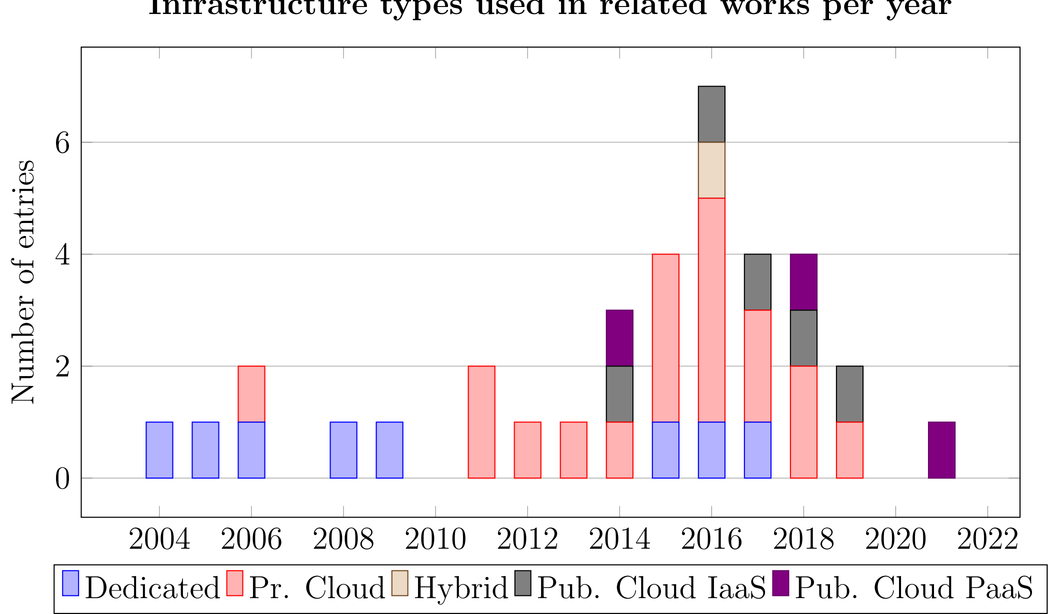


Figure 2.2: Choice of infrastructure over time — as derived from the studied works.

In addition, figure 2.2 presents the use of these infrastructure types over time. By taking into account the year of publication for each entry, it is possible to find the patterns of infrastructure use. Based on the results, the use of dedicated infrastructure is more prevalent in older studies, even though several newer studies have also utilized this approach. The rise of commercially available cloud computing services in 2006 (Lu & Zeng 2014) introduces the first use of private clouds for the deployment of MMOG backends, even though dedicated hosting remains the primary option until the early 2010s. However, the widespread adoption of cloud computing since then has led to more game developers and researchers utilizing clouds for their MMOGs. Through this period, we see an explosion in the use of private clouds, and to a smaller extent, public clouds. In public clouds, the IaaS layer remains the most popular option, even though recent studies have also begun exploring the serverless approach.

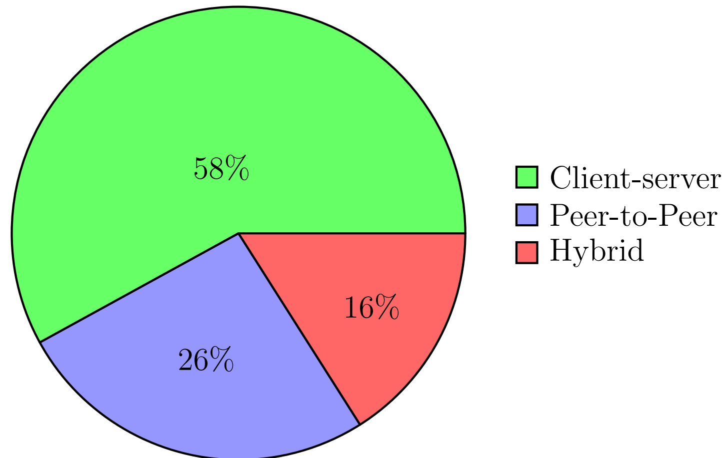


Figure 2.3: Architecture approaches used in terms of frequency of entries.

2.9.2 Architecture

The majority of entries (58%) have utilized the well-known client-server (CS) model as network architecture. The peer-to-peer architecture was used to a lesser extent, in only 26% of the studies. The hybrid client-server/peer-to-peer approach is more popular than expected, being mentioned or used in 16% of the studies. This can be attributed to the disadvantages of P2P networks when used in the context of MMOGs, which were discussed in section 2.8.2. The lack of centralized control may have driven developers to utilize hybrid architectures instead of pure P2P.

The popularity of the client-server architecture remains relatively consistent throughout the years. As evidenced in figure 2.4, both older and newer studies utilize this architecture for MMOG backends. However, in recent years, researchers have begun exploring the use of P2P or hybrid CS/P2P networks. In many studies, hybrid architectures are often coupled with fog and edge computing to provide a high QoE. As suggested by the contents of the related works and the trends seen from these results, the use of hybrid architectures in combination with fog or edge computing seems to be a promising future research direction.

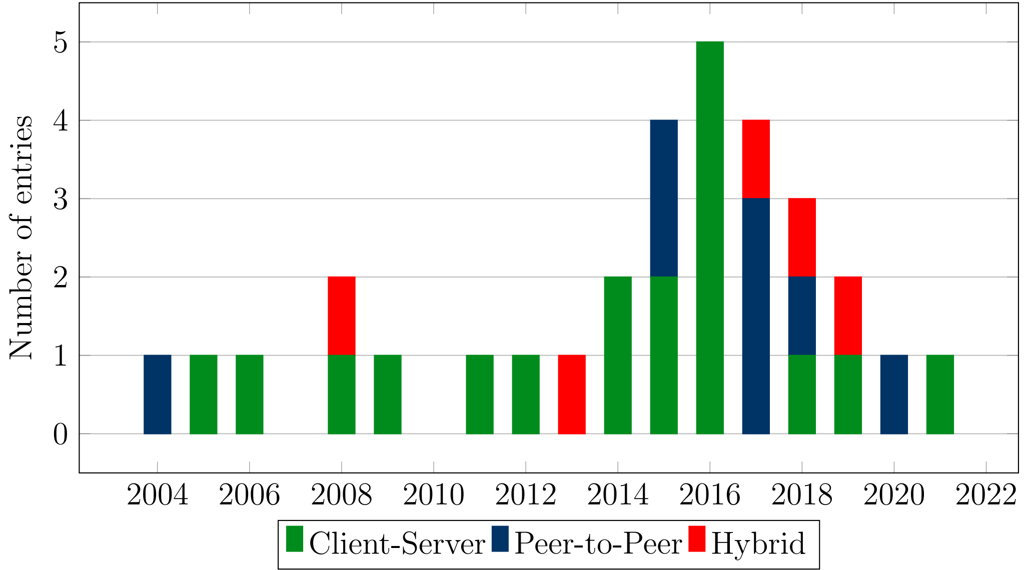


Figure 2.4: Choice of software architecture over time as found from the studied works.

2.9.3 Performance

Performance is a critical aspect of MMOG backend development and this is reflected by the large number of entries mentioning the use of various methods and metrics to measure performance. 33 out of 41 (80%) entries mention the use of at least one performance evaluation approach or tool. Out of 35 entries mentioning the use of an evaluation approach, 26 (74%) opted to use simulations to carry out their evaluation, whereas only 9 (26%) used either software or mathematical models. In addition, out of 19 entries mentioning the use of various tools to carry out a performance evaluation, 14 entries (74%) utilized either an MMOG prototype they created themselves or a pre-existing MMOG. The rest of the entries (26%) opted to use other tools or frameworks which are not an MMOG prototype or a publicly available MMOG. These results highlight the potential usefulness of simulations and utilizing prototype or existing MMOGs in evaluating the performance of MMOG backends. By far, these approaches were the most popular among the studies referenced in this analysis. In addition, it is important to note that most of these evaluations examined simple metrics such as latency, bandwidth, and resource consumption, while only a handful used their own composite metrics.

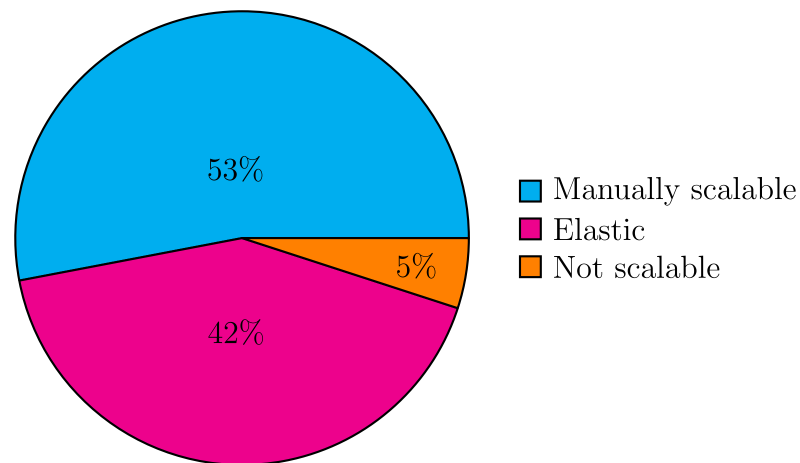


Figure 2.5: Scalability types, as observed from the approaches taken in the related work entries.

2.9.4 Scalability

While scalability can remain relatively unexplored in the early phases of development, or even when an online game is initially published, it becomes crucial when players are attracted and resource demand rises. For MMOGs, their inherent need for scalability means that it may be beneficial to think about this aspect in the early phase of development. During the early stages, it is expected that user demand will fluctuate sharply. Thus, using an *elastic* approach that enables a game to procure and relinquish resources automatically based on demand might make things easier in later stages. Despite this important advantage, the results, presented in figure 2.5 show that most paper entries (53%) used a manually scalable approach. Elastic scaling follows with 42%, while only 5% of the approaches were non-scalable.

Firstly, these results underscore the importance of scalability when it comes to developing MMOG backends – the vast majority (94%) of approaches studied offered some form of scalability. Secondly, the scalability aspect is strongly related to infrastructure, as infrastructure types can determine if a system is scalable, and what type of scalability it can offer. An unexpected outcome of these results is the fact that infrastructures that could be described as "manually scalable" or "non-scalable" vastly outnumber elastic infrastructures, but this is not the case when it comes to the actual scalability of these systems. In terms of infrastructure, the group of dedicated, private cloud, and public cloud IaaS approaches – which can be considered either non-scalable or manually scalable – vastly outweighs the group of elastic-capable

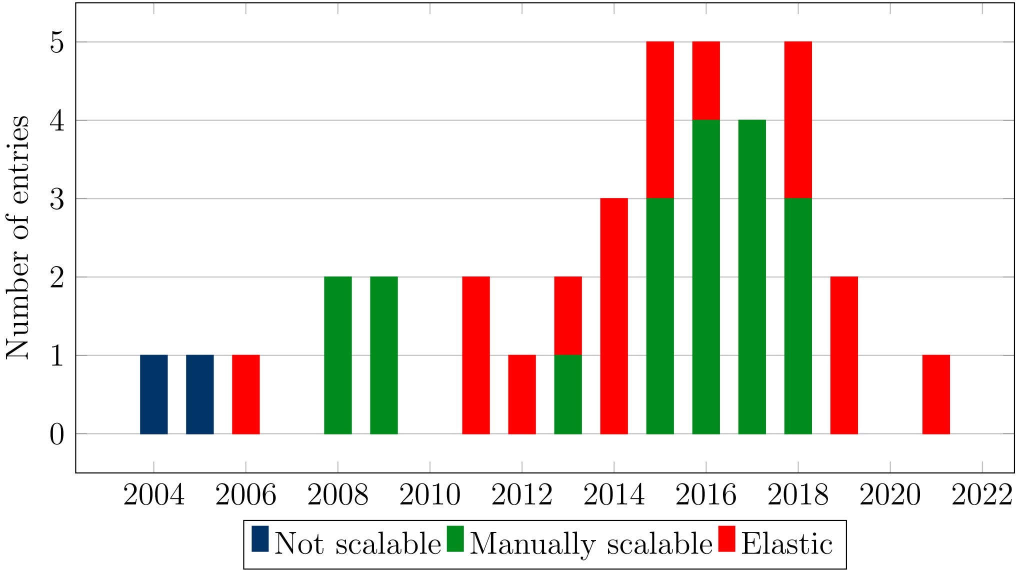


Figure 2.6: Scalability types, as observed from the approaches used in the related work entries.

approaches (serverless and hybrid cloud) by more than 4 to 1. However, when it comes to the actual scalability of the systems studied, it is discovered that this ratio is much closer to 1:1. This might be attributed to the fact that in a large number of studies involving normally non-elastic infrastructures, authors have made significant contributions to the area of scalability, either through proposing new load prediction and balancing algorithms or by using dynamic resource provisioning tools.

Analyzing the entries by year in terms of scalability also creates a clearer picture of the evolution of this aspect. As seen in figure 2.6, non-scalable approaches were only used in the earlier years of MMOG development. These appear to have been phased out, most likely due to the creation of more complex social games that attracted larger numbers of players. Thereafter, we see a mix of manually scalable and elastic approaches, evenly distributed through the years, with elastic approaches slightly edging manual scalability in the last three years. Whether this pattern is to continue is subject to speculation, but one thing is for certain: the rapid increase in the use of cloud infrastructure for MMOGs is going to make it easier to develop elastic MMOG backends.

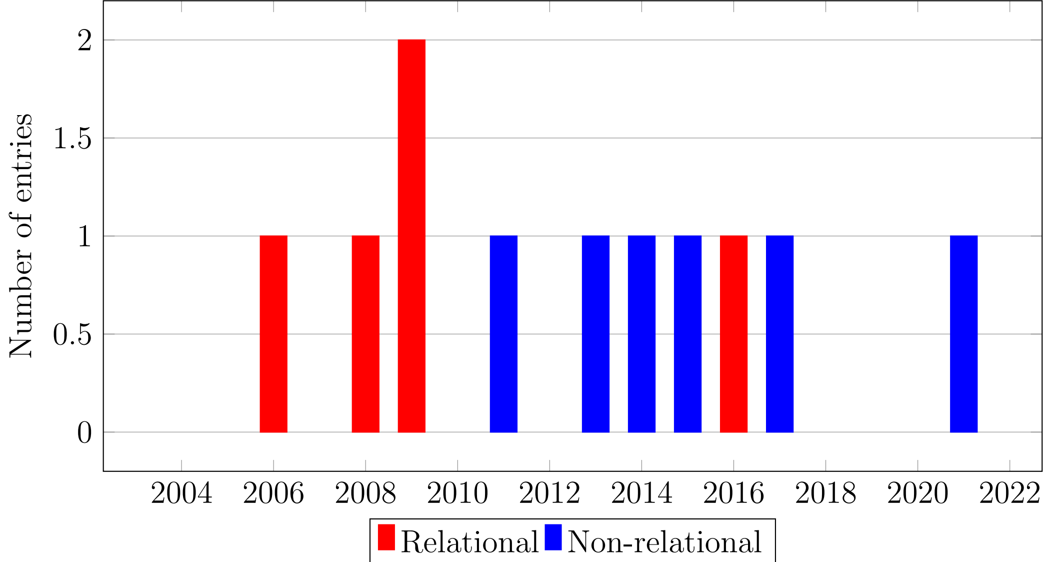


Figure 2.7: Approaches used in dealing with persistence in MMOG backends, in each year.

2.9.5 Persistence

While not as significant as other aspects, persistence is still an important feature that is necessary for the vast majority of online games. This importance mainly stems from the fact that a persistence system has a direct effect on the performance of the backend, as database interactions are some of the heaviest operations the system must perform. Most games will typically utilize multiple layers of persistence, using permanent persistence options like databases or datastores, while also employing caching systems to improve performance. These types of systems vary greatly in the features they provide, and there is no consensus on which is the best fit for MMOG backends. As the results point out, there is a relatively even split among relational (45%) and non-relational (55%) approaches, which may be statistically insignificant. The use of these approaches through the years can be observed in figure 2.7. A pattern emerges toward the use of non-relational systems, especially after 2010. Since this period, research appears to have focused more on non-relational datastores and caching systems rather than relational persistence. Even though this cannot be generalized to all existing studies, it clearly illustrates a trend toward using non-relational persistence.

2.9.6 Security

The aspect of security is left last as it is assigned a lower priority in research. This is inferred by the fact that only 6 out of 41 entries mention this aspect. However, this aspect should not be overlooked, as an increasing number of modern games include features that require more serious considerations on security. The security policies discussed in the related works are most often discussed in conjunction with the P2P architecture, where implementing security features is more challenging. Most entries (67%) that discuss security utilize loose security methods. These are based on non-architectural methods like algorithms, encryption, and more, which cannot be controlled at the hardware level. The rest of the entries use tighter security, which relies on hybrid architectures and the use of centralized systems that inherently provide more control and security. While these results reveal that software-based security is preferred, their usefulness is uncertain due to the small number of entries and the fact that this aspect is mostly overlooked, especially in research studies.

2.10 Insights and future research directions

The results presented in the previous section reveal several trends in the aspects that were studied and have reinforced the notion that the development of MMOG backends on commodity clouds, remains an open problem. Furthermore, the large number of resources found suggests that this is a research-active area.

Based on the studies analyzed, most MMOG backends are designed to be hosted on private clouds and on the IaaS layer. While this layer provides various advantages associated with the virtualization of resources, it does not offer simplified development and deployment methods and ties game logic and features to the infrastructure. Such features may be available in serverless computing environments, which have shown a promising outlook for future research. These higher cloud layers are continuously developed and enhanced by commodity cloud providers, offering new opportunities for developing scalable MMOGs that run by leveraging elastic resources. In terms of architecture, most authors opt for the safety and control provided by the

client-server model, even though hybrid architectures are also worth exploring.

The most useful performance evaluation approaches use custom simulations and pre-existing MMOGs to generate live data. Important metrics include latency, bandwidth usage, network distance, and resource consumption. Thus, the evaluation of any potential methods and tools related to developing MMOG backends may benefit from using these approaches.

Another useful insight is that the use of non-relational persistence systems seems to be preferred. Using this type of system improves performance and scalability in MMOG backends, and enables them to offer additional features. Perhaps, non-relational and relational systems can be used simultaneously to serve different purposes in the same system, in combination with high-performance distributed caching systems that can greatly improve performance.

Furthermore, security must not be overlooked in these types of systems, as newer games must incorporate additional features. The improvements in development methodologies and tools in the last ten years have undoubtedly set the stage for a revolution in terms of online game features. With some of these features involving sensitive information, security may finally take on a more important role in the development of MMOG backends.

From these insights it can be argued that there is an opportunity to enable MMOG backends to be deployed directly on serverless cloud environments, utilizing modern persistence technologies and scalability methods. To achieve this, online games must be modeled in a way that enables abstraction, and new methods and tools must be created to improve their development process.

Chapter 3

Feasibility study

"*The value of a prototype is in the education it gives you, not in the code itself.*"

Alan Cooper

3.1 Introduction

Chapter 2 presented the state of the art in the development of MMOG backends and analyzed several research problems and questions. Previous studies have made outstanding contributions in this area over the years by proposing new architectures, frameworks, performance evaluation tools, algorithms, and much more. However, to the extent of the author's knowledge, only one vision paper has explored the potential of utilizing commodity clouds and serverless environments for hosting MMOG backends, while others have focused on other approaches. Despite its obvious advantages, this area remains relatively unexplored, requiring further study to explore its feasibility and potential. This section reports an exploration of how commodity cloud platforms and the services they provide can potentially support the development of MMOG backends, and discusses their limitations, through an experimental feasibility study.

3.2 Objectives

To assess the capabilities of commodity clouds to provide the necessary features for a scalable MMOG backend, a multiplayer game is implemented and evaluated using three approaches including both IaaS and PaaS environments. The first objective of this experiment is to identify the constraints, peculiarities, and challenges presented when developing online games on serverless platforms. Secondly, it aims to explore how these development processes are different from those found in IaaS, or more traditional hosting approaches like dedicated servers. Thirdly, it attempts to enable a performance comparison between these approaches based on latency – which most significantly affects the QoE. Through this experiment, several exploratory questions mentioned in section 1.3 are also investigated.

3.3 Experiment overview

The experiment is conducted by developing a version of Minesweeper (Becker 2001), a popular puzzle game introduced in the 1960s. Minesweeper is adapted as a multiplayer game, based on three different cloud deployment methods. Those familiar with the game will quickly realize that Minesweeper is a single-player game, that features no support for online gameplay. In this feasibility study, the game is modified to run as an *Online Multiplayer Game* (MOG), allowing players to join the same session and simultaneously play the same game. Despite being a traditionally single-player game, Minesweeper was chosen for several reasons. Firstly, it has a relatively simple set of rules, which reduces development complexity, while also compelling the developer to enforce these rules by programming them into game services. Secondly, it is a 2D puzzle game, which means that no advanced 3D graphics are required to present the game's state to the player – something that is beyond the scope of this research. Thirdly, it presents the challenge of modifying an existing game, with a known model, to run as an online game thus differentiating the items being developed for its online functionality from those that would already exist in its current version. Lastly, it can be used to demonstrate the requirements for developing a backend that supports the features identified in section 2.6, such as satisfactory

performance, scalability, consistency, and so on. Data from the three different versions of the game are measured to deduce its performance and scalability using simulations, which may enable comparison between these approaches.

The implementation and evaluation of this feasibility study build on some assumptions. Several factors are kept in control, which are discussed in section 3.6. All implementations use similar types of services across the three tested commodity cloud providers and utilize the same logic, data model, APIs, and software architecture. For this particular study, no considerations are made for scalability, resource allocation, consumption, and monitoring, with focus instead placed on understanding the software processes and challenges that arise when developing an MMOG backend.

3.4 Implementation

In Minesweeper, the game state can be represented using a two-dimensional *grid* array, which is also known as a *tile map* (Coleman et al. 2005) or *matrix*. Tile maps are a common type of game state that is also utilized in games like *Clash of Clans* or the *Civilization* series. Such games offer *persistent worlds*, which means that their state is sustained for a long time, unlike other types of games which are executed in sessions and are finished when a set of conditions is met. In these types of worlds, players may control entities and/or have a direct effect on the game state through their *actions*. In a multiplayer Minesweeper game, for example, a large game board could be used to enable gameplay, allowing players to simultaneously attempt to solve a single puzzle. To model the state of the game, classes such as the `CellState`, `BoardState`, and `GameState` are created, which model different parts of the game. For instance, the `CellState` class models the state of a single cell, which in Minesweeper can be either a number indicating the adjacent number of mines (0-8) or a mine (*). In addition, to develop the game, its rules must be enforced and its actions must be identified. In Minesweeper, the player can perform three simple actions – either *reveal*, *flag*, or *unflag* a cell, which can be identified by its coordinates in the array. In addition, the following rules are identified and

implemented: (a) players can make a single move on a single cell per turn, (b) when an empty cell is revealed, the game should display the number of mines placed in adjacent cells, (c) if a revealed cell has no adjacent mines, all adjacent cells without a mine should be recursively revealed, (d) when a cell containing a mine is revealed, a point penalty is applied, and (e) when there are no mines to reveal, the game ends. This implementation deviates from the original rules of the game in (d), where the game would normally end in a loss if a mine is revealed. This is changed to allow experiments to run for longer periods of time and therefore yield more useful data.

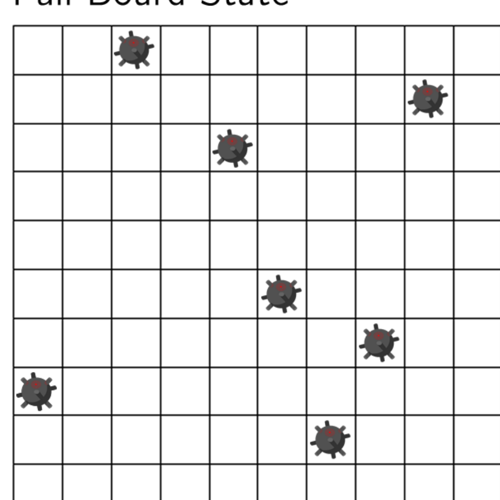
To enable the evaluation of different hosting approaches, nearly identical versions of Minesweeper are developed for three different cloud platforms:

- Amazon Web Services (AWS), using EC2 instances and DynamoDB
- Microsoft Azure, using VM instances and CosmosDB
- Google Cloud Platform (GCP), using App Engine and Cloud Datastore

The selection of these platforms enables a meaningful comparison between the services provided by three major public cloud providers. To keep things fair, all projects are implemented using Java 8 and the same code base to model the game state, logic, rules, and so on. In addition, the client of the game, which is mainly used for simulations, debugging, and state representation remains identical in all approaches. The client can optionally run with graphics, although this is normally disabled to improve performance. The client runs simulations by concurrently running player threads that simulate the behavior of players. To enable different configurations during simulations, the client can be provided with a variety of parameters, including the number of players, the delay between player actions, the game board size, and more. The results of a simulation are saved by the client in CSV (*Comma-Separated Value*) format after the simulation is completed. These include simulation information such as the number of players, the board size, timestamps at which requests and responses were sent and received, latency data, and more.

The client-server architecture is used as it is simpler, provides more control, and has received extensive research and support from the software development community. In this architecture, several components are identified when constructing an MMOG backend, such as the *client*, *server*, and *database*. In its simplest form, the server hosts the game's backend, implementing the game and providing access to the game's services for the clients. The functionality of these services is exposed in a set of commands, accessible through an API which the client can interact with. The game consists of five core functions which are exposed through this API: `/createGame` is used to create a new game session and board, `/list` enables players to list available game sessions, `/join` allows them to join a session, `/getState` fetches a game's state, and `/play`, performs an action on a selected cell. Meanwhile, to persist game state information the server interacts with a cloud-based datastore provided by the corresponding cloud platform.

In all implementations, the *Area of Interest* (AoI) concept discussed in section 2.8.4 is used to reduce the bandwidth required to communicate the state of the game. To achieve this, the state of a board is differentiated into `FullBoardState` objects, which feature the entire state of a single game board, and `PartialBoardState` objects, which include only a part of the game state, around a certain area. This is utilized to provide context-based states to players, based on their camera position. For instance, if a player is currently "focused" at position (5,5), and their AoI is set to 2 cells, they would be able to view a partial state of the board spanning 2 cells outwards of this position in each direction. This is illustrated in figure 3.1. In this implementation, full board states are used by the backend to perform state computations, whereas partial board states are communicated to players during the game to reduce latency. It is noted that players may only perceive the game state within their AoI, shown using the red square in figure 3.1, but can freely re-focus their position at any time, retrieving a different partial state.



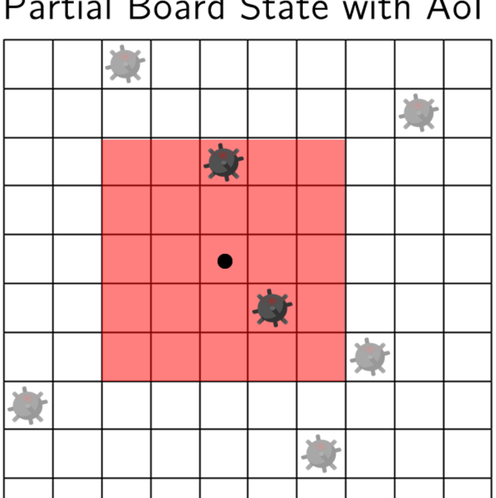


Figure 3.1: Illustration of the differences between full board states and partial board states using the AoI concept. In the right figure, the player's AoI is highlighted in red color, with translucent mines being outside of the AoI and not being perceived by the player.

3.5 Approaches

To evaluate these experimental implementations, the game was initially deployed on AWS EC2 t2.micro instances running Linux Ubuntu 18 with 1vCPU, 1GB of RAM, and "*low to moderate*" network performance (Amazon Web Services 2019*a*). The VM instance communicates with a low-latency DynamoDB instance, Amazon's NoSQL data store, with provisioned capacity within the free tier (Amazon Web Services 2019*b*,*c*). In this project, Apache Tomcat (Foundation 2019) is used to host game services that are powered by Java Servlets, with each of these implementing a different endpoint of the API as discussed above. To communicate with these endpoints, the client uses HTTP to issue requests and receive responses. To efficiently manage data in DynamoDB, a library called DynamoDBMapper (Amazon Web Services 2019*c*) is used, which maps classes to DynamoDB objects using code annotations. The project uses Ably (Ably 2019), a scalable real-time messaging technology to publish state update messages to clients. Using this approach a client can subscribe to channels created by the server, and receive state updates when necessary to reduce bandwidth consumption.

Similarly, the Microsoft Azure-hosted version of Minesweeper is powered by a B1S-type VM

running on Linux Ubuntu 14 with similar characteristics: 1vCPU and 1GB of RAM. The two projects are almost identical, being powered by Apache Tomcat and Java Servlets, and utilizing the exact same game model, algorithms, game API, and state update mechanisms. The only functional difference between these two projects is the datastore being used. In this approach, Azure's CosmosDB (Microsoft Azure 2019) is used for persistence. CosmosDB is very similar to DynamoDB and offers real-time access to data regardless of scale.

The third approach is based on Google's App Engine (GAE), a serverless platform that fully manages application deployment without the need to deal with server configuration. This allows applications to scale seamlessly and without any supervision, which is a major advantage over the previous two approaches. This project uses App Engine's Java 8 version with Jetty 9 (Eclipse 2019), which is similar to Apache Tomcat used in the other two approaches. Similarly, communication occurs over HTTP, and the game's model, components, API, and state update mechanisms are kept identical. App Engine works well with the utilized Cloud Datastore (Google Cloud 2019), which is Google's NoSQL key-value store service. To utilize this datastore efficiently, a library called Objectify is used (Objectify 2019).

3.6 Evaluation

The evaluation of these three approaches is guided by performance and scalability, as these are considered the two primary aspects of success in an MMOG. During this evaluation, several factors have been kept in control, which are listed below:

- The network conditions were kept as similar as possible by running the experiments within the same wired network. The network was monitored, verifying that it was not being utilized by other programs at the time. The experiments were run at similar times and days of the week to minimize the risk of invalid data due to different network conditions.
- The client device conditions were kept as similar as possible by running all simulations on the same computer while it was initially idle.

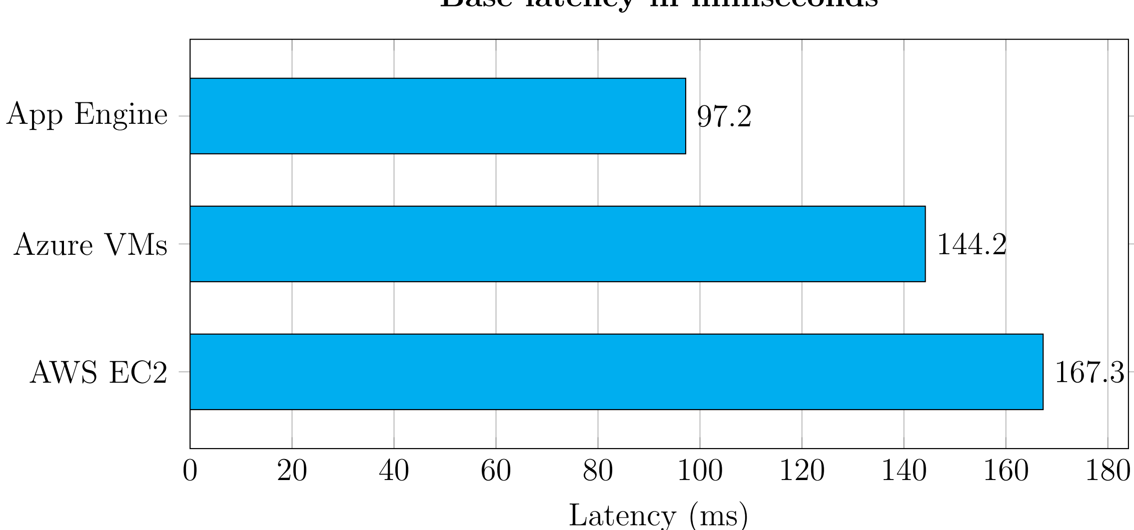


Figure 3.2: Base latency in each approach, in milliseconds (ms).

- Comparable data center locations were used (Eastern United States) for all experiments.
- NoSQL data stores were used for all cloud approaches to allow a comparison between them. This type of persistence is used because it can be easily scaled and appears to better match the needs of MMOG backends (Baker et al. 2011, Chang et al. 2008, Shabani et al. 2014).
- Virtual machines with similar specifications were created to keep the backend processing power as comparable as possible.
- Experiments were conducted based on the same model and a set of identical commands and made sure that the parameters and logic of those calls stayed the same throughout all simulations.

Initially, a *base latency* is established for each approach using a call to an inert service. Base latency is used to establish a minimum latency for each deployment approach which is only affected by network communications and data center latency rather than game code. The calls to this service were repeated 10 times in each approach, and averages were calculated. As shown in figure 3.2, App Engine surprisingly performs significantly better than the IaaS approaches, at 97.2ms latency, while Azure VMs has a base latency of 144.2ms, and AWS EC2 has 167.3ms.

In addition to the base latency, several tests are carried out to determine the maximum size of the game board state possible in each approach. In this experiment, the size of the game board is limited by the size of each entity being stored in each of the data stores being utilized, as each game board is stored in a single database entry. This entry data limit varies throughout the different datastores, with DynamoDB supporting only up to 400KB per entity, Cloud Datastore up to 1MB, and CosmosDB up to 2MB. While there may be ways to circumvent these limitations to support bigger states, this feasibility experiment does not attempt to solve this issue for several reasons. Firstly, this is beyond the scope of this experiment. Secondly, workarounds implemented for a specific datastore may not necessarily work for others, making it difficult to compare the results. Thirdly, to keep the experiment fair, it is necessary to keep the implementations as simple and consistent as possible. This experiment is conducted by creating square-sized game boards, starting from the size of 100x100 cells, and increasing the size by 50% each time the datastore could successfully store the state. In cases where the datastore failed to support the scale of the board, the size was reduced by 25%. Eventually, the exact size of the state in terms of cells was discovered for each approach. As seen in figure 3.3, Azure's CosmosDB was able to support the largest state, up to 229x229 cells, with GCP's Cloud Datastore supporting up to 158x158 cells and AWS's DynamoDB supporting 98x98 cell states. These results directly correlate to the maximum entity size for each of the datastores. It is also worth noting that these state sizes are dependent on the game state data being recorded. For instance, games with simpler states may be able to scale to bigger sizes while those with more complex states may not be able to scale as much.

Upon establishing the maximum state size supported by all datastores, the minimum of these values (98) is used for the next experiment. This test evaluates the performance of the `/createGame` service. HTTP requests were sent to this service, which included various parameters such as the maximum number of players allowed in the created game, the board size, difficulty, and so on. Throughout the experiment, these parameters were kept constant. The results, out of ten calls to the `/createGame` service in each approach, reveal a higher latency in Google's App Engine (496.6ms), while AWS EC2 and Azure VMs were both significantly lower (332.7ms and 346.3ms respectively).

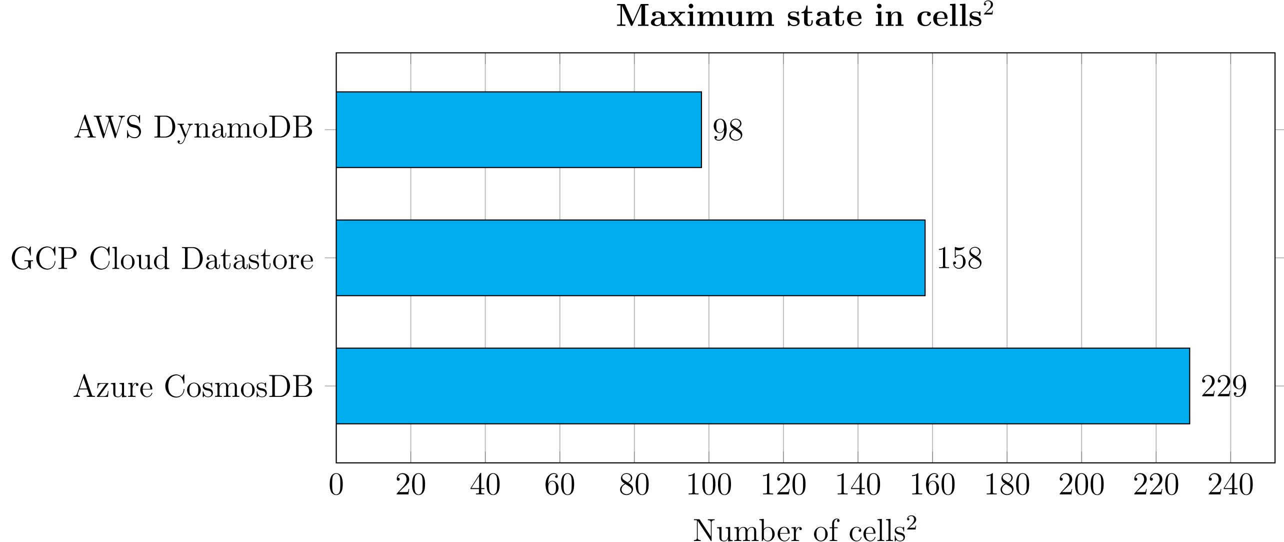


Figure 3.3: Maximum state size supported by each of the datastores used, in cells^2

The next test focuses on the `/list` service, which lists the available games, without returning the game's board state. In this test, two games were created in each of the datastores, and the service was called to measure its latency. Out of ten repetitions, Azure had the lowest average latency, taking 555ms to list the available games, while AWS took 568ms and App Engine took significantly longer, at 1153.2ms.

In order to evaluate the performance of the `/join`, `/getState`, and `/play` services ten simulations were executed with a specific configuration: a game size of 10x10 with only two players, the difficulty set to easy, and a partial state of 5x5 for each player. When joining a game, AWS performed best with a latency of 201.8ms, compared to Azure's 234.8ms and App Engine's significantly longer 554.6ms. This approach also performed marginally better in retrieving the state of a game, at 176.3ms, with App Engine taking 176.7ms and Azure 245.4ms.

The most relevant test is the one related to the `/play` service because calls to this service are made continuously and in rapid succession during gameplay. In contrast, other services are only called once at the start of the simulation by each player. Therefore, the `/play` service has by far the most impact on performance. As shown in figure 3.4, AWS and Azure approaches performed very similarly, with 176.9ms and 175.8ms average latency respectively. Meanwhile, App Engine was slower, at 201.2ms of latency, even though this is still good performance given

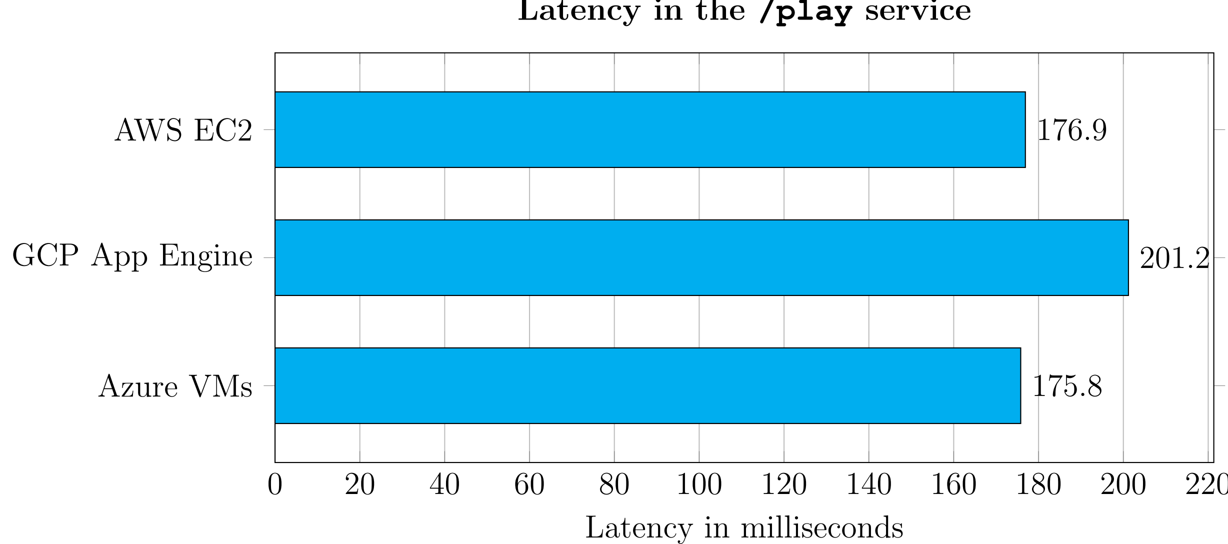


Figure 3.4: Average latency for the `/play` service.

the large network distance.

3.7 Commodity cloud support for MMOG backends

The implementation of Minesweeper as a multiplayer game that can be hosted on three major commodity cloud platforms, at different computing layers, and without any modifications or improvements, demonstrates the ability of public clouds to support MMOG backends. The findings presented in this section suggest that MMOG backends can be engineered to run on these types of clouds, not only at the IaaS level but also in serverless environments like Google's App Engine. The experiments discussed above have utilized several services provided by public clouds, including computing, data persistence, deployment, and more. This shows the readiness of commodity cloud computing to be used for online games, and as an extension for MMOG backends.

Furthermore, the results showed that IaaS approaches generally perform better than PaaS, which is expected, as serverless approaches have higher overheads. However, for a specific set of games that do not require very low latency, such as RPGs and RTSs, serverless environments can be viable options that offer many other advantages, including inherent elasticity, abstraction, and much more. It is therefore concluded that serverless layers can support at least a subset

of game genres, which provides the opportunity to further explore these types of systems.

These experiments also uncover several constraints and technical limitations of the mentioned systems. For instance, the serverless layer presents several problems, including no support for bi-directional communication methods, mainly due to hard time limitations in service execution time. For MMOG backends specifically, this is an important problem, as state updates must be somehow distributed to clients in a timely manner. In this experiment, third-party frameworks were used to achieve this functionality. However, it is envisioned that a framework that supports the development of MMOG backends must internally provide solutions for such problems.

The utilization of several key-value datastores also reveals problems in terms of scalability. While these services are considered more scalable, they enforce hard limitations on the amount of data that can be stored in a single entity. This means that game developers utilizing these datastores must find ways around these limitations to enable scalable game states. Currently, to the best of this author's knowledge, there is no standardized way to achieve this and it is anticipated that developers utilizing such services will provide ad-hoc, game-specific solutions to this problem. While this is a reasonable assumption, it is argued that providing a standardized way to achieve scalable worlds will dramatically improve the development effort by allowing programmers to seamlessly scale their world without having to implement such game-specific solutions.

The results[1] presented in these experiments have some limitations. For one, the number of players used in the simulations was kept relatively low, which is not representative of commercial-scale MMOGs. Secondly, Minesweeper is implemented as a turn-based game, which does not allow the generalization of these results to other types and genres of games. More importantly, the experiments only lightly explore the issue of scalability through the datastore tests. To improve the understanding of how these systems work and how they could potentially support larger scales in terms of players and states, further exploration is required. Nevertheless, this feasibility study demonstrates the possibilities offered by commodity clouds and serverless systems to support MMOG backends and provides the foundations to construct the necessary

[1] The raw data for results presented in the feasibility experiments is available in appendix 9.A.

models, methods, and tools to fully reinforce their development. Despite these limitations, it encourages the consideration of important technical questions:

- In this test, the services are tied to a specific approach used by the infrastructure (i.e. web containers) which reduces the clarity and maintainability of the code by mixing infrastructure management with game logic. *Is it possible to de-couple service logic from each infrastructure approach to improve code maintainability?*
- The full board state, while useful in this test, may severely impact the performance of a backend that serves a very large game world due to the large amount of data being transferred from the datastore. *Is it possible to utilize partial states for computational purposes as well? How can this be done seamlessly, without forcing the developer to retrieve each state part individually?*
- In IaaS approaches, no considerations are made for the scalability of the game's runtime (i.e. supporting larger numbers of players). *How can this be handled for such non-elastic environments?*
- The Minesweeper implementation features simplistic, relatively disjoint states without any moving entities. Different games may need to feature entities that can move in the game world, and which can interact with each other. In addition, players may need to control more than one entity at the same time. *How can game worlds and entities be modeled in a way that handles all of these different functionalities?*
- Game worlds may vary greatly from those created for Minesweeper, requiring entities to freely move in space rather than attaching them to specific coordinates or cells. *How can different types of worlds be supported using the same code base?*
- The communication protocol and game API are exclusively tied to this implementation and cannot be reused in any other game, even though there may be similarities in their functionality. *Is it possible to abstract the communication layer and API creation to accelerate the development of game services?*

- The three approaches discussed utilize a single datastore and tie their service logic to each of these services, which reduces code maintainability and the ability to experiment with different persistence options. *Firstly, is it possible to utilize multiple layers of persistence in the same MMOG backend, like a datastore supplemented by a caching system? Secondly, how can the persistence layer be abstracted to reduce code dependencies with other layers and ensure that the backend can be maintained and expanded?*
- The AoI concept, while useful in this test, is only based on proximity and does not provide the flexibility to utilize other factors, such as obstructions, relevance, etc. *How can this concept be abstracted for use in different types of games? How can it be generalized, yet allow for customizations for game-specific needs?*
- Similarly, the state-update framework used in this test is tied to a specific third-party approach. As such, it may not be feasible to utilize a specific tool for all games. *How can this component be abstracted and standardized, yet also allow for customizations needed by specific games?*
- In these experiments, no considerations were made regarding security. *How can a rudimentary level of security be offered to secure MMOG backends from attacks? Can practices like encryption and hashing be provided to and used by developers without them having to learn how they work?*

3.8 Conclusions

This chapter presented the development of an MMOG prototype and its deployment on three different commodity cloud services, in the context of a feasibility study. Benchmarks which recorded the performance and scalability of this prototype showed that services offered by commodity clouds can support MMOG backends. As expected, dedicated servers deployed on cloud IaaS performed better than those utilizing serverless offerings. However, for a specific set of games that do not require very low latency, serverless computing is well within the performance constraints and offers a significantly more streamlined development approach.

Finally, the feasibility study has identified important limitations and technical challenges that are associated with the development of MMOG backends and the use of commodity clouds. These challenges can be addressed to provide developers with the models, methods, and tools needed to create scalable MMOG backends – ultimately motivating the development of a new software development approach to solve these problems, which is presented in chapter 4.

Chapter 4

The Athlos framework

"*Computer Science is a science of abstraction – creating the right model for a problem and devising the appropriate mechanizable techniques to solve it.*"

Alfred Aho

4.1 Introduction

The feasibility study presented in the previous chapter has established the possibility of deploying MMOG backends on commodity clouds. To the author's understanding, this is an advancement of current knowledge in itself as no other study has explored this possibility in a practical way. Furthermore, the experiments described in this section have raised a significant number of questions that can be attributed to several challenges related to the development of MMOG backends and their deployment on public clouds.

This section first motivates and then presents a suite of novel models, methods, and tools with which scalable MMOG backends can be hosted on various layers of commodity clouds. Using the knowledge gathered from the literature and experience acquired from the feasibility study, an investigation is made into how such resource-intensive and latency-sensitive applications can be developed for these types of clouds. The general focus of this investigation is to navigate several problems related to scalability, consistency, performance, and maintainability. The ap-

proach presented in this section targets a specific set of games that feature fully-persistent, scalable states and must be able to support large numbers of concurrent players without necessarily attaining very low latency. Nevertheless, the approach may be able to handle other types of games as well, although this is not within the scope of this thesis. The proposed models, methods, and tools are partly based on the *observation* and application of methods, techniques, and phenomena that were established by previous research and are adapted to work harmoniously within a unified framework that also includes new, *innovative* contributions.

The models, methods, and tools which are proposed are integrated into a *scalable MMOG backend software development framework*[1] called *Athlos*. Athlos includes a *default model*, which aims to abstract the development process and allow developers to quickly and easily create applications. This model can be customized and expanded, offering the ability to create a large variety of games. It also de-couples various game development components and functions and thus provides the opportunity to use a modular architecture with independent components. Such processes, which may be common across games, can utilize standardized *methods* which can reduce development time and effort while also providing facilities that allow games to attain better performance and scalability. To facilitate the use of these models and methods with software development environments, Athlos also provides *tools* with which developers can create and manage abstract game definitions which are not tied to a specific development approach, and with which it is possible to generate boilerplate projects in a variety of languages. This promotes analysis, experimentation, and the rapid development of game prototypes. It is predicted that using Athlos in combination with serverless computing environments will provide a means for better utilizing commodity clouds and the opportunity to develop more scalable MMOG backends.

[1] A software development framework is defined to be a software abstraction which enables the development of software using a set of guidelines, specifications, and tools which allow the developer to create an MMOG backend by leveraging existing code that helps solve various problems related to development, without having to solve these problems by writing code from scratch.

4.2 Motivation

To further promote the process of developing the new MMOG backend development approach, this section describes other frameworks similar to what is being envisioned. Using inspiration from these, and by exploring a conceptual MMOG case study, it elaborates on the thought processes and motivations behind the models, methods, and tools which are presented and analyzed in the subsequent sections.

4.2.1 Other frameworks

Based on the information gathered from the related works, it is obvious that the development of MMOG backends is a lengthy and complicated process that involves a variety of aspects, all of which require their own unique set of skills and knowledge to master. There are many game development frameworks that support online multiplayer gameplay, many of which fuse third-party offerings in an attempt to utilize specialized tools for each aspect. A popular framework is Photon Engine (Engine 2022), which provides multiplayer functionality to games developed using Unity by utilizing private cloud services. Another example is Amazon's GameLift (Services 2022), which provides multiplayer functionality through dedicated servers. Perhaps the most relevant among these is Amazon's GameSparks (Amazon 2022), which is an in-development platform that allows the deployment of multiplayer games on serverless cloud infrastructure. Many of these frameworks provide tools that aid developers in managing various game requirements such as social networking, game economies, matchmaking, and more, while others also offer the ability to go serverless.

Despite their usefulness, these frameworks have limitations. For instance, some of these frameworks may offer elastic, serverless infrastructure to games, but lack the development tools necessary to create them. On the other end, many of these frameworks offer the tools and methods to develop MMOG backends but lack the necessary abstractions that would enable them to be developed using a variety of tools and for different types of infrastructures or clouds, or as elastic applications. In many cases, the models of such frameworks may be incompatible

with other tools, which complicates the development process. This thesis uses the limitations of these frameworks as motivation to develop a framework that can enable MMOGs to (a) feature an abstract, approach-agnostic model, (b) utilize standardized methods to reduce development effort and time, and (c) be deployed on a variety of environments and commodity clouds, including serverless technologies.

4.2.2 Case study: Mars Pioneer

To aid the presentation of the proposed approach and to motivate its creation, this section introduces a case study MMOG called Mars Pioneer. Mars Pioneer (MP) is a conceptual real-time strategy multiplayer online game in which players must colonize the planet Mars by building constructions in their base to gather resources, conduct research to improve their resource-gathering rate, and ultimately win over their opponents by controlling a larger percentage of the planet.

The game *world* is infinite and can expand as more players join the game, thus offering continuity to its gameplay. New *players* are inserted into the world at a safe distance away from their opponents, but not too far away to allow for meaningful gameplay. When players initially join the world a building is automatically constructed for them called a "Hub", which acts as their empty base. Players must carry out *actions* that are validated by the game's *rules* in order to interact with the game *state*. For example, they can construct more buildings using the initial resources given to them at the start of the game or by collecting resources to further expand their base. Buildings are *entities* that exist within the game world and can be constructed or sold but cannot move or be moved. Each of the buildings can only be constructed on specific types of terrain, enables the player to gather a particular resource, and has a specific resource cost. For instance, a farm can only be constructed on *soil* terrain (assuming the use of the soil in an imaginary greenhouse-like structure) and allows players to gather food. Another example is a mine building, which can be built on rocky terrain and can extract metals for construction. These resources can be gathered once a minute, with uncollected resources being compounded. The game world is *tile-based*, meaning that entities can only exist on pre-defined points inside

a grid-like structure. To generate this infinite, expandable world, it is necessary to utilize some form of *procedural terrain or level generation.*

MP allows the identification of several requirements and fosters the development of an abstract model that could potentially be used for other types of games as well. A complication during this process is that the proposed approach must offer a balance between abstraction and development effort. These two properties may contradict each other in the sense that an approach that is too generic may hinder the development process, slowing it down and making it more difficult. On the other hand, an approach that is geared towards rapid development may not offer the necessary level of abstraction. To acquire both of these properties in a balanced way, the proposed approach may have to make some sacrifices in both. Thus, it would be acceptable for the framework's structure and code to not be as optimal as possible in some specific scenarios, in order to ensure the necessary level of abstraction, and vice versa. This is further complicated by the fact that it may be impossible to foresee all prospective game models and development techniques, simply because there are many types of games and technologies. Consequently, the approach should (a) be flexible enough to provide customization and expansion options alongside those utilized by default, (b) foster the development of unforeseen requirements in as large a set of games as possible, and (c) allow developers to create high-performance, optimized MMOG backends.

4.3 Model

This section presents an abstract model that can be used to develop MMOG backends. The model describes various types of game and management objects, their attributes, and the relationships between them. It attempts to capture a wide range of game modeling requirements using object-oriented principles for multiple types of games, while still allowing customizations and expansions for specific games. Using this model, developers can rapidly create code without the need to redefine these properties in each game, thus reducing development effort through code reusability. Furthermore, this abstract model enables the use of various methods and tools

that offer solutions to a variety of problems, which are presented in section 4.4.

4.3.1 Data types

Before exploring the details of the model it is important to define how data types are categorized. The model features a set of basic data types which are defined by Google's *Protocol Buffers* (PB) (Feng & Li 2013) – also known as *scalars*[2]. These scalars are basic data types that can be found in a variety of programming languages: characters, strings, integers, and floating-point numbers, which include variations like unsigned and signed versions, as well as shorter and longer range versions. The use of PB and the motivations behind it are discussed in section 4.4.5. Apart from scalars, the framework defines *default data types*. These are essential data types for developing an MMOG backend using the framework, are included by default, cannot be removed, but can be customized to include additional information. Two *generic data types* are also included: *lists* and *maps*, enabling developers to create either a sequential list of items or a map of key-value pairs using specific data types[3]. Finally, games can be customized by declaring *custom data types*. These are developer-defined types that are game-specific and can be used just like other data types. These are sub-divided into custom *classes* and *enumerators*.

4.3.2 Type extensibility

The default types in the presented model are categorized into two groups based on whether they can be extended to form sub-types – a property similar to *inheritance*[4]. Some objects may be *Non-eXtensible* (NX), which means they cannot be specialized to form sub-types. These non-extensible types force the use of a single data model across all their instantiated objects. An example of a non-extensible type is the *Player*, which incorporates personal information about a player. In this particular case, it makes sense to record the same personal information about all

[2]Scalar data types are described here:
https://developers.google.com/protocol-buffers/docs/proto3#scalar

[3]Any generic data structures can be used in code, but data serialization mainly relies on these two types.

[4]Type extensibility provides the ability to implement inheritance and polymorphism at the framework level, rather than the project level. This is further elaborated in section 4.3.3.

players because all of the players can be described by an identical set of properties. Conversely, *extensible* types (X) are those that can be specialized to form sub-types. The data model of an extensible type can be extended in its sub-types, allowing them to record extra information in addition to those originally defined in their parent type. These specializations are necessary to capture the requirements of each game's unique mechanics and modeling requirements. For instance, *actions* differ greatly from game to game and within games themselves, requiring these specialization properties. Additionally, there might be different types of *entities* in a game, featuring different characteristics. Extensible types cannot be instantiated themselves, rather acting as an *abstraction* for defining sub-classes and subsequently enabling the use of object-oriented programming principles like inheritance and polymorphism.

4.3.3 Static and dynamic models

The development of MMOG backends using the Athlos framework involves the use of a variety of tools that simplify the work of game developers but complicate the structure and functionality of the framework itself. As mentioned above, one of the main objectives of the proposed approach is to utilize an abstract, yet customizable model which works in conjunction with various technologies that eliminate the need to develop some common game components. While this approach will be discussed in more detail in section 4.4, it is necessary to preface the presentation of the model with information that helps understand how it works.

At the center of the proposed approach lies Google's Protocol Buffers (PB) mechanism, which is utilized to serialize information transmitted to and from the backend. Although this mechanism provides a lot of potential for various abstractions both within the model and during the development of MMOG backends, it does not support inheritance – a very important principle of object-oriented design that helps create higher-quality, reusable code. The only way to support a limited level of inheritance and use PB at the same time is to utilize composition – i.e. create different, unrelated objects and manually define composition properties among them. This approach is seen as problematic, as (a) it does not capture the modeling requirements of many use cases, (b) it adds many overheads for developers, (c) it complicates the design of the

model, and (d) creates a higher likelihood of making mistakes when developing the backend. To support inheritance while also using PB, the framework's model is split into three sub-models, some of which are dynamically generated. The purpose of this division is to enable control over code design at the framework level, rather than at the development level, using *dynamic model generation* to circumnavigate many issues related to supporting inheritance and polymorphism. The use of Protocol Buffers and how its lack of support for inheritance and polymorphism is handled is further discussed in section 4.4.5.

The first component of the model is the *Athlos API model*, which is part of the Athlos API (discussed in section 4.4). This is a fully abstract set of interfaces modeling the functions of various game objects like *players*, *entities*, and so on, and is fully static – i.e. it does not change regardless of the game being developed. The second component is the *Athlos model* or *default model*, which is a set of abstract classes that contain the default data types, attributes, and operations, as intended for use within the framework. This component is semi-dynamic, meaning that it has a default set of attributes and methods for each class, but can be dynamically changed or extended based on the type of game being implemented. For example, if a game features a certain type of world, the `World` class model can be adjusted to adhere to the functionalities of that type of world. The third component is the *game-specific* model, which is fully dynamic. It is based on the Athlos (default) model and implements the functionality defined in the Athlos API model, but can also be extended to include additional attributes. For an example, see figure 4.1, which illustrates the use of the `Player` class through these components. The `IPlayer` interface, with light purple color, is part of the Athlos API model and defines the default functionality of players. This is implemented by the `Player` class, an abstract class that incorporates the default attributes of players. Subsequently, game-specific classes like `MyGamePlayer`, are dynamically generated, extend their abstract counterparts (e.g. `Player`), and can be fully customized to game needs by featuring their own attributes and operations.

This separation makes it possible to control class relationships and define inheritance at the framework level. This can be done by defining relationships between classes within a *game definition*, rather than in actual code, thus circumventing the problematic nature of using

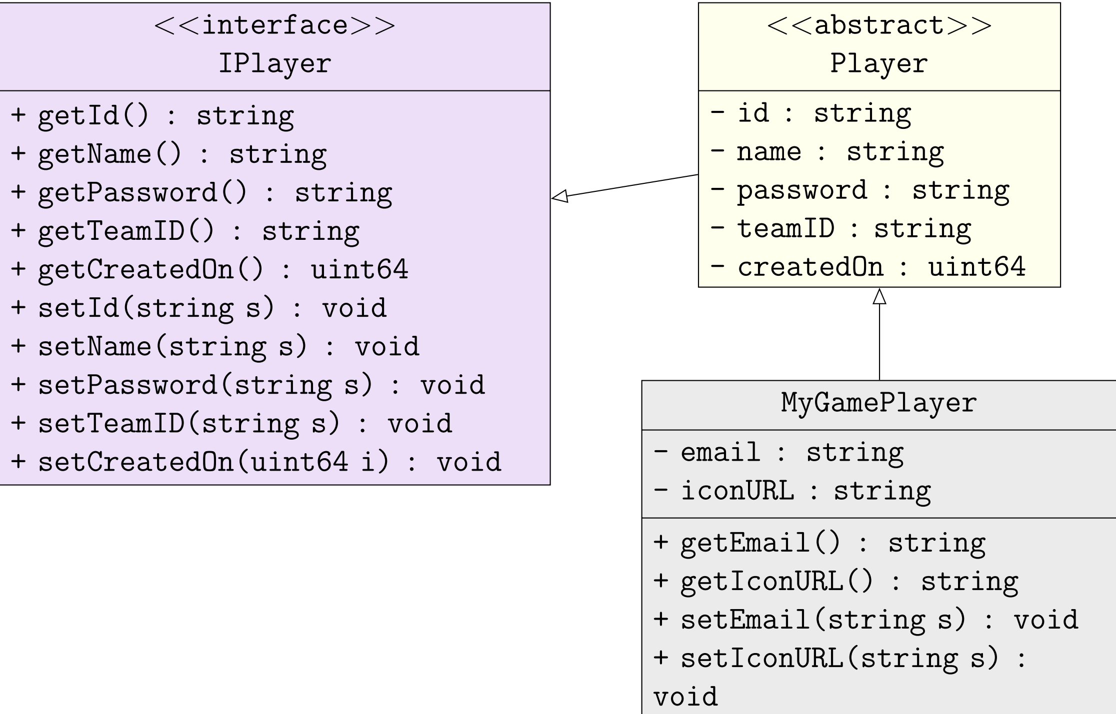


Figure 4.1: The different components of the model, illustrated using the Player type as an example.

Protocol Buffers with composition. Despite many supporting the use of composition over inheritance when possible (Bloch 2008), the logical "is a" relationship between types cannot be avoided altogether. The de-coupling of attributes and operations between these classes also enables their use in context-free situations where they may be used to perform generic operations – e.g. finding which players are not in a team. These generic operations are not game-specific, and can therefore be included at the API layer, thus increasing the framework's functionality and support. To ensure that such generic, context-free operations can be carried out without having to be implemented by game developers, this level of separation is necessary. Finally, using this approach, the model can include both default attributes and operations within the default model, while also allowing customizations at the game-specific level. In subsequent diagrams, static API-level interfaces are marked with light purple color, default abstract classes are marked with light yellow, whereas concrete, game-specific classes are marked with light gray.

Using the MP case study game presented in section 4.2.2 as initial guidance, various types can be abstracted. The following sections describe each of the default types of the model individually, explaining the motivation behind them and their potential uses. The relationships between

these types are summarized in Appendix 9.B, figure 9.4. While most of these types are derived from the MP concept, some others are motivated by other case studies that will be presented in section 5. These types can be broken down into three main categories: (a) *state-related types*, which are involved in modeling the state of a game, (b) *management types*, which help manage various processes of the game (e.g. authentication), and (c) *utility types* which either aid the development process or are used to enhance scalability or performance. Several objects in the subsequently described model are identified using a unique, string-based `id` attribute which may act as a key to retrieve them either from memory or the persistence layer. This attribute is also used to define various types of relationships between objects – similar to a foreign key or an object reference – either using a single attribute for a one-to-one relationship or a list of IDs for a one-to-many relationship.

The following sections describe various data types which make up the core of the Athlos model. Some of these types are well-known in the game development community and are therefore briefly mentioned in this section and elaborated further within Appendix 9.B. Meanwhile, other data types are motivated by the need to manage consistency, performance, and scalability in MMOG backends. The problems, motivation, and structure of these types are fully elaborated within the following sections.

Like other frameworks, Athlos identifies the *Player* (NX) as one of the core entities of a game. As an extension of the player, *Teams* (NX) may also be formed, consisting of collections of players working towards a common game objective. The use of these two types is mainly motivated by the need to model information about game *actors*. These two types are further elaborated in Appendix 9.B.

4.3.4 Worlds (NX)

Another commonly used game item is the game *world*, commonly described as a MultiServer Distributed Virtual Environment (MSDVE) (Yusen et al. 2016), Modifiable Virtual Environment (MVE) (Donkervliet et al. 2020), or Networked Virtual Environment (NVE) (Mildner et al. 2017). This data type is motivated by the need to distinguish the behavior of different

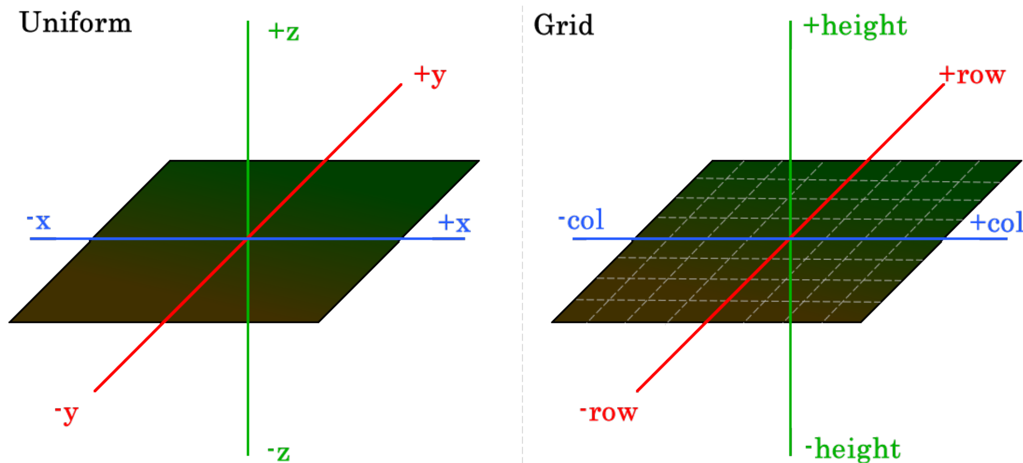


Figure 4.2: The coordinate system used for uniform and grid-based game worlds.

games based on how they represent their states, as games may have different state characteristics. In general terms, a game world is a physical domain in which *terrain* and *entities* can exist. Depending on the game, there may be multiple worlds or a single world that players can join. The worlds featured in various types of games dramatically differ from each other and impact the way the game is played. Therefore, the proposed framework must support a large variety of games and world states, and its model must distinguish among different world types based on their information requirements and how the entities within them can interact with the game state. For this reason, game worlds are categorized into two main sub-types: *uniform*, and *grid-based* worlds. The world type is also used to record the game state in games that do not necessarily feature a virtual environment. For instance, puzzle games can still use this type to record the state of a board or level. The following subsections motivate toward different types of worlds which have different characteristics.

Uniform worlds

In *uniform worlds* entities can exist in a 3D plane and are free to move to or between points within this plane. The positions of entities are recorded using a 3D coordinate system, depicted on the left part of figure 4.2, which features an east-west axis (x), a north-south axis (y), and a height axis (z).

This type of world is commonly utilized in combat, racing, and other games in which entities

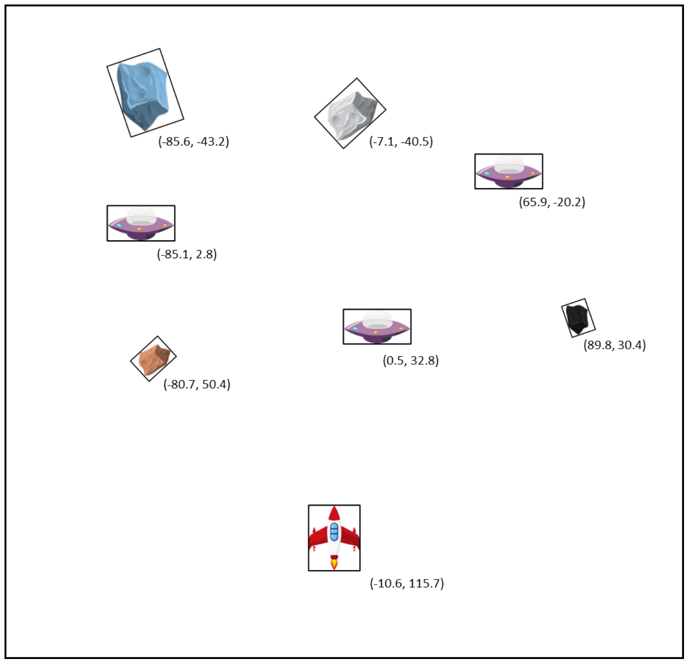


Figure 4.3: A uniform world, from a bird's-eye view.

are able to move at any location in space without being attached to a particular set of points. Figure 4.3 graphically demonstrates a uniform world, in which entities can freely move around in space without being fixed to particular points – indicated by their floating-point coordinates.

Square grid worlds

Square grid worlds are divided into a two-dimensional grid that consists of *tiles* or *cells*, similar to that being described for Minesweeper in section 3. In this type of world, any number of entities may exist on each cell depending on the game's rules and may move up, down, left, or right to the adjacent cells, provided those cells exist. Entities cannot exist outside of the grid. Cells are identified using row and column *indices* – i.e. integer numbers – with their coordinates being similar to those of uniform worlds but in a 2D plane. Row indices move in a north-south direction, whereas column indices move in an east-west direction. These types of worlds can also support 3D space using a *height* property that is utilized by entities when necessary. There are many examples of games utilizing these types of worlds, including those in turn-based strategy, puzzle games, and so on. Figure 4.4 graphically illustrates square grid worlds.

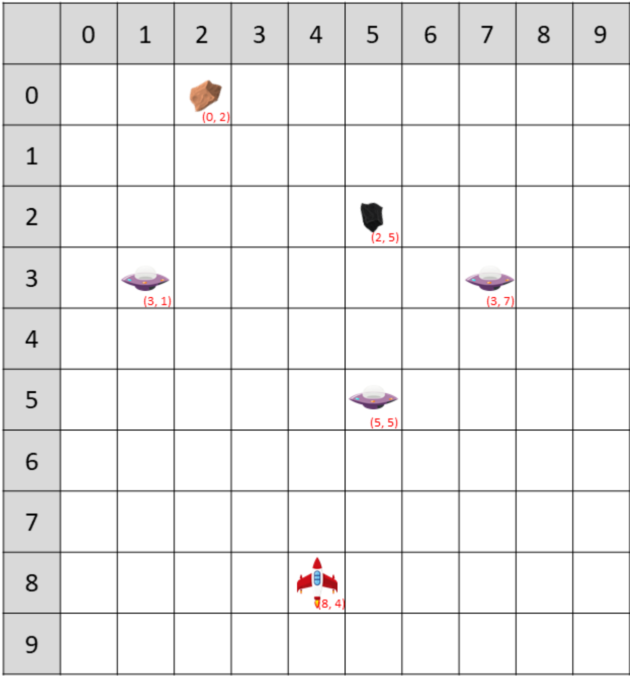


Figure 4.4: A square grid world, from a bird's-eye view.

Hexagonal grid worlds

Hexagonal grid worlds are very similar to square grid worlds but allow a wider range of movements to be made by entities. In these worlds, entities can move up, up-right, down-right, down, down-left, and up-left. This is shown graphically, in figure 4.5.

The differences between state requirements, rules, and gameplay between games necessitate the use of these different types of worlds. Furthermore, this approach introduces several advantages in terms of efficiency. For instance, if a game developer knows in advance that their game is best suited for a square-grid world, using this type of world may reduce their efforts, as the framework includes built-in support for moving entities according to the available space and coordinate system of each world in an efficient manner, which can improve the backend's performance. Additionally, it facilitates the creation of terrain, which is described in section 4.3.5. The relationships between the different types of worlds are shown in figures 4.6.

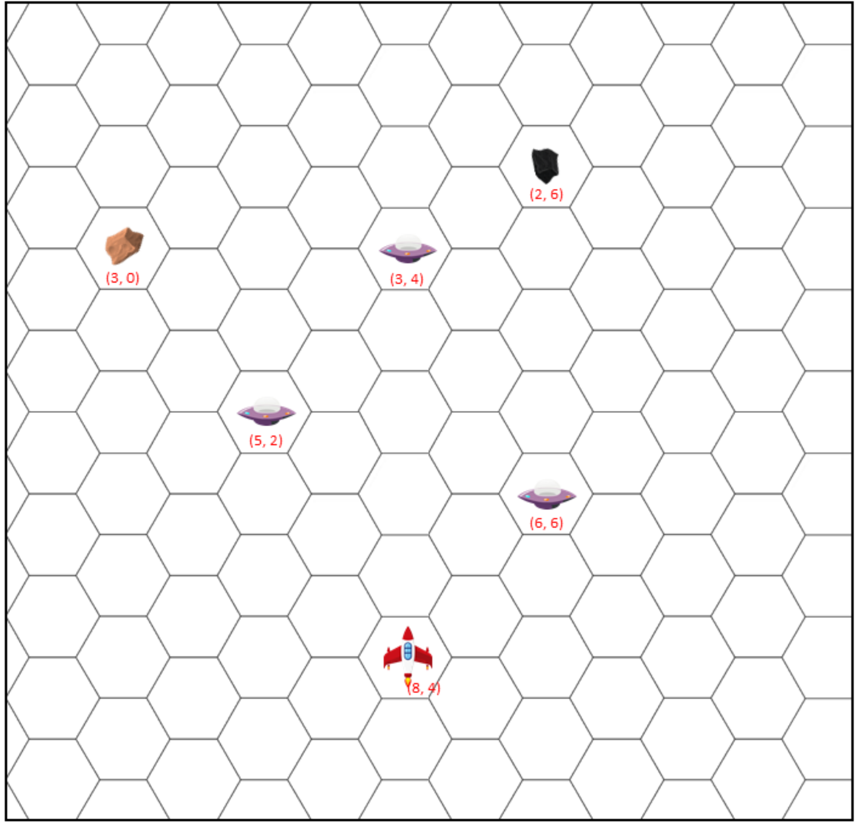


Figure 4.5: A hexagonal grid world, from a bird's-eye view.

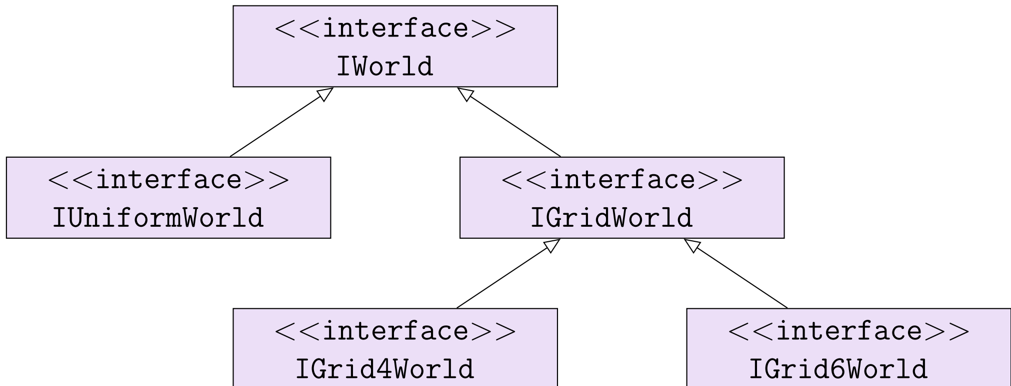


Figure 4.6: The interfaces supporting the world models.

```
<<abstract>>
World
- id : string
- name : string
- seed : sint64
- createdOn : uint64
- chunkIDs : List<string>
- heightLimit : uint32
- maxRows : sint32
- maxCols : sint32
- ownerID : string
- subscribedSessionIDs : List<string>
+ cellIsInBounds(sint32 cellRow, sint32 cellCol) : bool
+ cellIsInBounds(MatrixPosition cellPosition) : bool
+ hasChunk(string chunkID) : bool
+ addChunk(string chunkID) : void
```

Figure 4.7: The default world model.

4.3.5 Terrain

The formation of world *terrain* is also an important aspect of gameplay. While several games may indeed include terrain as part of a virtual world (e.g. a strategy game), others may require simpler states that do not require an actual terrain (e.g. a puzzle game). Within the Athlos framework, the terrain type is utilized as both geographical terrain and a simplistic game state when required. The terrain is always modeled using a matrix and row-column indices regardless of the world type. This makes it easier to model, efficiently generate, retrieve, and manage terrain states across the framework. The state of the terrain and entities that may exist within it are separated, which means that entities in a uniform world can exist on a terrain that is modeled using a grid-like structure. The motivations and reasoning behind this terrain implementation are further discussed in section 4.4.

A world will typically have geographical limits on its terrain, outside of which terrain may not be generated, and entities may not exist. To impose these restrictions several attributes are utilized within the model to indicate the maximum number of rows (`maxRows`) and columns (`maxCols`) in the terrain. Similarly, a limit can be used for the height axis (`heightLimit`). These properties can be seen in figure 4.7, which shows the model of the default world class.

Conversely, infinite worlds can also be supported by setting these attributes to negative values, thus indicating the absence of these limits. All of these restrictions are combined to force entities to only move within the legal bounds of the terrain and to limit the valid chunks that a terrain generator may create. While these restrictions are implemented for all games, individual implementations can fine-tune their own restrictions to create out-of-reach zones that encompass certain existing parts of the game's world.

Terrain cells (NX)

In the Athlos framework, the terrain is formed using *cells* and *chunks*. Only the model and relationships between these two types are discussed in this section. The motivation behind these types and further elaboration on their use are provided in section 4.4.7.

A terrain cell is an individual cell of terrain which contains the state of terrain at a particular point. The terrain's state is used as part of the game's mechanics and rules and can be used in games that do not feature virtual environments to form game levels or stages. In each game, the terrain grid which encompasses these cells may be handled and rendered differently, depending on the level of detail required.

Terrain Chunks (NX)

A terrain *chunk* is a collection of adjacent terrain cells. Chunks are used to group cells together in a single object rather than many smaller cell objects, which allows them to be efficiently generated and stored. The maximum number of cells (defined as `SIZE` within the chunk model) included in a chunk must remain constant to ensure that the terrain can be accessed reliably and scaled smoothly. However, this constant can be changed from game to game, ranging from 4×4 to 64×64 collections of cells in each chunk depending on the game. Figure 4.8 graphically demonstrates the relationship between cells and chunks. Furthermore, the `TerrainChunk` model contains several utility methods for computing chunk limits, which are used internally by the framework. These and other attributes of chunks are shown in figure 4.9. The reasoning

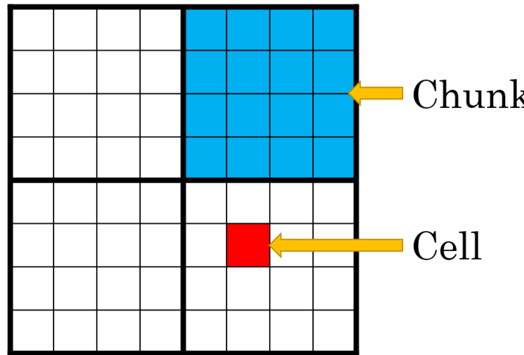


Figure 4.8: A representation of the terrain grid, demonstrating the difference between a cell (red) and a chunk (cyan).

and method behind the use of cells and chunks are discussed in section 4.4, and evaluated in section 6.

4.3.6 Terrain identifiers (NX)

The terrain chunk type described above is used to aid the creation and management of scalable terrain by omitting its full state. The chunk type is relatively heavy as it may contain the states of thousands of terrain cells. To allow developers to efficiently manage terrain and identify terrain elements, *terrain identifier* objects are used. These act as intermediary objects, modeling various properties of terrain chunks, while omitting their states. This helps reduce loading times, thus alleviating the backend from extra latency during a variety of operations. Terrain identifiers are further discussed in section 4.4.7, and evaluated in section 6.

4.3.7 Entities (X)

An *entity* is an item or object that exists inside a world. Some entities may belong to and are controlled by a player. Entities have a position, direction, and the potential to change these attributes. These attributes and the ability of an entity to change them are dependent on the type of world it exists in, and therefore the entity model must be adjusted accordingly for each game – a use case of dynamic modeling requirements. For instance, an entity that can exist in a uniform-type world needs to have a `GeoPosition` type for its position, whereas an entity

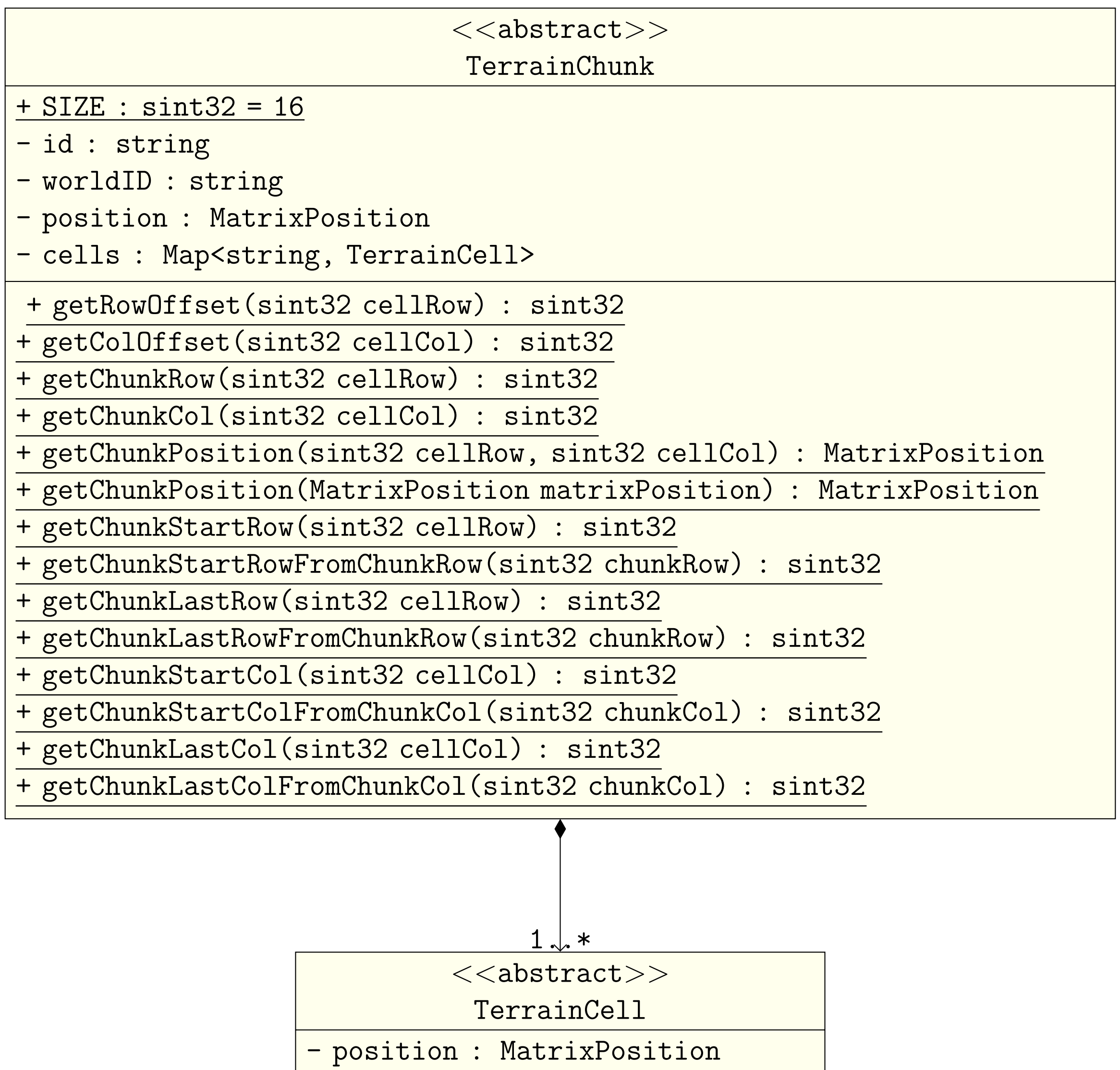


Figure 4.9: The model and relationships between cells and chunks, used to represent terrain

in a grid-based world needs to have a `MatrixPosition`. These two positioning types are described in section 9.B.3. Each entity also has an area of interest, which is the distance at which it can perceive game events and affect the game state. Entities can also be categorized into three groups based on their state's ability to change:

- **Static entities** are those of which the state cannot be changed during gameplay. This mainly includes items that cannot be deleted or depleted. Known use cases of such static entities are spawnpoints[5] and waypoints[6] (Ballabio & Loiacono 2019).
- **Stationary entities** are entities of which the state can change, but they cannot be moved or rotated. For example, a tree or a flower may be described as a stationary entity because its state can be transitioned – i.e. the flower or tree can grow – but its position cannot be changed.
- **Dynamic entities** are those of which the state can be changed fully.

These categories are abstract and their only purpose is to identify the potential of entities to change their state. While they are not concretely differentiated within the framework itself, they can provide developers with guidelines on how to implement specific MMOG backends. This categorization can be used to improve the efficiency of the backend by minimizing the number of operations required during each game cycle. As an example, this can be achieved by excluding certain types of entities like static entities, of which the states are known to remain constant, from the processing cycle. Removing these types of entities from the processing pipeline can alleviate the backend, improve performance, and lead to a reduction in bandwidth use if these entities are also excluded from state updates.

Information about where entities are located within a world and how they may behave is modeled differently based on the type of world they exist in. To manage various operations, the Athlos framework uses *positioning*, *movement*, *direction*, and *rotation* types. These are conditionally added to the model of each entity based on the world type and allow developers

[5] A spawnpoint is the location where an entity is placed when it is created.
[6] A waypoint is a point that an entity must pass through to complete a game objective.

```
<<abstract>>
Entity
- id : string
- worldID : string
- playerID : string
- areaOfInterest : float
- position : Any position
- direction : Any direction
```

Figure 4.10: The default entity model.

to manage various attributes of these entities. Descriptions of these spatial types and constants can be found in Appendix 9.B.

Within Athlos, *Events* (NX) are also used to represent information about an event that takes place at a certain time during gameplay and may involve one or more entities. Events are managed through the Event Manager – a mechanism that allows their creation and substantiates each event. As an extension to events, *Actions* are also identified as events that are initiated by a certain player, at a specific location and time during the game. The models for events and actions and more information about these two types can be found in Appendix 9.B.

4.3.8 Partial states (NX)

A *partial state* is a type that enables the modeling, storage, and communication of a specific part of a world's state. Partial states can include entities and terrain, or other optional game-specific components, and may be relative to a specific player's perspective. By using partial states it is possible to limit the size of the state being accessed at any single time by a player, or the backend itself, which can be beneficial to performance and scalability. The very large scales seen in MMOG worlds make it impossible to retrieve the entire state of the world continuously without suffering from significant reductions in performance. Using partial states, it is possible to allow backends to (a) carry out operations more efficiently, (b) retrieve only the necessary information, and (c) reduce bandwidth use, which may result in a better economy over time.

For state updates sent to the players, partial states are composed based on the combined area of

interest of the player's entities. For instance, if a player controls two entities with two different AoIs, they will be able to receive state updates occurring within the AoIs of both entities. Depending on the AoI of players and various other game-specific factors, each partial state may contain different data and have a different size.

4.3.9 State updates (NX)

State updates model changes in the existing state of the game and must be sent to the clients by the backend continuously. They are received *after an initial partial state has already been received by the clients* and are meant to update an even smaller subset of the world's state. This is achieved by only communicating updates about the updated entities or terrain, rather than the entire partial state. For example, when a player P moves their avatar or entity within the world, it would be necessary to update the state of all other players in the world, so that they can perceive P's movement. This would entail re-sending an entire partial state, containing all of the data that these players can perceive, rather than just the movement of P's avatar. This is rather inefficient, as irrelevant information is being sent along with the useful part of the update. State updates solve this problem by sending state "deltas" or "diffs", which are small, usually incremental changes in the state that do not cause the system to query and transmit irrelevant information.

4.3.10 Other types

Finally, the model also includes several management and service definition types. For example, the *game session* is used to record information about client interactions with the MMOG backend. As an extension of game sessions, *world sessions* are also used to track player progress and actions within the context of a certain game world. Further to these, Athlos also uses the *Service* data type to define various types of services related to games – including some of which can be generated by default. The models of these services are based on two other subsequent types, *requests* and *responses*, which are used by services to define their expected input and

output data sets. Services and sessions are further discussed in Appendix 9.B.

4.3.11 Games and rules

Even though a *Game* type does not exist within the model itself, it can be used to describe the set of subsequent data types and relationships that define a game – like its players, teams, and worlds. A game type can be optionally implemented by developers if needed when there is a need to persist globally available information which is not related to a particular world. A common use case of such a type would be to model game policies that may affect gameplay in all worlds or to record other information for analytics or logging.

Game *rules* are perhaps one of the most important aspects of online games. Rules can be found in any game, dictate what actions are possible, and how these actions can affect the state of the game. Each game features its own set of rules, which is largely what makes each game unique. Thus, it can be argued that a generalization of the rules in all MMOGs, or even a small subset of MMOGs would not be practical or desirable given the infinite possibilities of game design. To the extent of this author's knowledge, games are made unique by featuring different rules and attempting to abstract this element may result in inefficiencies during development.

In addition to the items described in the previous sections, there are possibilities to abstract more items which are specific to particular game genres. For instance, strategy games may include various types of *resources* within their worlds, which can be collected either by harvesting them (*depletable resources)* or by simply accessing them (*infinite resources*). Similarly, many types of games may employ a *virtual game economy*, allowing players to gather *virtual currency* to purchase items. Lastly, an extended game model could also support data types for handling *social interactions*, creating *leaderboards*, implementing *code execution* for code-based games, and much more. Even though this thesis presents a limited subset of what is possible, such extensions are entirely within reach by utilizing the proposed approach and could be implemented as plugins, or as part of the framework itself in the future.

4.4 Methods

Some of the most important challenges in developing MMOG backends are related to the introduction of support for scaling, maintenance, and evaluation at the software level. MMOG backends are usually developed as monolithic applications, following rigid software architectures that result in tightly-coupled code. This may allow MMOG backends to be developed faster and with less effort because there are fewer components to create and design decisions to make. Problems, however, arise when there is a need to make adjustments to the code of these backends, either to maintain them by fixing bugs, or by adding new features. Especially for the latter, making even simple changes often requires a software disassembly which is a time-consuming, cumbersome process. Therefore, developers may initially spend less time writing simpler, monolithic code that intertwines the functionality of different components, only to find themselves having to deconstruct it later to make simple additions. This problem is further exacerbated when there is a need to change the technology stack. In such cases, game developers may find it impossible to modify their game to run using different tools or technologies without making extensive changes because game logic and components are entangled with the code written to utilize the underlying technology.

The methodology presented in this thesis makes the case for using a modular approach. This first separates game components from the technology stack as much as possible and subsequently separates game components themselves to form a *modular software architecture*. It is assumed that MMOG backends can greatly benefit by moving away from monolithic structures and towards a modular architecture. This can provide several advantages such as (a) facilitating code re-usability and thereby reducing development effort, (b) enhancing code structure, which may lead to improved maintainability and expandability, and (c) allowing software components to be interchangeable, and be swapped in and out as required, thus promoting the evaluation of different technologies before deployment.

Throughout this section novel methods are described, with which MMOG backends can be developed as modular applications with an emphasis on performance and scalability. The process of development is broken down into several steps, as shown in figure 4.11. At the start

of the process, game developers conceptualize their game by establishing its core elements: the basic entities, constructs, rules, and features it must provide. This step is mostly theoretical and is meant to help developers envision how the game will be played, determine the cost of development, and whether the concept is feasible. As with other software engineering projects, online games must also pass through a phase of design, during which these core elements are concretely defined and relationships between them are made. This entails the definition of data types and their attributes, class hierarchies, considerations for persisting data, the game's activity and event flow, the design of the game's user interface, how the rules of the game will dictate the outcome of actions, and more. While some of these steps – e.g. rules – are game-specific, some others can be generalized to streamline the development process. To help with the process of designing a game, a *game editor* is used to create *game definitions*, which describe the core elements of the game being developed. Once the design step is over, the game definition is passed to a *code generator*, which parses it and automatically generates the project's boilerplate code using the selected programming language and runtime environments. Developers can then start the implementation of game-specific elements and make customizations to the generated code. When the development of a certain set of features is completed, developers can choose to first test, and then deploy their backend, or go back to the editor to make changes or add more features, repeating the process of design, generation, and implementation.

This development approach follows an agile software development paradigm which enables the employment of several techniques like rapid prototyping, incremental, and iterative development. Using these techniques, game developers can achieve a lower time to market, maintain control over the characteristics and features of a game, and reduce wasted effort and time through the abstractions provided by the proposed tools. This approach can also be used in a hybrid *Software Development Life Cycle* (SDLC) model, where developers can first design their MMOG through the editor, generate a project, implement it, and then deploy it in a single iteration. However, experience shows that the needs of game development necessitate a more agile approach because features tend to continuously evolve to enhance the game experience. Nevertheless, the proposed approach is flexible and allows a variety of software development methodologies to be employed. This process is partly based on the ideas of Model Driven

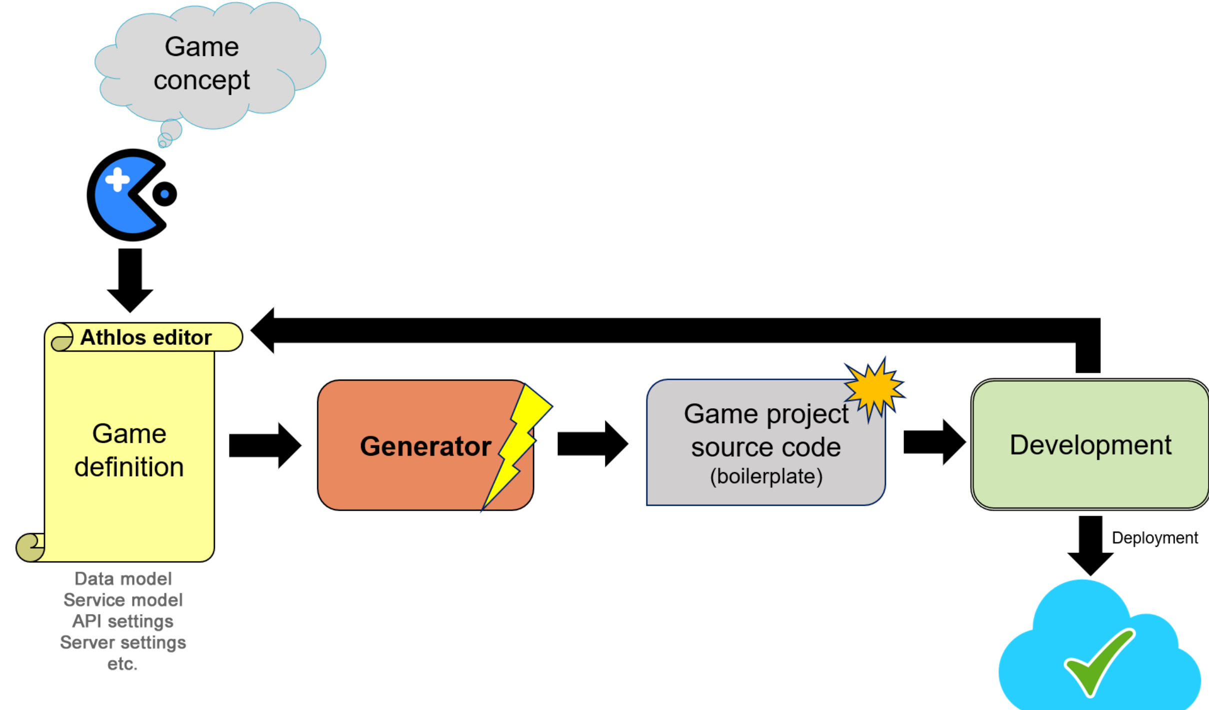


Figure 4.11: The process of defining and generating a new project.

Development (MDD), originally developed as "*a promising approach to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively*" (Schmidt 2006, Hailpern & Tarr 2006).

4.4.1 Game definitions

Before discussing the details of various aspects, it is important to understand the backbone of the proposed approach and how the model described in the previous section is leveraged to develop MMOG backends. The first step in doing so is to acknowledge that there is an abundance of methods and tools that can be used because each game is unique. However, as seen in section 4.3, it is possible to identify a number of similarities. These similarities are not limited to the modeling requirements of MMOGs but also extend to the development processes and methods used. The intention of the proposed approach is to leverage these similarities to expedite development as much as possible. To accomplish this task, the Athlos framework is centered around game definitions. A *game definition* is a fully abstract, technology and approach-independent definition of an MMOG and its related information and configurations.

Firstly, game definitions record basic information about a game, such as its name, the type of worlds included in it, the intended method of deployment, and so on. Each definition can be described as a different project that can be realized into a fully operational MMOG backend. Definitions contain a model of the game (i.e. the *game-specific model*), which incorporates data types, their attributes, and the relationships between them. The game-specific model is an extension of the Athlos model (or the default model) which is presented in section 4.3. Within game definitions, the default Athlos types can be extended to include additional attributes, and new, game-specific types can be created. In addition to the model, game definitions also include service models and their subsequent request and response types. *Service models* are approach-independent service declarations that specify the services that make up the game's API. Each of these services makes use of specific *request* and *response* models that must also be defined for the services to be declared. The request and response models themselves define the types of data expected as input and output from the services. Apart from these models, game definitions also include various configurations related to the game, such as:

- Selected runtime environments for both the client and server or serverless backend.
- Project information like abbreviations, time of creation, last update time, etc.
- Server configurations such as names, port numbers, API URIs, etc.
- Default values that may be adjusted for each game, such as chunk sizes, camera ranges, etc.
- Version/revision control information like version numbers and code generation dates and times.

By using game definitions, Athlos attempts to assimilate all information related to a specific game project. This single information model enables the use of various tools which can be used to create and manage these definitions, as well as to automatically generate boilerplate code. Game definitions can be created and managed using a software tool called the Athlos Game Editor, which is presented in section 4.5.2. The game editor stores these definitions in files,

which can be opened for editing at a later time, shared with collaborators, or hosted online. The ability to work collaboratively on such projects must be included as game development often involves teams of programmers. The inclusion of all game-related information within a single project also makes it possible to offer a higher level of customizability. Developers can leverage various abstractions, but also create fully-customized models that can support a large set of unique game designs. Finally, game definitions can subsequently be used by the code generation tools such as those presented in section 4.5.3 to expedite the development process by automatically generating boilerplate code.

4.4.2 Infrastructure

The first step towards enabling these propositions and utilizing a modular software structure is to differentiate the many development approaches that can be used. These methodologies are dependent on various aspects, the most prominent of which is infrastructure. Athlos categorizes infrastructure in terms of deployment on either *dedicated or IaaS* environments, and *serverless* environments. The distinction between these two types is made based on how MMOG backends can be engineered to run on these environments and the features that these environments can offer. The IaaS and dedicated approaches share many similarities in terms of development as they both provide full control over computing resources, regardless of them being physical or virtual. The customization opportunities provided by such infrastructures are quite extensive because developers are free to utilize any development environment, operating system, or programming language they choose. The dedicated/IaaS approach allows developers to create MMOG backends with fewer overheads than those developed using serverless, which means that they can achieve lower latency. In terms of features, this approach also allows MMOG backends to leverage unbounded execution time for their services, which allows them to establish bi-directional communication links with the clients. Bi-directional communication is especially important for MMOG backends, as state updates can be easily transmitted without utilizing third-party services. Another important feature of MMOGs is the ability to run background tasks or events, in concurrence with the normal request-response cycle. Dedicated and IaaS

infrastructures allow applications to run concurrent threads of execution to support this feature. Despite these advantages, and perhaps many more, these environments are inherently non-elastic. This means that to attain the necessary scalability for a commercial MMOG backend, developers must resort to vertical scaling, which is still limited in its potential, or utilize containerization systems like Docker (Vähä 2017). These systems are available for experimentation, or for commercial use in public clouds, using services like Amazon's Elastic Container Service (ECS), Google's Kubernetes, or Agones (Lundgren 2021).

On the other hand, serverless environments are limited in terms of these features. Popular examples like Google's App Engine and Cloud Functions, and Amazon's Lambda functions limit the ability of a developer to run code concurrently in background tasks. Instead, developers have to utilize additional services like Google's Cloud Tasks or Amazon's Step Functions for asynchronous execution. Moreover, only a limited set of serverless products like Google's App Engine Flexible or AWS's Lambda functions are known to provide an API for bi-directional communications, mostly using WebSocket technology. In such cases, developers may have to resort to third-party services, such as those provided by Ably (Ably 2019), use specific serverless environments that support bi-directional communications, or simply use *polling* methods where the game's requirements permit it. Unlike dedicated or IaaS approaches where developers can utilize any development approach, serverless environments are designed to use HTTP endpoints to make services available through web container technologies like Java Servlets and WebContainers in Node.js. The use of HTTP adds an additional layer of overhead to those already in place by TCP/IP (Transmission Control Protocol/Internet Protocol) or UDP (User Datagram Protocol), which may induce additional latency to services. Despite that, the more recent version of HTTP/2 provides several performance-improvement functions such as multiplexing, which can improve performance. Secondly, good service design coupled with compression methods like bit-packing can be used to reduce these effects by a substantial amount. In addition to these, the adoption of the cutting-edge HTTP/3 and QUIC protocols is expected to dramatically improve performance for serverless backends in the coming years (Carlucci et al. 2015).

Despite these overheads, the serverless approach allows MMOG services to become stateless,

and thus be more scalable. This can be leveraged to an unprecedented level and can make MMOG backends elastic, thus reducing the need to manage scalability to very simple configurations. Consequently, the stateless execution of services coupled with a more streamlined development environment can enable the use of various abstractions to make the software engineering process more efficient – ultimately reducing the time to market of MMOGs and improving their quality.

The primary objective of this thesis with regard to infrastructure is the exploration of serverless computing for developing MMOG backends. Even though the presented framework also supports the IaaS/dedicated approach, this is studied to a lesser extent. Understandably, these two approaches are very different and therefore developers must take different paths when developing MMOG backends on either of the two. MMOG backends designed using Athlos can use either approach, and it is possible to change the infrastructure approach being used while a project is in development. However, this is discouraged as it may complicate the development process. Instead, developers are encouraged to select between these approaches early on in their project's lifecycle rather than later, with the aim of simplifying development. In cases where it is necessary, they can still adjust their project to utilize a different approach by first ensuring that the game's services are valid for that approach, and secondly by making manual adjustments to their code to enable the utilization of these services through different protocols.

As discussed in section 2.8.1, each of the two approaches is advantageous for different games, depending on how their gameplay affects various aspects. The IaaS/dedicated infrastructure is more suitable for games that require low latency but do not feature persistent, scalable worlds or do not have to support a large number of concurrent players – for instance MMOFPS or racing games. On the other hand, games that do not require very low latency, such as MMORPGs and MMORTSs, may benefit from using the serverless approach to provide scalable worlds to larger numbers of concurrent players. On its end, the Athlos framework supports development for either of the two types of games using a unified code base. The *Athlos API* is a set of fully abstract models that are divided into four independent namespaces:

- The **core**, which contains common models required for development across the framework.

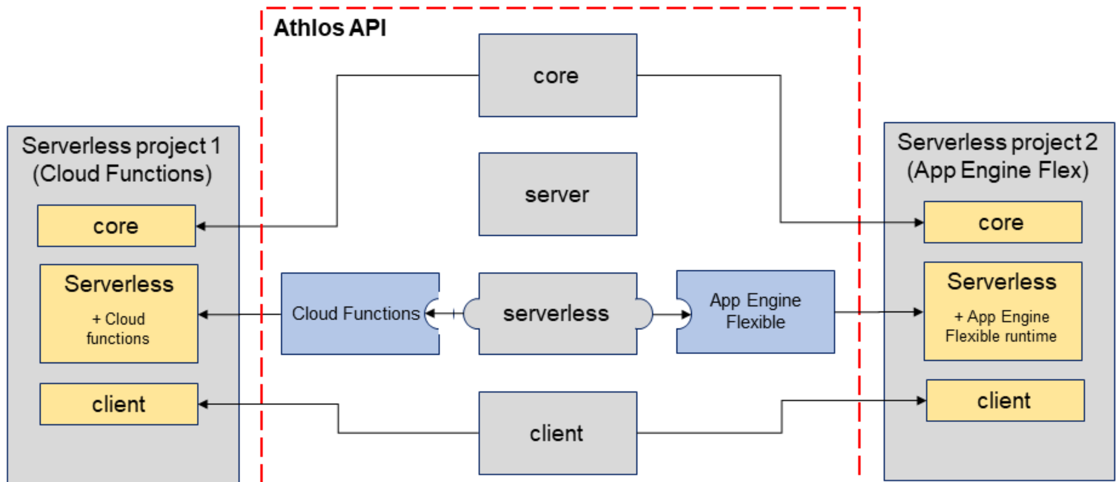


Figure 4.12: The Athlos API, with pluggable serverless components.

- The **server** containing the models for creating and managing servers in the IaaS/Dedicated approach.
- **serverless**, which contains models for creating and managing serverless backends.
- The **client**, containing models that define the functionality of MMOG clients.

The use of this structure is also defined in each MMOG backend's code, regardless of the infrastructure used. Currently, Athlos enables the development of serverless MMOG backends using Google's App Engine Standard and Flexible versions and using Google's Cloud Functions. While all of these approaches are different, the framework provides the necessary level of abstraction to deal with all of them and possibly additional approaches in the future, by utilizing pluggable software components and *dependency injection*, as shown in figure 4.12.

4.4.3 Architecture

The predominant network architecture used in MMOG backends is the client-server model. This type of architecture presents a plethora of advantages for MMOG backends, including better control over resources, tighter security, and reliability. While the related works also study the possibilities of deploying MMOG backends using hybrid architectures, this requires resources that may not be available to everyone trying to develop such systems. Alternatively,

P2P systems could potentially provide a more scalable option but are harder to implement and tend to make MMOG backends more vulnerable to malicious attacks. The client-server model itself has a single point of failure – the server – which makes it less robust, and its centralized control offers lower performance than that of P2P. However, when coupled with commodity clouds these problems are mostly eliminated due to the high availability and scalability they offer. While the client-server model itself involves only two major components –client and server– many have proposed the use of additional components that expand this basic model.

Inspired by other research (Chu 2008), a more advanced version of this architecture is proposed, which features components specifically targeting MMOG backends. As presented in figure 4.13, this architecture involves the use of a *client*. The client is a software application that can run on the player's device — a PC, smartphone, tablet, etc. Its responsibilities, in order, are to:

1. Receive input using from the player and convert it into a corresponding *action*.
2. Optionally enforce a subset of the game's rules based on the *local state*[7], such as physics, collision detection, and so on.
3. Update the player's local state.
4. Transmit the *intended changes*[8] within the context of an action to the server.
5. Receive and present an updated view of the game's global state to the player.

Service-oriented architecture

In this Service-Oriented Architecture (SOA) paradigm, the client does not communicate directly with the backend's runtime, instead relying on a communication medium to provide access to this component. This communication medium is the game's Application Programming Interface (*Game API*), which is responsible for exposing game functionality to the clients as service

[7]A local state is defined as the game state which is temporarily stored on each client device and is relative to the player.

[8]A small subset of the game state which defines which items must be updated and their new state after the action takes place.

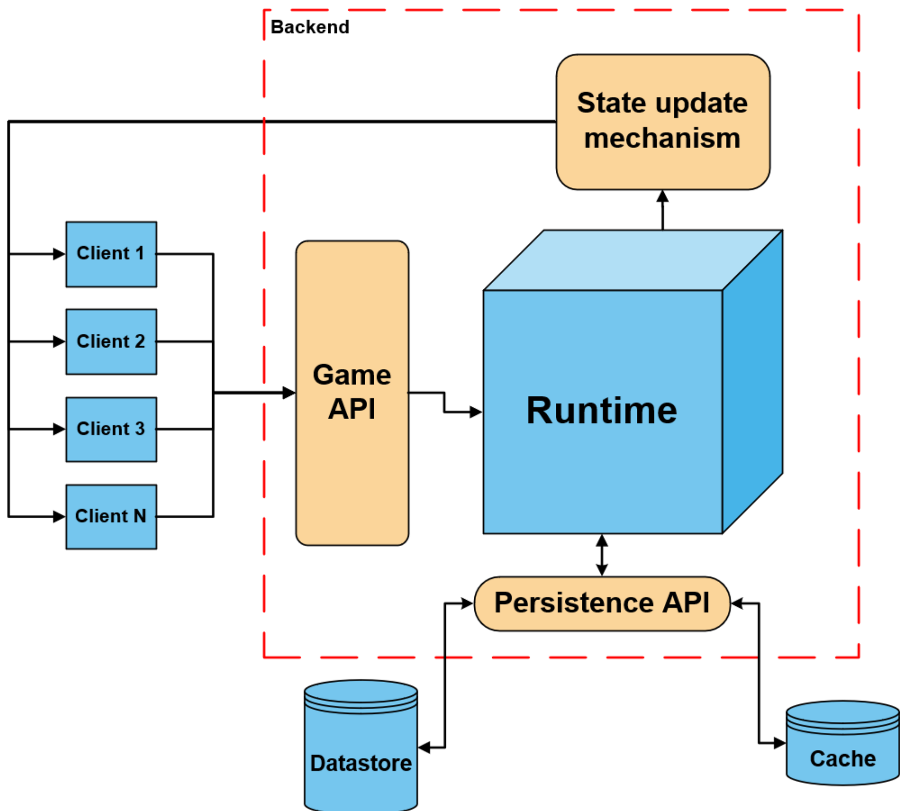


Figure 4.13: The proposed Athlos architecture.

endpoints. This enables the clients to communicate their intended actions and any associated data to the backend as *requests*, which are received, decoded, validated, and then sent to the *runtime* for execution. Upon receiving a validated request, the runtime will enforce the game's rules and execute the action accordingly. In this architecture, there are several possibilities for the validation and execution of rules. It is acknowledged that some games may perform resource-intensive operations to enforce their rules — such as the calculation of complex object geometry, collision detection, ray-casting, and more. While Athlos allows developers to choose where these operations are executed, it is proposed that all complex operations involving local state data be offloaded to client devices to alleviate the backend. Despite that, all local rule enforcements must never be able to affect the global game state and thus enforce *client passivity*[9].

Before an action is executed, the runtime must acquire a part of the global state to validate that the action is valid based on the game's rules. The global state may be persisted on a *relational database server*, *cloud datastore*, or *cache*, and can be retrieved by the backend through queries.

[9]A passive client is defined as a client that can simply 'observe' the game state and can only issue actions that must be validated by the game API before affecting the global state.

Upon receiving the state of the game, the runtime must make the necessary modifications to reflect the action intended by the player. For instance, if the player throws a grenade, this may alter the state of the terrain and entities within a certain radius when the grenade explodes. Such *state modifications* are a critical process in MMOG backends because they are carried out simultaneously for large numbers of clients, and often in rapid succession. Therefore, it is important to optimize them as much as possible to improve the QoE perceived by the players. Upon making these modifications the runtime must also update the state stored in the cache or database, by running an update operation.

Another improvement to the QoE can be made through the use of low-latency cache systems which offer much higher operational speeds than their database counterparts. It is argued that datastores, which tend to offer *weaker consistency* without severely impacting performance, can be complemented by distributed caching systems. These can greatly benefit MMOG backends by offering *strong consistency* with a lower impact on performance. The combination of these two types of systems can offer both strong and weak consistency while ensuring that performance does not degrade. For example, distributed caches can be used to efficiently access the game state with strong consistency, while databases or datastores can be used to store other data by executing background tasks in relatively large intervals. It is therefore argued that MMOG backends can benefit from combining these two types of persistence systems.

Updates made to the global state after an action must also be communicated back to the clients. However, the large numbers of players in MMOGs make the process of updating the state of all clients every time an action occurs very costly. Based on experience gathered from the feasibility study, it is argued that this may be a weak point in the architecture of an MMOG backend, greatly hindering performance and QoE. Motivated by the importance of state updates, as well as their resource-intensive nature, an abstract yet customizable *state-update mechanism* is proposed as a component of the architecture that may be used to efficiently carry out these operations. The use of this mechanism is elaborated in section 4.4.7.

The architecture presented in figure 4.13 offers more possibilities for abstraction. Game APIs and the services out of which they are composed can be divided into several components. As

seen in figure 4.14, the game API can be divided into three different sub-components. When a request is received, the *service mapper* is responsible for mapping it to a service. This component is technology-dependent as it maps a service container to a specific URI which is called by the client. Once this mapping is made, the related *service container* is called to respond to the request. Like the service mapper, the service container is also a technology-dependent component, — e.g. an HTTP function. The standard way to implement a service is to include its logic within the service container and have the response sent back directly. However, this ties the service logic to its container and thus the underlying technology. To provide a technology-independent way of defining service logic the *service* sub-component is introduced, into which service logic can be transferred. This works by having the service container acting as a *mediator* rather than a provider of the service itself, decoding data and forming a context before passing the request to the actual service for validation, and subsequently to the runtime for execution. Using this approach, it is possible to implement services in a technology-agnostic way, while service containers and mappers can be automatically generated based on the specific technology selected. In addition to simplifying the development process, using this approach makes it possible to substitute the underlying technology without having to make changes to the service component, which enables more comprehensive experimentation, testing, and evaluation. It also enforces the use of RESTful (Representational State Transfer) architectural constraints up to a certain extent. As defined in REST, this paradigm follows a client-server architecture made up of multiple layers and involves a hierarchical structure with many types of servers. Communications between these components are stateless, and requests are separated and disjoint, while the use of a cache streamlines interactions between them. In addition, resources and actions are available through a set of endpoints which are made available through URIs following the noun/id or noun/verb pattern defined in the REST specification. The inner workings of services are further discussed in section 4.4.6.

Persistence API

The persistence layer is another major point in the architecture where abstraction can be introduced. Traditionally, the persistence layer is directly linked to the technologies utilized to

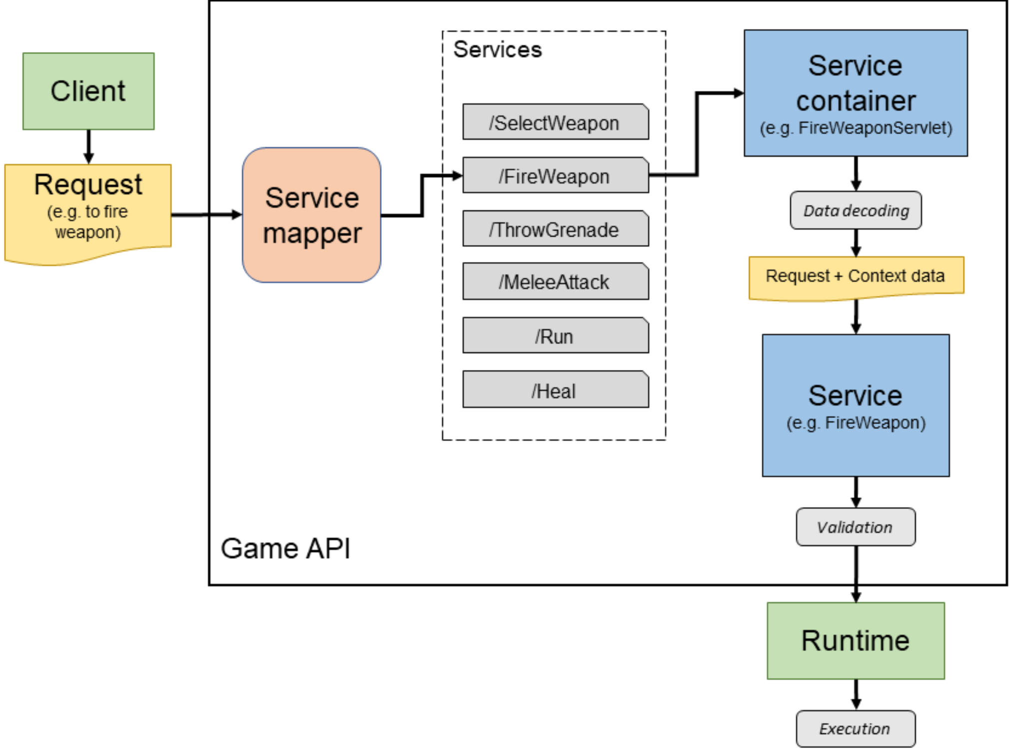


Figure 4.14: A closer look at the Game API component.

enable persistence. In such cases, developers create game services that contain code to connect to the database, retrieve information using a specific approach (e.g., queries using SQL), and then retrieve data using a specific data set format. It is argued that the combination of game logic and persistence logic increases the complexity of the code and intrinsically ties them together. As a result of this merge, the code becomes significantly less maintainable.

The architecture presented in this section introduces an additional architectural component to the backend called the *Persistence API*. This component serves the purpose of de-coupling game logic from persistence logic, with the purpose of increasing project maintainability and code reusability. Figure 4.15 shows the software sub-components included within the persistence API. This component makes use of the *Database Access Object* (DAO) design pattern to separate game logic from the data layer. Firstly, a connection is established to the database or cache through the *database connector*. This is a technology-specific component that is responsible for connecting to one or more databases or cache servers. While being considered a sub-component of the persistence API, in reality, the database connector may be a completely disjoint component that instantiates the objects required to achieve a database connec-

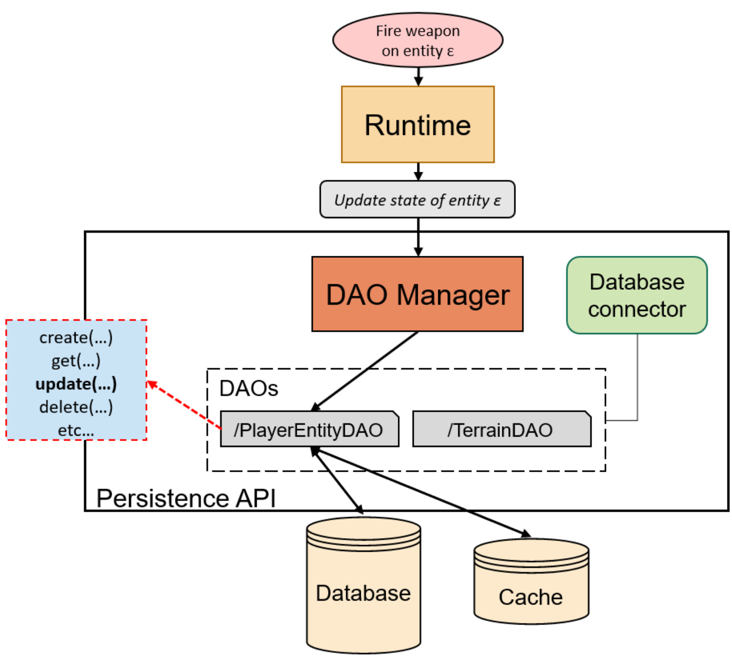


Figure 4.15: A closer look at the Persistence API component.

tion. These connections are subsequently utilized to execute database operations by DAOs. These are game-specific components that are generated automatically based on a game's model and utilize the connections made by the database connector. DAOs may connect to multiple databases and caches simultaneously to perform various types of database operations, which are further discussed in section 4.4.4. Encapsulating the use of DAOs is the *DAO Manager* which instantiates them and allows the runtime to easily access these sub-components. Using this structure, database management operations and data operations themselves can be fully managed within the persistence API. Game services can make simple, one-liner calls to initiate these operations through a helper class called the *Database (DB) manager*, significantly reducing the complexity and size of the code and de-coupling data management from game logic.

Event mechanism

This service-oriented architecture coupled with the request-response model can serve the needs of many use cases found in MMOG backends. To make use of these concepts in cloud environments, developers often create *microservices*, which are fine-grained, de-coupled SOA imple-

mentations. In systems employing the SOA paradigm, the client is responsible for initiating an interaction with the system. While this has the potential to serve a large variety of use cases, it does not provide a way to execute *background events* – i.e. events that occur in the background without the explicit involvement of the user, or in this particular case, the player.

To enable MMOG backends to schedule, manage, and execute background events, an abstract *event mechanism* is proposed which can then be implemented for specific games and cloud environments. This works by first defining the Event type, initially presented in section 9.B.4. An event can be instantiated in memory and subsequently persisted in a cache or database. Events are directly associated with a world, have a specific execution time, and must define their execution logic – i.e. what will be done when such an event occurs. Events are typically categorized by their state. Newly-created events that are scheduled are considered `PENDING`. Pending events can be `CANCELLED` before their execution or left to be executed, in which case their state is changed to `COMPLETED`. Controlling these events is the *Event Manager*, which defines an interface for managing events. This includes various functions, such as retrieving all events, those which are still pending, scheduling or canceling an event, as well as the behavior for handling events in the backend. The event mechanism relies on the game's runtime logic to instantiate an event. Events can be scheduled as one-time or recurring occurrences depending on their nature and the game's rules. To schedule an instantiated event, the runtime must communicate with the event manager, calling its `scheduleEvent()` method to schedule the event. This entails the creation of the event in the database, through the persistence layer – an action that must be explicitly programmed by the developer depending on the persistence option being used. The process of creating an event is illustrated using solid lines in figure 4.16. Once events are scheduled they must be 'handled' by deciding which events to execute depending on the system's time. In each game, the event manager implements a method called `handleEvents()`, which includes a default approach to handling events, as shown in pseudocode in 4.16. This method can be customized in each game to include additional logic. It must also be scheduled by the runtime to run in intervals, so that events can be first retrieved, then checked, and executed once their execution time is reached. This process is illustrated in figure 4.16 using dashed lines.

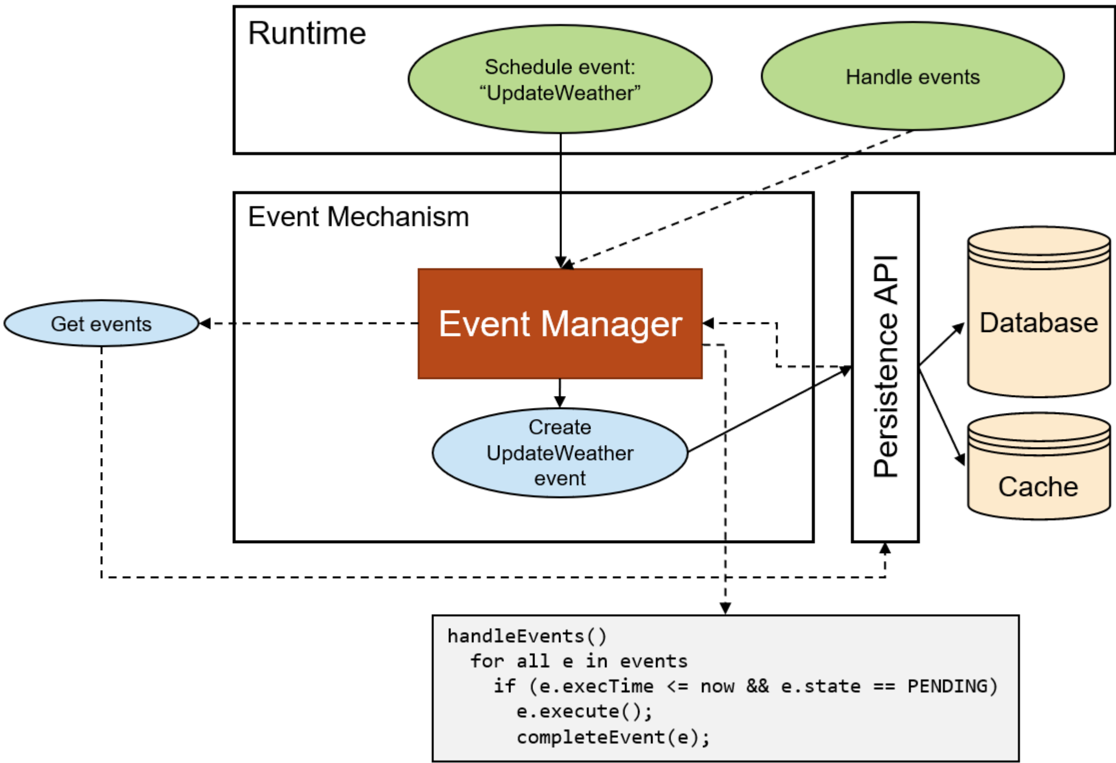


Figure 4.16: The structure of the event mechanism component, used to schedule events.

In public cloud, horizontally scalable IaaS/Dedicated systems or serverless systems, this can be achieved through the use of a cloud-based task scheduling service, such as Google's Cloud Tasks, or Amazon's Asynchronous Actions. For horizontally-scalable IaaS/Dedicated backends, this can also be achieved by having each instance repeatedly execute `handleEvents()` in intervals in the background, and using *distributed atomic locks* to ensure that no event is executed twice. In other cases where the system is not horizontally scalable, such as single-instance IaaS/Dedicated environments, this can be achieved just by having this method executed in intervals. After an event is executed, the `completeEvent()` method is called to 'complete' the event by either setting its state to `COMPLETED` or deleting it altogether, depending on the game's functionality. Alternatively, events can be canceled before their execution through the `cancelEvent()` method.

4.4.4 Persistence

In section 4.4.3, the persistence API was presented as a component of the Athlos architecture. The use of this component makes it possible to separate the presentation and logic layers from the data layer, enhances code readability and maintainability, and also provides better insights regarding performance. At the core of this approach is the Database Access Object pattern, which provides a common API to manage database records or objects. DAOs are automatically generated based on the model defined in the game definition and through the use of a special meta-attribute called `DAOPolicy`. This meta-attribute exists in all type definitions and determines how each type must be handled in terms of data persistence. The Athlos framework defines three different DAO policies, which are based on how many instances of a type can exist in the database:

- The `NONE` policy disables types from being stored in the database – i.e. the type cannot have any of its data stored in the database.
- The `UNIQUE` policy is used for types that may not have multiple instances — similar to the *singleton* pattern. In such types, only one instance may exist in the database, even though multiple instances can still be created in memory. A use case of this policy is when a game must only have a single world for players to join. In such a scenario, it is possible to leverage this policy to enforce this rule by limiting the instances of worlds to just one.
- Conversely, the `MULTIPLE` policy enables types to have multiple instances. This can include items like sessions, players, and more.

Based on the type of policy selected for each type, a corresponding DAO type may be generated to manage its persistence. Each of these DAOs contains a set of default operations that change based on the selected policy. These operations are defined using multiple interfaces within the Athlos API. Firstly, the `DAO` interface acts as a minimal set of database operations that are available under all policies: *create*, *update*, and *delete*. Based on the selected policy, each of the

subsequent interfaces extends this basic set of operations. The `UniqueDAO` interface extends this by including an additional *get* operation to allow the retrieval of a unique instance, while the `MultiDAO` interface also includes additional operations for retrieving lists of items and batch operations.

The policies described can be used to automatically generate DAOs for each type[10]. These must then be implemented by game developers based on the database technologies they opt to use. Through the use of an additional boolean meta-attribute called `isWorldSpecific`, types that always exist in the context of a certain world can be defined. This property can be used to adapt their DAOs to include additional database operations. For instance, terrain chunks and entities are always relative to a world and therefore need to be managed in a *world context*. Such operations may include retrieving all entities associated with a particular world, retrieving its terrain, and so on.

The persistence API provides several advantages. Firstly, it reduces development effort by allowing developers to select from a pre-defined policy and then implement specific, automatically generated operations tied to that policy. Despite this abstraction, developers are still able – and encouraged – to extend the generated DAOs to include additional operations where necessary. Furthermore, the use of DAO policies reduces the possibility of making errors because database operations are tied to the purpose of each type, as defined in the model. For instance, in a game that only needs to support a single world, DAO policies and meta-attributes will instigate the necessary constraints in the creation and retrieval of objects of this type. This guides developers by disabling functionality that should not be allowed, such as creating multiple instances of the worlds, unless they intentionally deviate from this pattern.

4.4.5 Data serialization

Data serialization is an important process for communicating data across nodes in a networked system and plays an important role in the development of an MMOG backend. Even though

[10]Types using the `NONE` policy are omitted and do not have a corresponding DAO.

data serialization is sometimes intrinsically linked to specific messaging protocols and communication methods, this section specifically focuses on the conversion of data objects which are stored in computer memory to other formats that can be easily communicated across networked devices. As with other aspects of MMOG development, there is a large variety of serialization approaches, with each one having its own advantages.

Serialization approaches

One of the most widely-used text-based data serialization formats is the eXtensible Markup Language (XML). XML is a markup language designed to store and transport data using self-descriptive tags that can be nested within each other to provide structure to a message. XML is platform-independent, which means that any system may support reading and writing in this format. Even though its extensibility and utilization within the world-wide-web have made it a popular option, XML is slower to process and uses more bandwidth than other alternatives. Another text-based data serialization format is JavaScript Object Notation (JSON), which is based on how JavaScript converts objects to and from string-based information. While the syntax of this format is inspired by JavaScript, the format itself is text-only and all major languages have support for converting objects to and from this format. Like XML, JSON is extensible and self-descriptive, but it does not use tags to define content, making its size smaller. Consequentially, JSON is easier and quicker to parse compared to XML, an important merit when it comes to the performance of MMOG backends. Another major advantage of JSON over XML is the inclusion of arrays, which makes it easier to serialize collections of data. Apart from these standardized formats, other text-based formats can also be devised to serialize information. For instance, bearing in mind the use of a specific communication protocol, games may choose to employ a custom format in which they only transmit the necessary information based on the context. This can lead to lower bandwidth use, as data sizes are reduced. As an example, consider that an action is made to move an entity to position (3, 4). Given that this action and its parameters (e.g. `row` and `col`) are known to the server, the client may only transmit the data `3,4` to a specific service, thus reducing the size of the message compared to XML or JSON. While this is advantageous for performance, two problems must be considered.

Firstly, despite offering better economy and performance, such non-standardized formats may take time and effort to implement. Devising a serialization scheme for each game is a hard and time-consuming process that may hinder the development of other, more important aspects of the game. Secondly, we must also consider the need to transmit large collections of data – a very likely scenario in an MMOG backend– or even worse, the ability of the serialization mechanism to support nested information. In such cases, the use of text-based formats, standardized or otherwise, may not provide the necessary features or performance.

An alternative to text-based serialization is binary serialization, which converts a data object into a stream of bytes. Replacing text with multiple types of data such as integers and booleans can reduce the size of a message significantly, especially in larger messages because these various types are encoded into the same number of bytes regardless of the number of digits they contain. Programming languages used in game development, like C++, C#, and Java include support for byte-based serialization through streams, which make it easier to convert data objects to bytes. However, these features are technology-specific, which adds extra challenges. For instance, it may not be possible to convert data into a common format that can be understood by machines that use different technologies. Secondly, streams are *consumable* items, meaning that once their data is consumed, the stream has to be flushed. Thus, when a stream is read the data has to be saved in memory for later use before it is lost. The nature of streams makes it harder to reason with serialization and makes the whole process considerably harder.

To handle data serialization, an experimental tool called ByteSurge is developed. ByteSurge uses Java streams to handle data serialization, by first allowing the developer to define data schemas, and subsequently using data containers to store data based on their schemas. The evaluation of this tool is described in Appendix 9.G, where it is compared with other approaches. Based on the results of this evaluation as well as analysis and comparison between ByteSurge and other approaches, it is determined that other, more standardized methods may be more suitable to handle data serialization.

While ByteSurge may provide a good alternative for further exploration in the future, the maturity and support of other frameworks like *Protocol Buffers* is preferred. Protocol Buffers

is a widely-used data serialization mechanism developed by Google, which features support for a growing set of languages, including those used in game development. While not specifically developed with MMOG backends in mind, Protocol Buffers offer a way to create extensible, language-independent data structures that can be easily serialized into binary form and then transmitted over the network using a variety of tools. This eliminates the need for developers to create game-specific serialization mechanisms and communication protocols, which is a tedious, time-consuming, and error-prone process. Unlike streams, PB offers a way to serialize data for games that run on multiple environments. This unlocks the potential to create multiple clients for each game and promotes experimentation through the creation of backends that utilize different technologies. Furthermore, PB offers relatively efficient serialization of data into binary form, which is faster to encode and decode into data objects compared to string-based counterparts like JSON. The use of PB also opens up the possibility of utilizing other related tools in terms of communicating serialized data. For instance, PB can be utilized in conjunction with Google's Remote Procedure Call (gRPC) framework to facilitate game services and the communication of data across the network (Wang et al. 1993).

Using Protocol Buffers in Athlos

The standard way of utilizing the Protocol Buffers mechanism is to first define how the data will be structured. This is achieved by using the Proto language, which provides a specialized syntax to define language-neutral message types. Developers then have to compile the Proto file which contains these definitions into code in their language of choice. The PB compiler converts these definitions to native protocol buffer classes in the selected programming language (i.e. Java, C++, etc.), which can then be used in a project. These *PB classes* contain the fields defined using the Proto syntax, accessor and management methods, and patterns used to create instances of these types.

While this offers many opportunities in terms of serialization, it involves the addition of extra steps during the development process. Moreover, it is very likely that some developers may not be familiar with this concept and especially the Proto syntax, which may hinder their

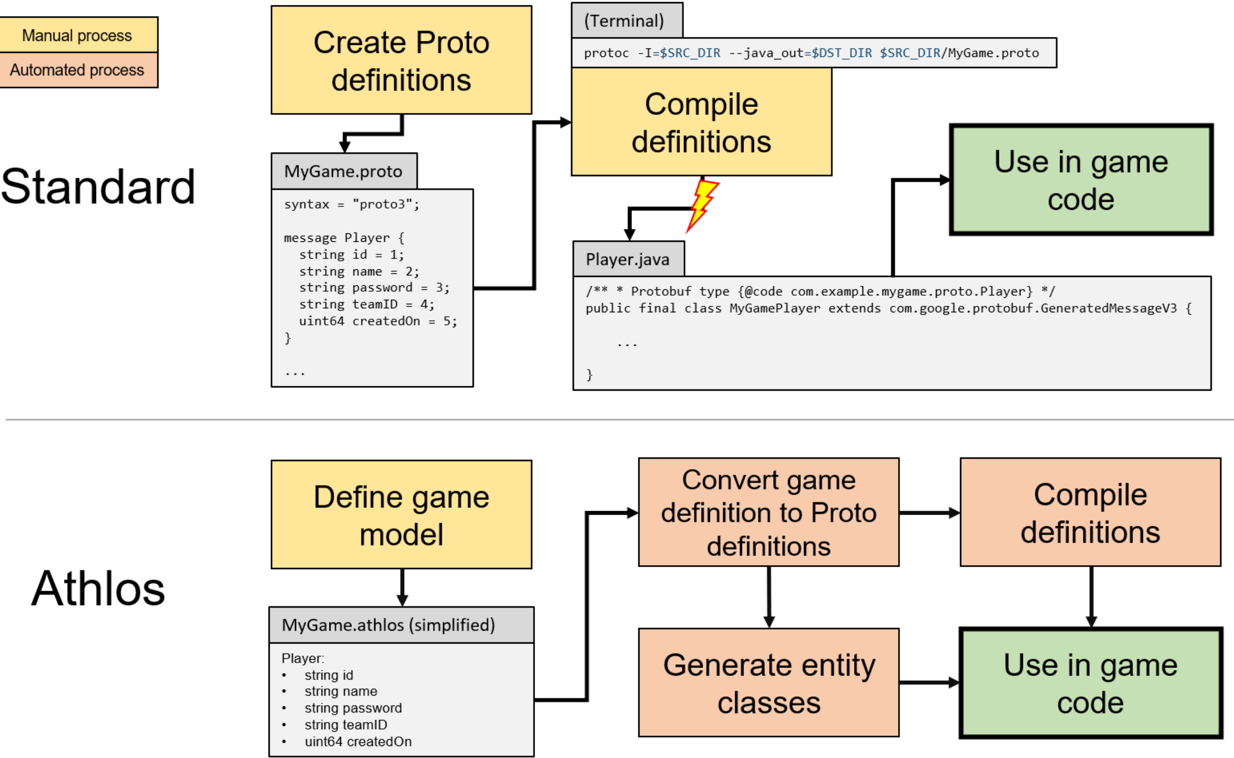


Figure 4.17: The processes involved in utilizing PB in the standard and Athlos approach.

efforts to develop an MMOG backend that works using Protocol Buffers. Using this approach as described above can also create inconsistencies in the game model because Proto types must be defined outside of the game definitions. To resolve this inconsistency, the Athlos framework completely hides Protocol Buffers during the game's design process, as illustrated in figure 4.17. This is achieved by having developers define types as they normally would, as part of the game definition's model using the project editor. Since both the game model and Protocol Buffer definitions are language-agnostic, it is possible to convert type definitions in the model to corresponding definitions in Proto syntax. This step is automated using various tools that are incorporated within Athlos, and thus developers do not have to learn the Proto syntax and manually define Proto definitions. Due to the elimination of these development overheads, it is expected that MMOG backends will be developed with less effort allowing game providers to focus on other, more important aspects.

While PB classes can be used as stand-alone classes to model, serialize, and communicate data, the amount of meta-data they contain is relatively large. As a result, their memory footprint is much bigger than that of a standard class that simply contains the declared attributes and

accessor methods. Using PB classes can be problematic in some cases. When trying to handle large numbers of objects in memory, such as in circumstances that are frequently seen in MMOG backends, this may result in memory overloads. PB classes are also immutable and can only be altered using special *Builder* sub-classes. The Athlos framework attempts to solve these problems by using both PB classes and simple, Plain Old Data Object (PODO) entity classes. In this approach, PB classes are reserved for serialization and communication, whereas *plain classes* are also generated and used for other tasks like runtime execution, persistence, and more, allowing backends to have a reduced memory footprint. While these classes are different items, they can be used to represent the same actual object as they can contain the same attributes and values. Within its persistence namespace, the Athlos API offers a way to convert between these two types of objects seamlessly through the use of two generic interfaces: `Modelable` and `Transmitable`. The `Modelable` interface is implemented by a PB class and allows the conversion of a PB class to its corresponding plain class, while the `Transmitable` interface is implemented by the plain class and can convert its objects into PB objects. These interfaces contain single methods: `toObject()` and `toProto()` respectively, which are automatically generated, implemented, and injected into the corresponding classes by the framework's tools. While most of the items defined in the model follow this paradigm, some items which may not be persisted but must still be serializable (such as enumerators) are excluded and only generated as PB classes. Similarly, utility items that may not be serialized (such as terrain generators) are also omitted and are only generated as plain classes.

4.4.6 Networking

The serialization of data within an MMOG backend is often tied to how the data is transmitted within the network, as well as how services are utilized. One of the most popular approaches used to enable communication and data sharing among machines in a network is the request-response cycle. In this approach, a client in need of a resource sends a request to a server, which sends back a response with the resource requested. This paradigm is relatively simple and relies on the client to initiate short-lived connections to the server, which makes it useful

in applications like the World Wide Web. On the other hand, the publish-subscribe model (aka. pub/sub) provides a messaging pattern that enables clients to subscribe to a server and listen for incoming updates, with the server providing these updates whenever required. While being considerably more complicated, this model allows for a more dynamic network topology and increased scalability. The diversity in approaches in terms of infrastructure, architecture, and development methodology makes it hard to standardize the use of these models. Game developers often opt to use concrete implementations which bind the underlying infrastructure with their communication protocols and services, which reduces maintainability, and expandability, and discourages experimentation. In MMOG backends, services must also operate efficiently at scale and accommodate simultaneous processing on multiple computing nodes regardless of the infrastructure type being used.

Common service model

At their foundation, dedicated/IaaS and serverless infrastructures work differently and thus need to be handled accordingly with respect to networking. Despite many differences, services used in both these infrastructures can still be represented with the same models. As shown in figure 4.18, a service typically acts as a ‘networked function’ by receiving a request object, executing a series of logical steps based on the input received, and then outputting a response. Within Athlos, services are defined through the project editor, with each service having its own unique name and being associated with a request and response model. Request and response models themselves can exist independently of any service and may be used by multiple services simultaneously. The definition of services through the editor allows both client and server components to have implementations of the same protocol, and thus communicate information reliably without the need for developers to define these protocols manually in each component. This works effectively in both dedicated/IaaS and serverless infrastructures. Due to the standardized nature of the request-response model, this can be implemented in various concrete implementations, such as Java Servlets, Node.js web containers, HTTP functions, and many more.

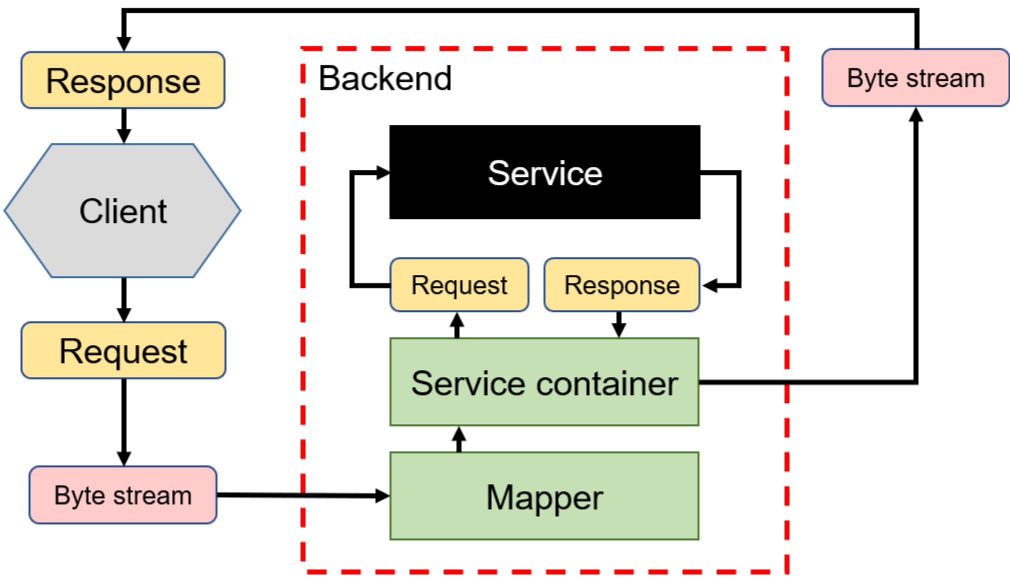


Figure 4.18: The service execution pipeline.

IaaS/Dedicated networking

While it is possible to abstract services by using a common model, the concrete approach used to deploy and execute these services is largely dependent on the infrastructure type being utilized. In IaaS and dedicated systems, the full customizability of the system can be leveraged to provide services using any approach. While a plethora of these approaches are available, those that provide the best performance and scalability are naturally preferred. One of the most widely-used approaches to providing services and data through a network are *sockets*. Socket technology can be used to establish a bi-directional link between two nodes in a network. The low-level nature of sockets allows full flexibility and the ability to utilize any protocol, including those created by Protocol Buffers. Despite offering a very efficient way of transmitting information, socket technologies come with increased complexity, reduced security, and large overheads in development effort – especially if there is a need to scale a system or provide load balancing. While many programming languages offer tools that may improve the process of developing applications that use sockets, their APIs are wildly different, and there is no standardized way to transmit information between applications running on different technology stacks.

Motivated by the shortcomings of using low-level sockets, the proposed approach aims to lever-

age a more standardized method. Remote Procedure Calling (RPC) systems provide an additional layer above sockets by defining a specific system-level protocol and then utilizing it to invoke services. RPC can be used with any communication protocol (i.e. TCP, UDP, etc.), and leverages the underlying power of sockets at a more abstract level. An example of such a system is Google's Remote Procedure Call (gRPC) framework. gRPC is a high-performance system that can run on a variety of environments including those frequently used in MMOG backends (Huang et al. 2021). It provides efficient access to services both in terms of performance and development effort and includes pluggable support for scaling, authentication, and more. With regards to MMOG backends, gRPC can fully support bi-directional communications, while also allowing services to be executed concurrently on different computing nodes.

The standard way to utilize gRPC is to define services using Protocol Buffer definitions, which makes its use fully compliant with the methods utilized in other aspects of development. Upon defining the services, the PB compiler can be used with a special gRPC plugin to convert these definitions into code for both client and server components. This conversion process automatically creates the necessary client and server stubs based on the protocol defined and thus makes it easier for developers to utilize this approach to communicate information to and from the backend.

The gRPC framework is used within Athlos to automatically generate services, stubs, and communication APIs for both clients and servers. In the proposed approach, game definitions and the subsequent service models included within them are automatically converted into gRPC service definitions in the PB files of each project. The PB compiler then generates the necessary classes to realize these services, their client stubs, and APIs. Within the boilerplate code of their project, developers can subsequently implement these services and their corresponding client stubs. Among other benefits, gRPC also provides the ability to call services both asynchronously and synchronously, using specialized stubs for each. Within the Athlos framework, services themselves are defined based on gRPC's service types:

- **Message to message** services transmit a single outbound message and receive a single inbound message.

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

Based on the needs of their game, developers can define each service to use each of these different types. This variety in data flow allows for more efficient access to resources, follows the request-response paradigm, and also facilitates bidirectional communication through stream-to-stream services. This feature is very useful in MMOG backends, as the state update mechanism can be integrated with gRPC. As a result, gRPC is preferred for IaaS and dedicated environments for many reasons, including:

- Its compatibility with the previously defined methods (e.g. serialization using Protocol Buffers).
- Its good performance and low latency, which are crucial for MMOG backends.
- Its suitability for use in public clouds, as it includes support for scalability, load balancing, and other important features necessary for cloud operations.
- Its support for the automated handling and generation of services, stubs, and communication APIs on both the client and server which produces a consolidated protocol and reduces development effort.
- The inclusion of APIs for both synchronous and asynchronous execution, and operation in multi-threaded environments.
- Its support for a variety of programming languages, which follows the approach-independent nature of the proposed framework, and support from a large community of developers.

- The ability to simplify development as developers do not need to learn the specifics of how gRPC works, but only have to implement the generated services and stubs.

Serverless networking

Despite these advantages, gRPC is not compatible with most serverless environments. The bounded execution time of services in environments like Google's App Engine Standard, Flexible, or Amazon's Lambda Functions makes the use of gRPC's streaming services impossible. Even for unidirectional services like message-to-message, gRPC works by invoking procedures at lower levels of abstraction than those found in many serverless computing options. The incompatibility of these approaches creates a challenge in facilitating service invocation and data transfer in serverless systems. To meet this challenge, the proposed framework provides support for abstract services in serverless layers by only utilizing the message-to-message service type and defining the `AthlosService` interface within its API. This interface provides a way to model services based on the paradigm shown in figure 4.18. Services themselves, their corresponding service containers, and mappers are automatically created by the framework's code generator based on the models defined within the game definition. These components are intrinsically linked within the produced code and allow services to receive and transmit information in byte streams without the need to explicitly serialize or de-serialize data from and to objects, link services to containers, and so on. This leaves only one task up to game developers: the implementation of these services based on their game's logic. In listing 4.1, a simplified Java implementation of a service for a serverless environment is shown, illustrating the use of the `AthlosService` interface, the request-response models, Protocol Buffers, as well as the database manager and DAO handlers. To allow quick access to these services from the client-side, service stubs similar to those found in gRPC are also generated and implemented to offer access to these endpoints. Despite the differences in communication methods and protocols, the client-side stub APIs are identical to those found in gRPC's stubs. This allows developers to utilize a common stub API in either of the approaches without having to adapt to different styles of invocation. In serverless environments, service invocation typically occurs over HTTP, with mappers like `web.xml` in Java Servlets or routing parameters in Node.js direct-

ing requests to the appropriate service based on the URI of each call. Another consideration is that for a limited set of serverless approaches, like Google's App Engine Flexible or Amazon's Lambda Functions, it may be possible to utilize bi-directional communications through stream-to-stream services. Such approaches use HTTP-friendly technologies like WebSockets to facilitate bi-directional communication, making it possible to integrate a state-update mechanism within services. To leverage the advantages of the pub/sub model, the Athlos framework embraces the differences and features of each serverless platform in a way that ensures MMOG backends can operate with the largest amount of in-house components possible. For approaches that support this feature, special amendments are made to the constraints of the framework to allow bi-directional services to be defined, even though this feature is disabled for other serverless approaches. Finally, for environments that do not support bi-directional communications, the framework allows developers to either resort to third-party state-update mechanism implementations or fall back to *polling* techniques.

```java
public class CreatePlayer implements AthlosService<CreatePlayerRequest,
    CreatePlayerResponse> {

    @Override
    public CreatePlayerResponse serve(CreatePlayerRequest request, Object...
    additionalParams) {

        //Create a response to send later:
        CreatePlayerResponse.Builder responseBuilder = CreatePlayerResponse.
    newBuilder();

        //Get player information from request:
        MyGamePlayerProto playerProto = request.getPlayer();

        //Check if a player with this name already exists:
        MyGamePlayer existingPlayer = DBManager.player.getByName(playerProto.
    getName());
        if (existingPlayer != null) {
            responseBuilder.setStatus(CreatePlayerResponse.Status.PLAYER_EXISTS)
    ;
            return responseBuilder.build();
        }

        //Attempt to create player object/record in database:
        if (DBManager.player.create(playerProto.toObject())) {
            return responseBuilder
                    .setStatus(CreatePlayerResponse.Status.OK)
                    .setPlayer(player)
                    .build();
        }
        else {
            return responseBuilder
                    .setStatus(CreatePlayerResponse.Status.SERVER_ERROR)
                    .build();
        }
    }
}
```

Listing 4.1: A simplified implementation of a service for a Java-based serverless environment.

4.4.7 Performance and scalability

The previous sections presented various methods that aim to improve the development process of MMOG backends and enable them to be deployed on commodity cloud platforms. Two other major aspects that influence the development of MMOG backends are performance and scalability. While the use of a novel architecture in combination with certain design principles and specific technologies promotes better performance and scalability, it does not guarantee it. This is exemplified in section 3, in which the feasibility study demonstrates the severe

limitations imposed by the use of cloud technologies that are not complemented by additional software-based methods for providing improved performance and scalability.

State scalability

In the experimental MMOG developed in the feasibility study, several NoSQL data stores were used to provide persistence in each implementation. Even in a simplistic game like Minesweeper, the experiment revealed severe limitations on how many tiles/cells can exist within a game board, given that the entire board is stored as a single data store object. The object size limitations of each data store are implemented by design to improve their ability to scale. However, this is at odds with attempts to create scalable game worlds as this limit can be quickly reached, even for simple games like Minesweeper. This thesis defines the ability of an MMOG backend to provide scalable game states as *state scalability*. While state scalability is limited by data store object sizes, other solutions like cloud-based RDBMSes can offer the ability to create larger states but are less efficient at distributing data across multiple nodes. The first step in bypassing the limitations in state scalability imposed by cloud data stores is to de-couple the world's terrain state from any entities that may exist in it. In the feasibility study, a coupled terrain-entity approach was used to implement the state of the game, where entities are directly associated and exist as part of the terrain. Coupling allows developers to retrieve both terrain and entities that exist inside it at the same time, thus simplifying development. However, the inclusion of entities within terrain states increases their size and leads to decreased scalability.

A more systematic method to approach the problem of state scalability is to first allow the terrain of a world to exist independently of any entities and vice versa. This approach complicates the process of retrieving the game state – defined as *state retrieval* – compared to a coupled terrain-entity approach because it necessitates the management of more objects. Nevertheless, it creates a better logical connection between the game elements as some types of games might not feature any terrain or world state but still contain entities within their states. The de-coupling approach also makes it possible to significantly reduce the size of the terrain as

entities can now exist as independent objects. Reduced terrain size can allow games to create and manage bigger states than those previously possible.

Despite enabling larger states, the terrain-entity de-coupling approach is not conducive to scalability on its own. While positively offsetting the limitations of data stores, it does not negate them completely. This means that MMOGs which need to scale their states to huge sizes will still reach these limitations at some point. Various approaches for modeling and storing game states can be explored. For instance, the Minesweeper implementation uses a *unified state* approach in which all cells in the game state are stored under a single object. A major advantage of this approach is that a single query can be executed to fetch the entire game state, making this process more efficient both in terms of performance and development effort. However, this also has a major drawback as it assumes that the entire world state *can* be stored in a single object. This approach is by definition not scalable because as soon as the size of an MMOG's world reaches or exceeds the size limitations of the data store being used, it will no longer be possible to expand it.

A potential alternative to this approach is the exact opposite: storing the state of each cell in the terrain individually and thus having multiple, separate objects of which the terrain is comprised. Assuming that the state of a single cell will not reach the size limitations of data stores[11], the terrain state can expand regardless of its current size and reach the limitations of the hardware resources being utilized. This *cell-based* method would be very advantageous, with the exception of a major problem: the unacceptably large number of queries required to fetch the game state. While many persistence options provide support for more complex queries that include filters, not all datastores or caching systems include this feature. This is detrimental to the use of this method because an N number of queries would be needed to retrieve an N amount of cells. Considering that partial game states may be comprised of thousands of cells, the sheer amount of queries made – one of the most expensive backend operations in itself – will cause games to suffer from bad performance at even small scales. Further exacerbating this problem is the fact that public cloud providers tend to charge their users based on the number

[11]Reaching data store size limitations in each cell object is a very unlikely scenario that probably means game developers should rethink their game's design.

of queries made to the datastore which makes this approach cost-inefficient.

The chunk-based method

Considering both the unified and cell-based approaches, there seems to be a trade-off between good performance, mostly quantified by the number of queries made, and the scalability of a game world, measured in terms of the maximum possible cells. MMOG backends that use the unified approach may perform better but are very limited in scale, whereas those using the cell-based approach must use specific persistence options or suffer from poor performance to accommodate for increased scale. Inspired by Minecraft's method of world segmentation into "*chunks*" (Fridh & Sy 2020), this thesis provides a third alternative to the state scalability problem.

In the *chunk-based* approach, the terrain is modeled using a combination of cells and chunks. While cells are still used to enable the division of a world's area into smaller units with individual states, like a map of pixels, these are further organized into chunks. Chunks are objects that act as containers and are composed of collections of adjacent cells, which can be subsequently retrieved and managed. Using cells and chunks in combination may provide a solution to the problem of scalability and performance. The aims of this method are to a) allow theoretically infinite game states, b) enable efficient access to the game state, and c) balance the scalability of an MMOG backend with its performance. To achieve these goals, the chunk-based approach allows reliable state retrieval by setting a hard limit on the maximum number of cells that can be stored within a chunk – defined as `MAX_CELLS`. This game-wide value can be adjusted during development but must remain constant while the game is deployed. By using a constant number of maximum cells in a chunk, it is possible to geometrically calculate which chunk contains which cell. This unlocks the potential of accessing any part of the state, no matter how disjoint it may be, with minimal overheads. To implement this method, cells are stored within a chunk using a map data structure. In this map, the key is a hash value of the cell's coordinates separated by a comma while the value is the terrain cell. It is therefore possible to retrieve any cell within the chunk in constant time by using its coordinates and then retrieving the value of the key matching their hash. The same concept is extended to chunks. Firstly, the

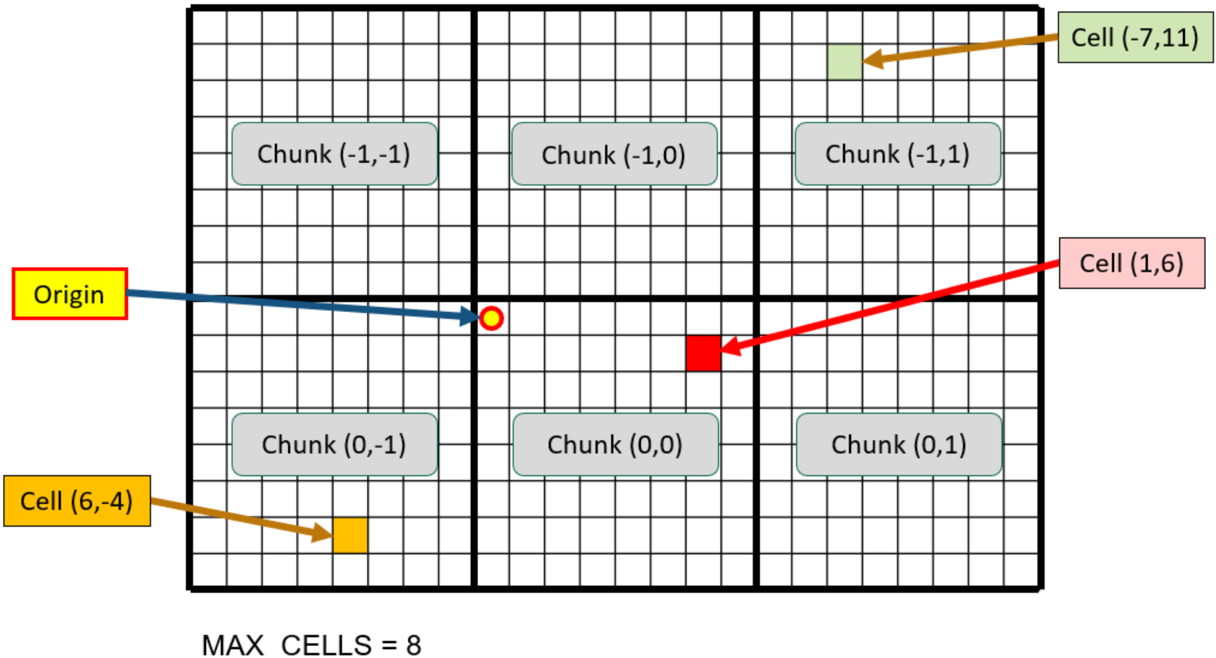


Figure 4.19: A depiction of a game world's terrain divided into chunks and cells, with each having its own coordinates.

IDs of chunks related to a world are added to their world's list of chunks, shown in the model in figure 4.7. This allows the retrieval of all chunks which belong to a certain world. Like cells, chunks also have positions within the game world as they are collections of adjacent cells. This relationship is visually demonstrated in figure 4.19. The position of each chunk is determined by `MAX_CELLS`, which explains why this value must be constant. It is possible to retrieve a chunk of cells, and subsequently a cell, or multiple cells within the chunk using rather simple arithmetic operations and logic.

The fact that cells are parts of chunks somewhat complicates their retrieval as these arithmetic operations are relatively simple in their nature, but follow a system that is not straightforward to understand. For instance, a frequent use case in an MMOG would be to retrieve a cell's state to update it based on a player's action. Assuming that the needed cell E has coordinates (r, c), it is necessary to first calculate the respective coordinates (R, C) of its enclosing chunk H. Such operations are made possible with *chunk arithmetic*, which utilizes combinations of the following: a) the cell's coordinates, b) the known geographical limits of the world, and c) the `MAX_CELLS` constant. For instance, the chunk's row R can be calculated using the

mathematical operations defined in equation 4.1. First, an offset s is calculated based on the cell's row. This offset value is necessary, as negative coordinate indices start from -1 rather than 0. This offset is only taken into account when the cell's row is not a number than can be divided with the constant M (`MAX_CELLS`), i.e. not a cell bordering another chunk. Applying these operations on a cell with coordinates $(-7, 11)$ and with $M = 8$, yields that this particular cell exists in chunk $(-1, 1)$, as shown mathematically in equation 4.2, and confirmed visually in figure 4.19.

$$s = \begin{cases} 0, & r \geq 0 \\ -1, & r < 0 \end{cases} \tag{4.1}$$

$$k = r \, mod \, M$$

$$R = \begin{cases} \frac{r}{M} + s, & k \neq 0 \\ \frac{r}{M}, & k = 0 \end{cases}$$

Example: Cell (-7,11), r = -7, c = 11

$$s = -1 \, (r \; < \; 0) \tag{4.2}$$

$$k = -7 \, mod \, 8 = 1$$

$$R = \frac{-7}{8} + -1 \;\; (s = -1, k \neq 0)$$

$$R = -1$$

$$s = 0 \, (c \; \geq \; 0)$$

$$k = 11 \, mod \, 8 = 3$$

$$C = \frac{11}{8} + 0 \;\; (s = 0, k \neq 0)$$

$$C = 1$$

$$\therefore Cell(-7, 11) \in Chunk(-1, 1)$$

Such calculations are implemented as utility methods in the `Chunk` class and used in various contexts such as state retrieval and update. Chunks are persisted using their ID as a key, which is managed internally by Athlos and like in cells, is the hash of their coordinates. It is therefore possible to compute the location of a chunk containing a cell using these operations, and subsequently retrieve it in constant time from a map-like structure that is consistent with key-value structures seen in NoSQL datastores. Further to these, *terrain identifiers* are used as special utility objects to allow chunks to be easily indexed, identified, queried, and retrieved. Terrain identifiers are introduced so that these operations can take place without having to transmit the large states of chunks, which may contain data about thousands of cells. Instead, terrain identifiers, which are lightweight objects, model the attributes of chunks without containing their cells, allowing for improved efficiency in such operations. The overheads of utilizing terrain identifiers and their potential benefits and drawbacks are discussed and evaluated in section 6.

Performance-wise, it is expected that this approach will yield significant improvements to various state management processes within MMOG backends. Firstly, it may allow parts of the state to be retrieved in constant time regardless of how far away they are from the origin or from other parts of the state. At the same time, the chunk-based method makes it possible to extend the game world state to massive scales, while also reducing the number of queries needed to fetch the state to a worst-case scenario of $Q = \frac{N}{M} + 3$, where $N > 0$ and is the number of cells to fetch, and M is the `MAX_CELLS` constant. This method also allows MMOG backends to overcome the object size limitations imposed by public cloud data stores. While this is a standardized method with which to store and retrieve the game state, Athlos enables some degree of customization. The `MAX_CELLS` constant is allowed to vary between a range of 4 to 64 (4×4 to 64×64 cells) in each game. This customization can accommodate a variety of scenarios. Especially in extreme circumstances where games may utilize very large cell states, it may be beneficial to reduce the constant to include *fewer* cells in each chunk and thus stay within the datastore size limits. More importantly, it may be possible to leverage this flexibility to balance performance with scalability in game-specific scenarios, a concept that is further studied and evaluated in section 6. It is also expected that by using chunks, the processes of generating and communicating game states will be made more convenient and efficient. For

instance, terrain generators can generate terrain in chunks using single batch jobs rather than having to schedule a job for each cell – an approach that is computationally and economically expensive.

World contexts, State API, and Terrain generators

The chunk-based method provides more opportunities to organize the development effort by consolidating different processes related to state management. *State management* refers to any process dealing with the retrieval, observation, modification, or dissemination of the game state. The objective of this effort is to allow developers to quickly and effectively use existing tools to carry out such operations without the need to implement them manually. The first step towards this is the organization of worlds into *world contexts*. World contexts are utility classes that encompass the functionality of a single world and allow developers to control worlds and their associated items as a unified, logically linked realm. While the structure of the Athlos model is designed to enable scalability by defining disjoint objects with relatively loose relationships, the concept of world contexts aims to reverse this pattern to a more constricted environment through the use of software components, aiming to provide access only to valid items and states. As an example, world contexts can guide the process or state retrieval by leveraging several specialized functions. Even though developers can manually retrieve items like terrain or entities, world contexts offer various default ways of achieving these relatively menial tasks. A world context implements several methods with which developers can request chunks based on their position – a much more relevant gameplay parameter – rather than their key/ID. This is achieved through the use of the `requestChunk()` method, shown in algorithm 1, which runs through several steps to provide access to the chunk being requested. First, the algorithm attempts to retrieve the chunk from the data store. If the chunk has already been created and generated, it is retrieved and returned. Otherwise, the game's terrain generator is summoned to *conditionally* generate a new chunk, which is subsequently stored in the data store and returned.

Algorithm 1: The default algorithm used in the `requestChunk` method.

```
   Data: chunkRow, chunkCol, worldID
   Result: Conditionally returns a chunk given its coordinates.
1  hash ← hash(chunkRow,chunkCol);
2  cIdentifier ← NULL;
3  identifiers ← List terrain identifiers from DB, where id == worldID;
4  forall i in identifiers do
5  |  if i.chunkPosition.hash == hash then
6  |  |  cIdentifier ← i;
7  |  |  break;
8  |  end
9  end
10 if cIdentifier != NULL then
11 |  Get chunk from DB, where id == cIdentifier.id;
12 |  return chunk;
13 else
14 |  generatedChunk ← generator.generateChunk(chunkRow, chunkCol);
15 |  Create generatedChunk in DB;
16 |  Get world from DB, where id == worldID;
17 |  world.chunkIDs.add(generatedChunk.id);
18 |  Add generatedChunk.id to world.chunkIDs;
19 |  Update world in DB;
20 |  return generatedChunk;
21 end
```

A related method, `generateChunk()` called in line 14 of Algorithm 1, defines the behavior for generating a chunk. The default algorithm used for the generation of chunks is shown in algorithm 2. Within this method, the bounds of the chunk are first calculated, the chunk is created, and its attributes are set. Upon its creation, the chunk is populated with the necessary cells. This is implemented in a loop, which runs through all columns and rows to generate the cells in the defined range. Before each cell is generated, its location is validated with respect to the world's bounds. These processes are implemented within the `TerrainGenerator`, which is a utility class that defines how terrain should be generated. While there are several default methods in this class, developers are asked to implement the `generateCell()` method to define how each cell will be generated for their game. The result of this fully custom method is subsequently used by the `acquireCell()` method, which is called within `generateChunk()`.

Within a world context, the functionality of generating and requesting terrain is hidden from developers. Instead, world contexts include other methods which can be used to retrieve, modify, and save the state of a world, compose state updates, manage world sessions, subscribed

clients, and more. These methods make use of the aforementioned processes and algorithms in the background, without requiring the explicit direction of the developer to define how terrain should be managed. If needed, developers are still able to customize these operations for game-specific circumstances by modifying the default code to improve efficiency and performance or provide extra features. A summary of the state API and all the related functionality is shown in Appendix 9.C.

Algorithm 2: The default algorithm used to generate chunks.

Data: chunkRow, chunkCol, world
Result: Conditionally returns a generated chunk.

```
1  chunkStartRow ← Chunk.getChunkStartRowFromChunkRow(chunkRow);
2  chunkLastRow ← Chunk.getChunkLastRowFromChunkRow(chunkRow);
3  chunkStartCol ← Chunk.getChunkStartColFromChunkCol(chunkCol);
4  chunkLastCol ← Chunk.getChunkLastColFromChunkCol(chunkCol);
5  chunk ← Chunk();
6  chunk.worldID ← world.id;
7  pos ← MatrixPosition(chunkRow, chunkCol);
8  chunk.position ← pos;
9  chunk.id ← pos.hash;
10 cells ← new Map();
11 for c ← chunkStartCol to c ≤ chunkLastCol do
12 |   for r ← chunkStartRow to r ≤ chunkLastRow do
13 |   |   cell ← acquireCell(r, c);
14 |   |   cells[hash(r,c),cell];
15 |   end
16 end
17 chunk.cells ← cells;
18 return chunk;
```

Partial states, snapshots, and modifiables

The retrieval of a scalable state can be a relatively complex process that sometimes involves many algorithms working together. As mentioned in section 4.3, only a part of the world state can be accessed at a time as the sheer size of the state of an MMOG would lead to a significant reduction in performance. The framework's state API defines several methods used to retrieve the state of the world within a context. Such methods can be used to retrieve terrain, entities, and more, either by using the coordinates of a center position with a range, or a list of observing entities. For the latter, the AoI of the entities is used as a range to compose the partial state. The default procedure for retrieving terrain using the AoI of observing entities, found within the framework's `getTerrain()` method, is shown in algorithm 3.

Algorithm 3: The default algorithm used in the `getTerrain` method to retrieve a partial state.

```
   Data: pEntities
   Result: Returns a partial state observed by a set of entities.
1  cells ← Map();
2  chunksNeeded ← new Set();
3  forall entity in pEntities do
4      minRow ← entity.position.row - entity.aoi;
5      maxRow ← entity.position.row + entity.aoi;
6      minCol ← entity.position.col - entity.aoi;
7      maxCol ← entity.position.col + entity.aoi;
8      INC_STEP ← minOf(entity.aoi, Chunk.SIZE);
9      for cellRow ← minRow; cellRow ≤ maxRow; cellRow += INC_STEP do
10         for cellCol ← minCol; cellCol ≤ maxCol; cellCol += INC_STEP do
11             chunksNeeded.add(Chunk.getPosition(cellRow, cellCol));
12         end
13     end
14 end
15 chunks ← List();
16 forall cN in chunksNeeded do
17     if world.chunkIsInBounds(cN.row, cN.col) then
18         chunk ← requestChunk(cN.row, cN.col);
19         chunks.add(chunk);
20     end
21 end
22 forall entity in pEntities do
23     forall chunks do
24         forall chunk.getCells() do
25             pos ← cell.pos;
26             distance ← pos.distanceTo(entity.position);
27             if distance ≤ entity.aoi then
28                 cells.put(pos.hash, cell);
29             end
30         end
31     end
32 end
33 return cells;
```

The concept of AoI, seen in multiple related works (Assiotis & Tzanov 2005, Nae et al. 2011), is instrumental in retrieving partial states. In algorithms like the one included in the `getTerrain()` method, the boundaries of the AoI of each entity are calculated based on its position and its AoI radius. These boundaries are used to determine which parts of the terrain should be included within the state retrieved. This concept can also be applied to retrieve entities, based on the positions of the observing entities owned by a player, thus combining

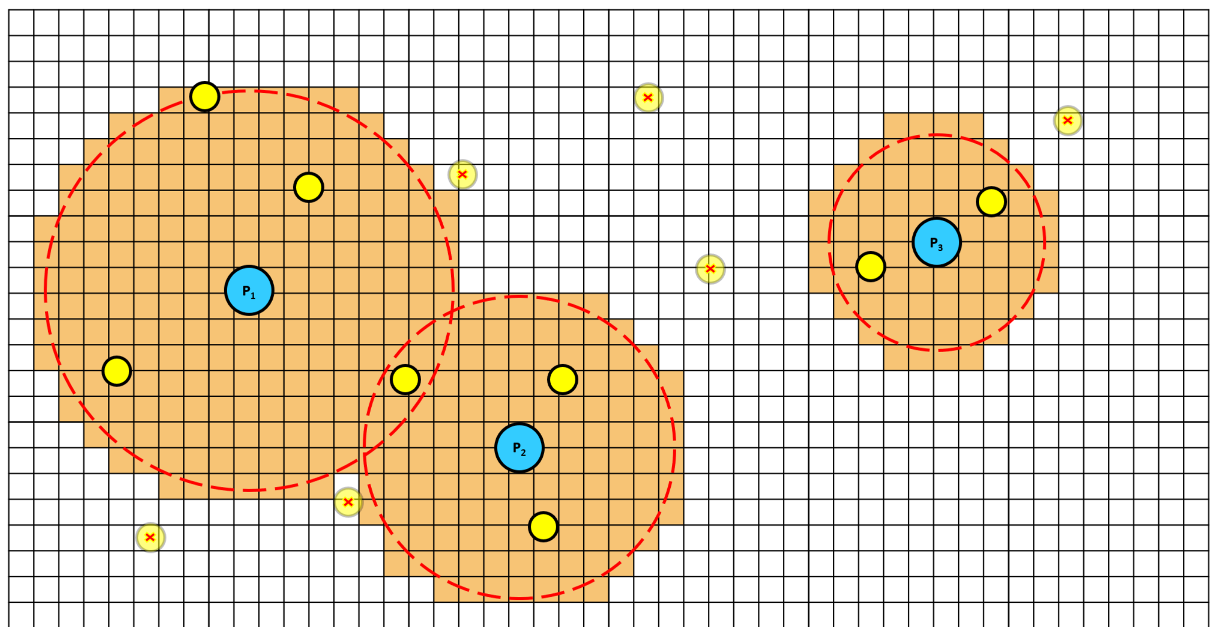


Figure 4.20: An illustration of the AoI concept in action, when retrieving the partial state.

terrain and entities to compose a partial state. This is illustrated in figure 4.20, where the AoIs of the entities P_1, P_2, and P_3, owned by player P, determine which parts of the terrain will be retrieved (highlighted in orange). Furthermore, these AoIs also determine which entities will be part of the partial state – with those being marked with an $\times$ and translucent background being excluded from the state.

Using formal notation, the AoI of an entity P_x can be defined as $A(P_x)$. The set of all available entities in a given world W, is indicated by W_ε, whereas the set of all terrain cells in the same world is W_k. The set of entities belonging to a player P can be defined as $P_{1..n}$, where n is the last created entity, or as P_ε collectively. The distance between entity ε and entity x is denoted as $D(\varepsilon, x)$. Finally, the partial state viewed by a player P is annotated as $\Omega(P)$.

Using these definitions, the subset of entities to be retrieved as part of the partial state ($\Omega(P_\varepsilon)$), can be described as:

$$R(x, y) = D(x, y) \leq A(x)$$
$$Y = \forall p \in P_\varepsilon[\forall w \in W_\varepsilon](R(p, w))$$

$$\Omega(P_\varepsilon) = P_\varepsilon \cap Y$$

Similarly, the inclusion of terrain within a partial state ($\Omega(P_k)$) can be formally described:
$V(x, c) = D(x, c) \leq A(x)$
$\Omega(P_k) = \forall p \in P_{1..n}[\forall c \in W_k](V(p, c))$

The full state of a world can be regarded as an ever-existing entity that evolves over time due to actions and events that unfold during gameplay. To reason with the state more effectively, it helps to think about clients and backends as observers of the world state at specific moments, with backends having the extra ability to manipulate it when necessary. A partial state of the world retrieved at a specific time is defined as a *snapshot*. Snapshots can be *localized* (i.e. player-specific) or *non-localized* (i.e. not player-specific), and can help backends observe a limited, time-specific view of the game state. Snapshots are ideal for *state observation* and *state dissemination*, because they are associated with a specific time. In contrast, raw partial states, which are more generic versions of snapshots, are not associated with a specific time and are used to carry out *state retrieval* and *manipulation* operations. Both partial states and partial state snapshots can be used to quickly retrieve fragments of the state from the point of view of players, rather than make calls to functions that retrieve a globally available state. Without snapshots, developers would have to manually provide additional parameters to those functions to achieve their objective, an approach that is more conducive to logical errors. Snapshots also play an important role in the state communication process by providing an order to the updates being received by the clients. As each snapshot is associated with a specific timestamp – the time at which it was captured – clients can keep receiving state updates as snapshots, placing them in an ordered queue, and subsequently start presenting these snapshots to the player in order of reception. This can help smooth out the presentation of state updates to the players, as the client can receive multiple snapshots from the state update mechanism and smoothly transition its local state to match the latest snapshot. An overview of this process is shown in figure 4.21.

Apart from observing and retrieving the state of a world, an MMOG backend must also be able to modify it. This is possible by first retrieving a partial state, modifying it within the runtime, and then saving it back to the data store. While this seems like a relatively straightforward process, it involves several additional steps, which are complicated by the use of immutable protocol buffer objects. Protocol Buffer objects can be converted to their plain-object, mutable counterparts relatively easily, but this conversion wastes valuable execution time and increases

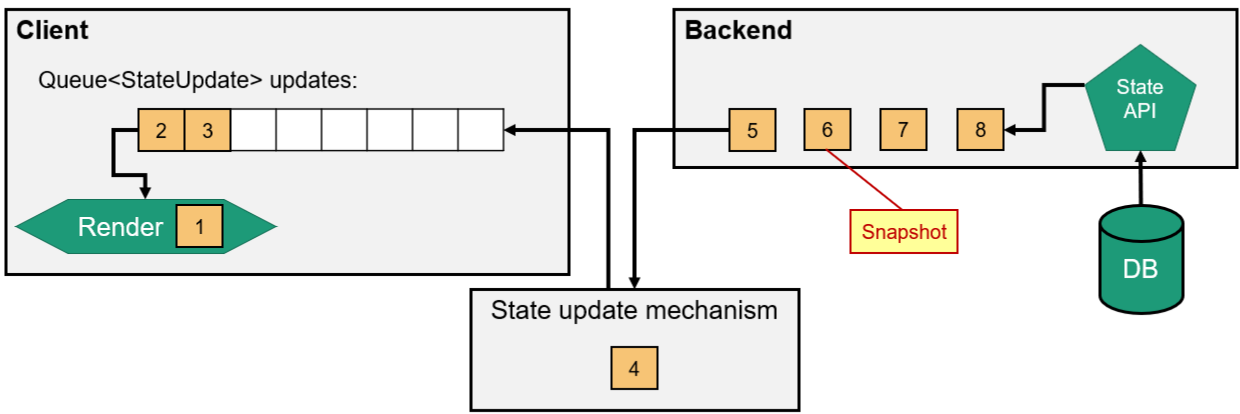


Figure 4.21: The process of retrieving and communicating snapshots of state updates from the backend to the client.

latency. Especially for bulky objects like partial states which are composed of potentially thousands of terrain cells and entities, this may not be a viable option. Instead, Athlos improves this process by offering the ability to internally de-compose protocol buffer objects into their subsequent builder objects and apply modifications using *Modifiables*. A modifiable is simply a generic interface that contains a single method called `modify()` that defines how an object should be modified. In such a case, terrain or entities can be modified by simply calling the corresponding `modify()` method already defined in the State API, passing the necessary parameters, and then implementing a modifiable. Considering the *standard approach* example shown in listing 4.2, developers would have to first manually de-compose a protocol buffer into a builder object, define the modifying behavior, and then compose the builder back to its original form.

```
1 EntityProto.Builder builder = partialState.getEntitiesMap().get(entityID).
      toBuilder();
2 builder.setDirection(Direction4.NORTH);
3 partialState.toBuilder().putEntities(entity.getId(), builder.build());
```

Listing 4.2: Modifying partial states using the 'standard' approach.

Instead of manually working with builder objects, developers can use Modifiables, as shown in listing 4.3 to create a more organized and readable code structure. While the number of lines of code is the same in both approaches, using modifiables eliminates the complexities of working with protocol buffers and their builders, removes the need to retrieve and associate information within them, while also offering the ability to leverage lambda functions – where the language supports it – to completely hide Modifiables from the code.

```
1 State.Entities.modify(partialState, entity.getId(), entity -> {
2     entity.setDirection(Direction4.NORTH);
3 });
```

Listing 4.3: Using modifiables to change the partial state.

The Area of Effect and the state update mechanism

Perhaps the most complex process related to the state of an MMOG is its dissemination to the players, which is defined within Athlos as a *state update*. The state update type, described in section 4.3.9, is used to store the information related to a state update, which includes an optional partial state, an update timestamp, lists of entities, and terrain to refresh or delete, and finally any other game-specific information as seen necessary by the developer. State updates are initiated when an action or event takes place in the game world. To correctly process an event, the partial state of that area must first be retrieved, updated based on which action or event is taking place, persisted in the data store, and then sent to the observing players. To handle state updates, the proposed approach uses a novel architectural component called the state update mechanism, initially presented in figure 4.13, which is responsible for the following tasks:

1. Defining what type of update has taken place. (**Definition**).
2. Identifying which clients should receive the update, based on game-specific rules (**Filtering**).
3. Composing multiple state updates from the perspective of each player (**Composition**).
4. Disseminating the state updates to their corresponding clients as efficiently as possible (**Distribution**).

In the first step (definition), two types of state updates are defined: *Refresh* and *Delete*. Refresh updates are used to refresh a part of or the entire local state of the client. Depending on the context, refresh updates may also transmit new data which does not exist in the local state. Delete updates are updates that delete parts of the state that should no longer be accessible to the player, either because of the game's rules or because they are regarded as outdated. To aid the creation of state updates, Athlos introduces the `StateUpdateBuilder`, which is responsible for defining the type of update taking place and which parts of the state are to be

refreshed or deleted. Instances of these builders, which are interim, mutable objects, can be passed to the next steps.

Many games choose to update the state of all clients regardless of what type of update takes place and where. This basic approach is simple to implement, but has a major drawback – it is extremely inefficient. As the number of players increases, the amount of time taken to retrieve and send the state updates to these players tends to increase exponentially as more actions take place, leading to higher latency and in some cases, complete depletion of resources. To solve this problem, Athlos introduces the *Area of Effect* (AoE), which is the area within which an action is perceived to affect the game's state. Each state update is associated with an event or action and by extension a specific AoE. The AoE is a circular area that spans a certain radius away from the position at which the action or event is taking place, and can vary depending on the magnitude of the action. For instance, an action that simulates the explosion of a grenade may have a significantly smaller AoE than an action simulating the explosion of a large bomb. While this is similar to the AoI concept used for entities, the AoE of actions is not directly associated with any action type in code and can be changed on a per-action basis. This allows the same action types to have different AoEs. This is by design, as actions are never instantiated and are merely converted to services when the project's code is generated. By using the concept of AoE, it is possible to limit the scope of the items being updated. In turn, this can lead to a reduction in the number of clients that need to receive a state update, and therefore lower resource usage, latency, and performance improvements.

The concept of AoE is particularly useful in the filtering step, during which a variety of factors are used to limit the number of clients which have to receive a state update. One of the most popular factors affecting this decision is whether the combined AoIs of the entities of an observing player intersect with the action's AoE. Both of these areas can be perceived as circles, and therefore basic geometric formulas can be used to calculate if any of these AoIs intersect with the action's AoE. If a player owns at least one entity that has an AoI that overlaps with the action's AoE, then the player must be made aware of the state update. This is visually illustrated in figure 4.22, where the position of the action is marked with $\times$, and its AoE is highlighted with light red. The entities owned by two players P and Q are also shown, and their AoIs are indicated with dashed lines. In this example, entities P_1, P_2, and Q_2 have AoIs that overlap with the action's AoE. Therefore, both players must be made aware of this occurrence,

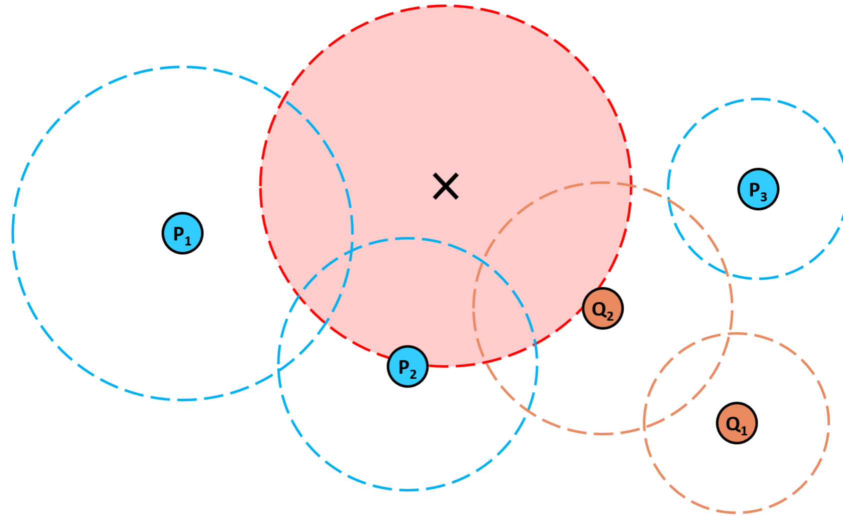


Figure 4.22: An illustration of the AoE being used to filter state updates.

and their subsequent state updates, as they both have at least one entity perceiving the event. The logic behind this filtering process is implemented by algorithm 4, which takes into account various parameters such as the intersection of entity AoIs and action AoE, the player's camera position, and whether the player is the initiating party. Like other default algorithms, this can be customized to suit game-specific needs.

Algorithm 4: The algorithm used to filter the sessions to receive a state update, based on the concepts of AoI and AoE

```
   Data: iSession (initiating session)
   Data: aoe (action area of effect)
   Data: aPosition (action position)
   Data: worldID
   Result: Filters the sessions that should be updated.
 1 allSessions ← State.forWorld(worldID).subscribedSessions;
 2 filterSessions ← new Map();
 3 forall ws in allSessions do
 4     hasEntitiesInAOI ← false;
 5     pEntities ← List entities of ws.playerID, and ws.worldID from DB;
 6     if iSession.id == ws.id then
 7         filterSessions[ws, pEntities];
 8     else
 9         forall pe in pEntities do
10             if pe.aoi > 0 then
11                 distance ← pe.position.distanceTo(aPosition);
12                 if distance - aoe < pe.aoi then
13                     hasEntitiesInAOI ← true;
14                     break;
15                 end
16             end
17         end
18         cameraInRange ← ws.camera.distanceTo(aPosition) - cameraRange ≤ aoe;
19         if hasEntitiesInAOI and cameraInRange then
20             filterSessions[ws,pEntities];
21         end
22     end
23 end
24 return filterSessions;
```

Game developers are encouraged to extend the `filterUpdateSessions()` method which implements this algorithm to handle game-specific needs that require further filtering. For instance, some games may want to take into account obstructions between entities in the world, or Line-Of-Sight (LOS) filters, which may further reduce the number of sessions requiring an update. Assuming that each player has a different view of the game world, the backend needs to compose different state updates for each player.

In the third step (composition), an algorithm is used to compose the state updates for each of the players based on their perspective of the world. If needed, developers can also customize the method implementing this algorithm (`composeStateUpdate()`) so that additional game-specific information can be added to the state updates. For instance, players may choose to

inject additional, globally-available information to the state updates, such as the state of the weather, time, resources, and more. Once these updates are composed, they remain in memory until their distribution to the clients. Algorithm 5 shows the default process used to compose state updates:

Algorithm 5: The default algorithm used for the composition of a state update.

```
Data: wsMap (Map of World sessions to List of Entities)
Data: suBuilder (state update builder introduced in step 1)
Data: rTerrain (option to fully refresh terrain)
Data: rEntities (option to fully refresh entities)
Result: Composes a state update by optionally refreshing the terrain and/or entities.
1  suMap ← Map();
2  forall w in wsMap do
3      if rTerrain then
4          suBuilder ← checkAndRefreshTerrain(w.key, suBuilder);
5      end
6      if rEntities then
7          suBuilder ← refreshEntities(w.key, suBuilder);
8      end
9      isuBuilder ← suBuilder.clone();
10     player ← Get player with ID w.key.playerID from DB;
11     cpEntities ← List all entities for player.id and w.key.worldID from DB;
12     forall uE in isuBuilder.updatedEntities do
13         if uE.playerID == w.key.playerID then
14             forall pE in cpEntities do
15                 if State.Entities.isOutOfAOI(uE, pE) then
16                     newTerrain ← getTerrain(uE.position, ue.aoi);
17                     forall t in newTerrain do
18                         isuBuilder.addUpdatedTerrain(t.value);
19                     end
20                 end
21             end
22         end
23     end
24     response ← UpdateStateResponse();
25     response.status ← OK;
26     response.stateUpdate ← suBuilder;
27     suMap[w][response];
28 end
29 return suMap;
```

The final step in the state update process is the distribution of the composed updates to their intended clients. In the previous steps, Athlos offers default implementations using various data structures and algorithms that can define, filter, and compose state updates. However,

the final step is technology-specific and therefore must be implemented manually. To enable the distribution of the state to the clients, Athlos defines three methods that control which clients can receive the update:

1. The `sendUpdate()` method can be used to send a state update to one or more clients based on the result of the `filterUpdate()` method.
2. The `multicastUpdate()` method sends a state update to one or more clients, without implementing any filtering.
3. The `broadcastUpdate()` method sends a state update to all connected and subscribed clients.

Depending on the context, developers may choose to utilize either of these three methods to distribute state updates to the players. For instance, events that influence all players regardless of any other factors may be broadcasted to all connected players. In other cases, it may be more beneficial to multicast a state update to a group of players – for example the players of a specific team. For most cases, however, developers should opt to use the first option, which is by far the most efficient in terms of resources. Despite being considered a single architectural component, the state update mechanism, summarized in figure 4.23, is dispersed in code. This is mostly due to the fact that its four processes are considerably different from each other and are initiated at different points in the runtime. By working together, these components can enable MMOG backends to distribute state updates to the clients using specific technologies. For instance, in the IaaS or dedicated approach, developers may use bi-directional streaming services provided by gRPC to send state updates to specific clients who are subscribed to a world. These subscribers can be internally managed using default methods provided by the backend, or manually managed by implementing custom routines. In serverless backends, developers may opt to use special APIs that enable bi-directional communication through WebSockets or other HTTP/2 technologies. The use of such APIs is limited only to a handful of serverless environments. For environments that do not provide such options, developers can opt to use third-party services like Ably or Pusher, or discard the use of the specialized state update mechanism completely, using polling techniques as a final resort.

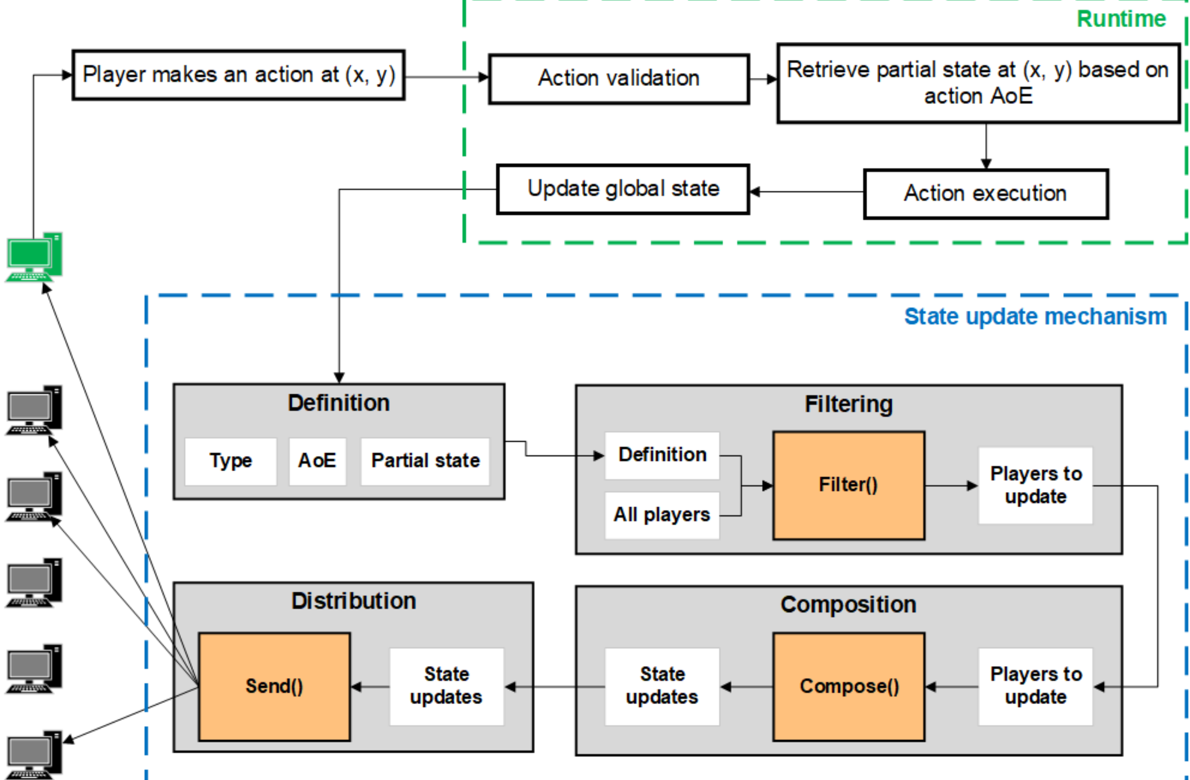


Figure 4.23: An overview of the state update process, involving the use of the state update mechanism.

Runtime scalability

Apart from upscaling the state of their worlds, MMOG backends may also have to support increasingly large workloads. The ability of an MMOG backend to increase or decrease its workload capacity is defined as *runtime scalability*. Runtime scalability can be achieved by expanding the provisioned set of resources within the same computing node – known as *vertical scaling*. From a software engineering and IT management standpoint, this is a relatively cost-effective solution that benefits from less maintenance concerning both hardware and software and does not require any complex communication protocols. For instance, if a game provider uses a single VM instance to power their MMOG backend, they can simply allocate more resources (such as more vCPUs or RAM) to their VM to handle increased demand. While being a relatively simple process, this entails adjustments in VM configurations that must be made manually, and thus the system must be monitored and actively managed. In addition, vertical scaling induces a single point of failure in the system, as all operations and data are held on a single server. This increases the chances of losing data if a failure occurs or suffering from downtime when there is a need to maintain the server. In the context of MMOG backends,

this approach severely limits the upgrade options, especially as the procured VM reaches its maximum capacity. While these limits are relatively generous in public clouds, going to the extremes entails significantly higher costs which can be prohibitive for the majority of game studios.

Runtime scalability can also be achieved by increasing the number of nodes in a system so that more resources become available – known as *horizontal scaling*. Horizontal scaling is significantly more complex compared to vertical scaling due to the need for virtualization technologies, load balancing, and other software to manage and maintain the infrastructure. However, this type of scaling is more resilient to downtime or failures. The existence of multiple nodes means that switching a single node off will not cause the system to suffer from downtime. Similarly, a failure on a single node may simply cause the system to direct traffic to other nodes while the problem is being resolved. From a software engineering standpoint, horizontal scaling entails the use of techniques that make use of computing on multiple nodes effectively. Therefore, game developers must be able to reason with the parallel execution of events on multiple nodes, which share a consistent, persistent, and unified state.

The proposed framework is designed to work with both vertical and horizontal scaling. Vertical scaling requires little effort in terms of software design, as the software components described previously can be incorporated within the same computing instance. In cases where a single, relatively powerful node is being utilized to host the runtime of an MMOG backend, developers must be able to execute code in parallel where necessary to improve performance. To this end, the Athlos API defines the `GameServer`, an abstract class that can be used to instantiate a server service running on a single node. As this approach is intrinsically tied with IaaS/Dedicated infrastructure, the `GameServer` class uses gRPC to handle the network traffic and provide services to clients. By default, gRPC handles connections to the server by instantiating new threads for each channel and is therefore capable of serving multiple requests simultaneously. To enable the execution of background tasks, the `GameServer` also includes a special *background execution thread*, which allows developers to execute code in the background at specific intervals. The use of the game server is complemented by the `DedicatedGameClient` on the front end, which includes the supporting functionality for concrete game clients to communicate with servers. While this approach can work well with dedicated and IaaS infrastructure and vertical scaling, `GameServer` instances can be launched on multiple computing nodes

at the same time. The concrete implementations of these definitions also allow their use in containerized environments, such as those created using Docker, and deployed using Kubernetes. In such cases, the server runtime can be deployed on multiple computing nodes, while additional public cloud services can be used to provide distributed persistence. This offers the ability to deploy game servers that utilize the IaaS approach while leveraging the advantages of horizontal scaling. Even though this is supported within the framework, such products, and services are not explored or evaluated as they remain outside the scope of this thesis.

On the other hand, serverless approaches are inherently designed for horizontal scaling. In such cases, developers do not have any control over how instances are scaled or which instance handles which part of the workload. Therefore, serverless backends must be designed in a way that accommodates stateless, isolated services. In the proposed framework, this is facilitated through the use of multiple, isolated service endpoints which, unlike in IaaS, are not controlled by a centralized runtime. In such cases, the workload is handled on a per-request basis, complemented by the use of several cloud-based services for persistence, background task execution, bi-directional communication, and more.

Complementing such runtimes are the provisions made by the cloud providers for dynamic resource allocation, scalability, and load balancing. In serverless environments, such tasks are handled internally by the cloud provider, with game developers being given little control and customization options over these aspects. For instance, Google's App Engine (PaaS) allows developers to define configurations of the runtime environments to control scalability and load balancing attributes such as the number of minimum and maximum instances, the instance types, target resource utilization, and more. This favors a more streamlined development approach that focuses on game logic rather than creating resource provision and load-balancing algorithms. Products like Cloud Functions (FaaS) employ a very limited set of options, which primarily concern the number of minimum instances to avoid cold starts. On the other hand, the Athlos framework supports runtime scaling for dedicated or IaaS environments by allowing developers to run multiple instances of dedicated game servers. Each dedicated game server can be set up to have its own set of responsibilities and areas to handle, allowing the game to scale its runtime across multiple computing nodes. Dedicated game server software components can further be managed with the use of containerization and container orchestration systems like Docker and Kubernetes, which are provided as Container as a Service (CaaS) offerings.

4.5 Tools

The previous sections have presented a variety of models and methods which target the development of MMOG backends on commodity clouds. These models and methods provide the foundations with which scalable MMOG backends can be engineered to run by leveraging resources provided by either IaaS or serverless cloud infrastructures. However, these propositions are purely theoretical and do not offer a concrete way of translating a game design, its architectural components, and its subsequent methodologies into concrete implementations. To allow developers to leverage the proposed models and methods, a set of tools is devised. Using these tools, developers can rapidly prototype MMOG backends, benchmark their performance, and deploy them on public cloud infrastructure. These tools are broken down into several categories: the Athlos API and its subcomponents, the project editor, the code generator, and tertiary tools for persistence, security, and world generation.

4.5.1 The Athlos API

The Athlos API is the centerpiece of the proposed framework. It defines a set of abstract, reusable software components that enable the development of scalable MMOG backends. The API is divided into four namespaces: `core`, `server`, `serverless`, and `client`, and is utilized in every Athlos project.

The `core` namespace includes common elements that are utilized by every other package. The first of these elements are data model abstractions, which are defined using interfaces. These interfaces define the behavior of default model classes, such as worlds, entities, players, and more. These interfaces are extended by game-specific, concrete class definitions, but also enable the framework to provide an abstraction layer by offering processes that utilize these interfaces in a game-independent way. Secondly, the `core` package includes abstractions for the persistence layer and serialization process through the definition of DAO interfaces and Protocol Buffer conversion interfaces like `Modelable` and `Transmitable` discussed in sections 4.4.4 and 4.4.5. In addition, the `core` includes various exception classes and the definition of an abstract event manager class which can be used to execute events on both servers and clients. Meanwhile, the `server` namespace includes elements needed to create dedicated server instances. The `GameServer` class defined in this package provides various abstractions for server manage-

ment. These are extended by concrete implementations to define game-specific game servers. Other features include support for logging, abstractions for interactions with the persistence layer, background task execution, and more. Similarly, the `serverless` namespace contains abstractions that allow developers to create a serverless backend. A major component of this namespace is the `AthlosService`, discussed in section 4.4.6. This is further expanded by provisions for the utilization of various serverless environments, such as Google's App Engine, Cloud Functions, and more. These provisions improve the development effort by implementing utilities and patterns that abstract technology-specific components. The `client` namespace includes various abstractions related to the creation of client applications. It defines the generic `GameClient` class which includes definitions for functions that enable clients to communicate with their corresponding backends. This is extended by the `DedicatedGameClient` and the `ServerlessGameClient`, which define protocols and methods of communication with dedicated and serverless backends. Within these definitions, clients also support logging, state management, and background operations.

The Athlos API and its components are utilized in their default form, at the lowest layer of the framework. These facilities are complemented by additional elements which are dynamically generated based on game definitions and can be further extended by developers to support even more complex functionality where needed. As a proof-of-concept, a prototype version of this API is implemented in Java 8, for both dedicated and serverless technologies. The API's components are imported into Java projects using the Maven project automation tool through automatically generated project configurations. Alternatively, they can be imported manually within any project through Maven or by using Java archives (JARs).

4.5.2 Project editor

As discussed in section 4.4.1, the proposed approach makes use of technology-agnostic game definitions which allow developers to design MMOGs by defining the data models, relationships, and configurations that make up a game. These are incorporated in *game projects*, which can be used to generate concrete, technology-specific implementations. In this section, the Athlos Project Editor is introduced, which allows developers to create and manage game projects reliably and efficiently, and ultimately utilize the underlying tools to generate boilerplate code. The editor provides facilities to manage game projects through a Graphical User Interface

(GUI), shown in figure 4.24.

Initially, developers may use the editor to create new projects. During the creation of a new project, developers are asked to select from various options. These include their intended server and client environments, the type of world, the name of the game, and more. Most of these properties can be changed after the creation of the project, allowing developers to make corrections or adjust to shifting needs. When a project is created, the editor creates a new game project file that stores the definitions and configurations of the created project. One of the first tasks during the game design process is the definition of the game model, by customizing and extending the default model presented in section 4.3. During the definition of the game model, developers may also define new, custom data types or enumerators, using the facilities provided by the editor. Existing model types can be extended by creating additional attributes within their models, whereas new types can be created by specifying a type name, and the associated attributes of the type. To enable this functionality, the editor allows developers to select from a list of available types for each attribute, while also internally managing data type references, and validating the use of various types under different contexts.

The editor also facilitates the creation of game APIs by allowing developers to define actions, services, and request-response models. These definitions are completely approach-independent and only provide the model for a service – its name, URI, and corresponding request-response models. Based on the URI of each service, the editor categorizes services in a tree-like pattern in its main project view, allowing developers to quickly inspect their game's API. A useful tool provided by the editor is the ability to automatically create a default API for the game, which features services that are commonly found across a variety of MMOGs. Some examples of these services are those related to authentication and world and state management. An expansion of this feature further allows developers to automatically create data management services. Data management services can be used to provide Create, Retrieve, Update, and Delete (CRUD) operations as services for each of the types defined in the project, including those manually created by the developers. The automated definition of management services aims to reduce development time, as defining all of these services is a relatively menial, repetitive, and time-consuming task.

Further to these, the editor allows developers to change some core game attributes, like the `MAX_CELLS` constant as well as namespace names, PB class names, and so on. Another feature

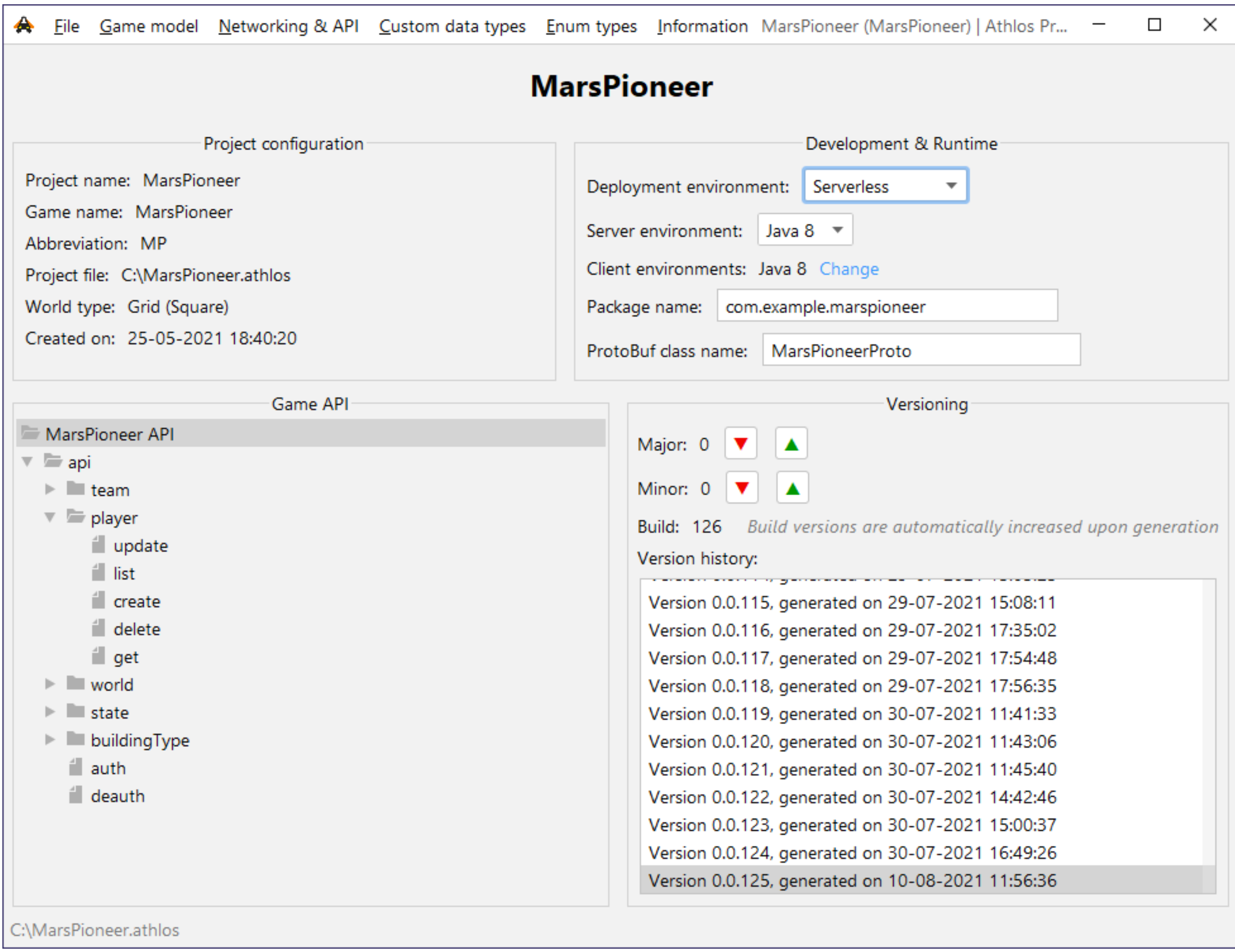


Figure 4.24: The Athlos project editor (prototype).

of the editor is the ability to control game versions. Developers can adjust the version of the game, through major and minor revisions, whereas the editor automatically tracks builds and increases the build number automatically every time a successful code generation is completed. Project files are saved as JSON-formatted text using the .athlos extension. These files can be inspected and edited manually using any text editor, although this is heavily discouraged as it can easily lead to corrupt definitions. Files created through the editor can be included in version control systems or communicated to other developers who can inspect them, modify them, and generate the same boilerplate code on their workstations.

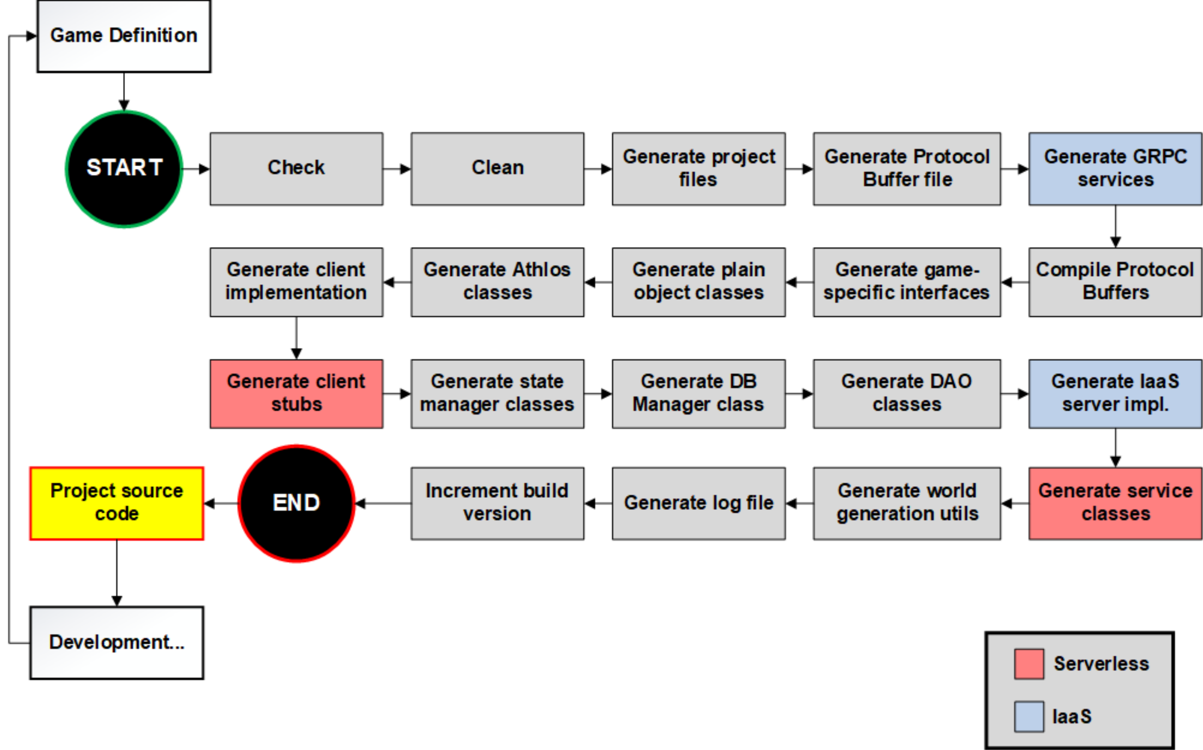


Figure 4.25: An overview of the generation pipeline – the processes involved in the generation of boilerplate code in MMOG projects.

4.5.3 Code generator

The project editor software tool is instrumental in the process of game design as it can enable the quick creation of approach-independent game definitions. However, these definitions alone do not offer any concrete way of developing an MMOG backend. The job of converting these definitions into code is handled by the *code generator*. The code generator is an independent software tool that is coupled with the project editor. The generator parses game definitions created using the editor and then generates concrete MMOG projects containing boilerplate code that is ready to run. Developers may subsequently start development, or go back to revise their game definitions in the editor and generate a newer version of the boilerplate code.

The complex nature of the code generation process combined with the need to facilitate a large variety of approaches at each stage of development makes the code generator one of the most intricately designed components of the Athlos framework. At a high level, the generator works by splitting its tasks into a series of processes. These processes are executed sequentially within the *generation pipeline* shown in figure 4.25.

The structure of the code generator relies on a highly diversified set of abstracted parts, which

are responsible for generating different parts of the game project, for different programming languages and runtime environments. At the same time, various techniques are employed to enable the generated code to function properly within the context of an MMOG backend, and to allow instantly-executable code upon generation. Figure 4.25, defines several steps which are involved in the generation of both dedicated/IaaS and serverless projects – represented with grey color, while some processes are specific to the IaaS/dedicated approach (blue), and others are specific to serverless (red). For instance, the Protocol Buffer generation stage is a common step in both environments and involves the generation of a Proto file. This file is automatically populated with PB and service definitions based on the game's model. Where applicable this step is followed by the generation of gRPC services for IaaS/dedicated projects. A PB compiler is used internally by the generator to generate concrete implementations of the PB definitions in this file, based on the programming language selected, thus hiding most of the complexities of this mechanism.

A problematic case during the generation process is the polymorphous nature of the extensible data types defined in the model. In these cases, using PB works against major software engineering principles like OOP, disabling the use of inheritance and polymorphism. To work around this problem, a special step is introduced within the generation process, which is complemented by the interfaces defined in the Athlos API. During this step, game-specific interfaces are created, which are subsequently implemented by their sub-types, The use of interfaces makes it possible to generalize a group of extensible types – for example entities – and allow developers to manage them collectively. Similarly, the conversion of PB and plain-object classes also involves adjustments in the codes generated by the corresponding stages. To ensure that these classes can be used interchangeably, the generator reads the previously-generated PB files and finds the proper location to inject code that facilitates their conversion to a plain-object type. Conversely, during the generation of a plain-object type, the generator implements the necessary methods to ensure that it can be converted into its corresponding PB type. Both of these processes work by first reading the attributes of the models created in the game definition and then generating code that allows their conversion.

The generation of state and database management classes are also important steps during the creation of a project, as these classes are mostly game-specific. Developers can obtain the state of the game using state management classes and interact with the persistence layer using

database management classes, making them valuable tools in the development process. The creation of state management classes involves the creation of dynamically generated code that is highly dependent on the programming language used, the infrastructure type, and the type of world selected, making its generation one of the most complex parts of the generator. To create the database management tools, the generator first creates a Database manager class that instantiates several DAOs. The classes for these DAOs are implemented in the next step and are based on the database access policies defined in section 4.4.4.

When the IaaS/dedicated approach is being used, there is a need to generate a gRPC server with specific extensions which allow it to work as a part of the Athlos framework. These servers are generated using gRPC plugins and have their own built-in memory caches that can be used to persist data locally if needed. For serverless projects, this step is omitted, and custom service classes are generated instead. These classes implement the `AthlosService` described in section 4.4.6, provide an intuitive way for developers to create service logic, and call these services without dealing with the specifics of each service container technology. For both IaaS/dedicated and serverless projects, the corresponding types of client stubs are generated for each approach to allow clients to communicate with the backend. Towards the end of the pipeline, the generator creates utility classes that aid the creation and generation of world states for the specified game. With the help of procedural generation tools, which will be described in section 9.F.3, developers can create theoretically infinite worlds which can expand as the players explore them. The generator subsequently outputs a *generation log* which details the generation process that took place, allowing developers to review it in case there is an error and debug their project. Finally, once the generation is completed, the generator increments the project's build version. The build version is used for version control, as well as to make sure that the generation process does not replace any previously generated code. The entire process of game definition and then code generation culminates in a boilerplate project that is ready to run but must be implemented to include game-specific elements and logic.

The structure of an Athlos project is outlined in figure 4.26. The Athlos API and standard model are used as a blueprint for developing new projects and can be extended within a game definition. The code generator is used to create a concrete game implementation based on the selected infrastructure, server, and client environments. Some components within the generated projects either feature common functionality or can be generated through the previously defined

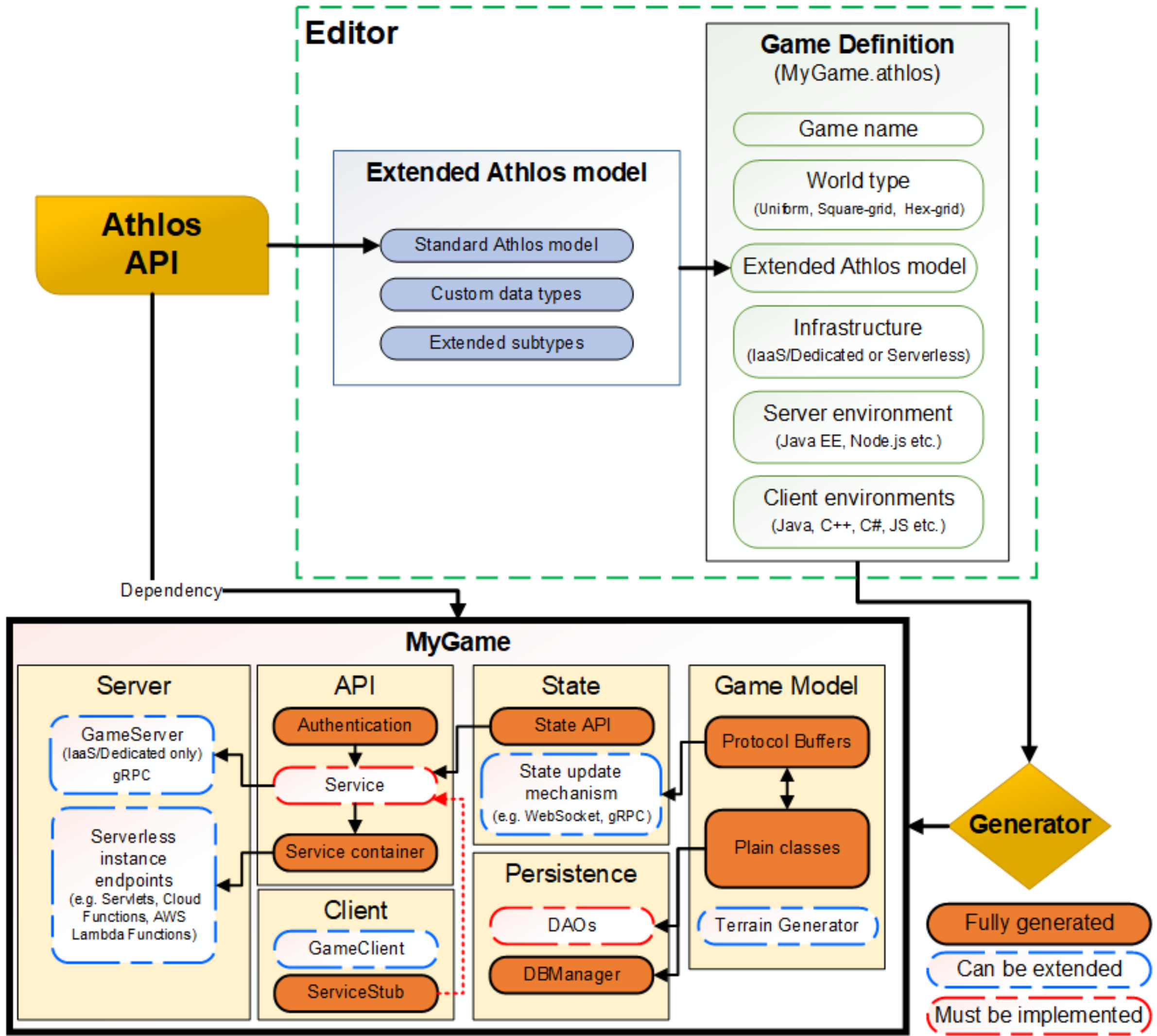


Figure 4.26: The structure of an Athlos project.

model, and are therefore fully generated. These are illustrated with an orange background in figure 4.26. Other components which are implementation-specific, such as services or DAOs are generated but it is left up to the developers to realize their functionality. Finally, some components are divided into parts that can be automatically generated, while some other parts are left unimplemented. For instance, the terrain generator contains abstractions that facilitate the generation of terrain but does not implement the process of generation itself as this is a game-specific process. Developers must extend the functionality of such mixed components to create a fully functioning MMOG backend. While many of these components and parts are generated automatically, developers have full control over the code and can even customize components that are fully generated.

4.5.4 Guide

One of the most useful tools for developers who are new to the Athlos framework is the framework's guide. The Athlos guide is a web-based tool that provides support and documentation for game developers. It includes a large body of content related to downloading, installing, and using the tools described in this section as well as detailed documentation of the concepts, models, and methods utilized to handle a variety of development aspects. While the guide is still a prototype, the content provided within it can help developers understand these concepts and utilize Athlos to quickly prototype scalable MMOG backends. In the future, the guide may evolve to include tutorials that can help developers practice these concepts under guidance, and show how different types of games can be created using examples. Furthermore, the guide can provide a platform for discussion, problem-solving, suggestions, and include news or other content such as interactive video tutorials.

4.5.5 Libraries

The Athlos API, project editor, and generator are the core tools of the framework and allow MMOG backends to be modeled, designed, and then implemented using specific technologies. While these core tools are sufficient to develop and deploy these backends, Athlos provides several tertiary tools to handle various aspects of the game development process, aiming to further reduce project complexity, effort, and time required to produce such applications. These involve handling data persistence, serialization, networking, security, as well as terrain generation. Some of these tools are embedded within the framework's projects and are also used internally by Athlos to carry out various tasks, while others can be optionally imported when specific approaches or technologies are used. The use of such tools further aids the development process – albeit only for a specific set of approaches. Nevertheless, it provides the groundwork for the future development of similar tools to support an expanding set of approaches and technologies. While these tools provide software engineering value to the proposed framework, they are of secondary significance for this thesis as they are not related to its research objectives. These are described in Appendix 9.F and evaluated in Appendix 9.G.

4.6 Conclusions

This chapter presented a suite of novel models, methods, and tools for developing scalable MMOG backends, which are incorporated in a software development framework called Athlos. Apart from dealing with the requirements set forth by the third research objective of the thesis, this chapter also sets a precedent to answering the hypotheses and addressing several technical challenges which were identified in sections 1.3 and 3.7.

The Athlos framework is based on a novel dynamic model which improves the development process through code reuse – thus enabling better code maintainability and modularity. It is believed that this model, through its abstractions, can support a very wide set of game types and their requirements, in contrast with existing models in other frameworks which are fairly limited and incompatible with other technologies.

Furthermore, the proposed approach solves several technical challenges which previously hindered the development of scalable MMOG backends on commodity clouds. For instance, novel methods such as the abstraction of state management, persistence, communication, and serialization through the State and Persistence APIs, the State-update mechanism, and the use of Protocol Buffers aim to provide standardized solutions to many problems seen in the development of MMOG backends – thus allowing developers to focus on game logic rather than dealing with these issues. This may ultimately streamline the development process and lead to higher quality and quantity of MMOGs. Unlike the in-house solutions presented in section 3.4, chunk-based state representation and other methods aim to unlock the full potential of cloud-based services (such as datastores), and enable MMOG worlds to reach massive scales.

Finally, this chapter also presented the Athlos API, the centerpiece of the framework which defines the software architecture used to construct a wide variety of MMOG backends. This API supports the development of different types of games and their deployment on a variety of infrastructures while following the constraints of the framework and utilizing the proposed methods. Finally, these are complemented by a set of tools, allowing developers to rapidly prototype MMOG backends while using specific technologies.

Chapter 5

Case studies

> "*With proper design, features come cheaply. This approach is arduous, but continues to succeed.*"
>
> Dennis Ritchie

5.1 Introduction

The previous section describes a set of models, methods, and tools which facilitate the development of scalable MMOG backends on commodity clouds. The proposed framework, Athlos, incorporates these concepts and implements tools that enable game developers to utilize them to create MMOG backend prototypes. While there are some practical elements, most of the work described in the previous section is theoretical and does not provide any insights into the suitability of the proposed approach. To explore the suitability of the proposed approach in enabling MMOG backends to run on commodity clouds and to investigate the questions that were raised in chapters 1 and 3, this section reports on the development of three multiplayer online games using Athlos. The implementation of these case studies aims to establish a proof-of-concept, thus verifying that the framework is suitable and capable of enabling the development of a variety of MMOG backends. Secondly, the implementation of multiple game backends further challenges the proposed approach, aiming to reveal any weak points that may need to be addressed in future work. Thirdly, the developed cases foster the thinking process required to handle various technical challenges and to define the limits and scope of the approach. Most importantly, these proof-of-concept implementations are also used to evaluate

the framework's performance, scalability, and code maintainability, with the aim of addressing key research objectives and questions.

5.2 Case study 1: Mars Pioneer

The first case study is *Mars Pioneer*, initially introduced as a motivational concept behind the development of an abstract game model in section 4.2.2. The game's concept is based on the RTS genre, where players can control multiple entities at the same time, collect resources, construct buildings and units, and compete with other players to meet game objectives. While most Real-Time Strategy games run in rounds under limited time and bounded spaces, Mars Pioneer takes some elements from the Turn-Based Strategy genre as well and incorporates a more persistent type of world. The game features a single, fully-persistent, square-tiled world in which the joining players must develop their Mars colonies by building up bases and constructing infrastructure to gather resources. This can be achieved by using an initial set of resources that are available to them to expand out of their initial base and gather more resources. The overall objective of the game is to compete against other players by obtaining more resources that help players rise through the ranks. Even though there is never a clear winner as the game can theoretically run forever, the winning party is considered the player who manages to exert a stronger influence in the game based on a variety of factors, such as their number of resources, gathering rate, colony size, and more.

5.2.1 Development

The first step in developing the game using the proposed approach is to select a type of infrastructure on which the game will be deployed. For this particular game, a serverless backend is selected for two main reasons: Firstly, there is a need to support scalable states, which can be better managed using an elastic option. Secondly, this type of game does not require very low latency. Athlos currently provides the tools to implement serverless backends for three different environments: Google's App Engine Standard, App Engine Flexible, and Cloud Functions. App Engine Flexible is chosen for this scenario because it offers the ability to use bi-directional communication through its WebSocket API, whereas the other two approaches lack this fea-

ture. After defining the world type, game name, and other properties, the next step is the definition of the game's core elements through the project editor. While the Athlos model itself is inspired by this same game concept, it is discovered that extra data items need to be defined within the model of this specific game. For instance, to model buildings and resources, two new custom data types are defined with their respective names and attributes shown in figure 5.1. New enumerator types are also introduced to model the types of buildings (`HUB`, `FARM`, `WELL`, `SAND_PIT`, and `MINE`), and their level of research – with higher levels leading to a more plentiful collection of resources. Terrain types are also defined using an enumerator, having several possible values as shown in figure 5.1. A single entity class called `Building` is defined which models buildings that exist within the game world. To allow players to interact with the game, several actions are also defined which allow them to construct buildings or sell them (e.g. `BuildFarm`, `BuildHub`, `SellBuilding` etc.). Several modifications are also made to the default types. For example, the `MPPlayer` class which extends the Athlos-defined `Player` includes several additional attributes such as a resource set, the last time of resource collection, and more. A game API is also defined, most of which consists of default management services that are automatically added using tools in the project editor. The services and actions defined for this case study are summarized in figure 5.2. After completing the initial stage of the game's definition, a project is generated using the project editor's tools, which execute the code generator to generate boilerplate code. The game's code utilizes Java 8, and is organized into an IntelliJ IDEA project consisting of three packages based on the framework's structure: `core`, `app-engine-flex`, and `client`. An additional package is later added for simulation purposes. The game definition and implementation of Mars Pioneer are hosted on GitHub[1].

Mars Pioneer uses the framework's internal implementations of Protocol Buffers to serialize information. Messages are communicated to and from the backend using web services for the defined services, or by establishing WebSocket connections for game actions. This allows low-frequency calls to services to be efficiently served while maintaining high-speed, bi-directional links between the client and server for in-game actions. To enable persistence, Mars Pioneer uses GCP's Cloud Firestore as a database, and Cloud Memorystore to access a Redis-based cache. Background operations can be also executed using GCP's Cloud Tasks. These public cloud services work in unison to provide strongly-consistent access to the game's world, while also maintaining a backup of the state. To implement the DAO interfaces related to the persistence

[1]https://github.com/nkasenides/MarsPioneer

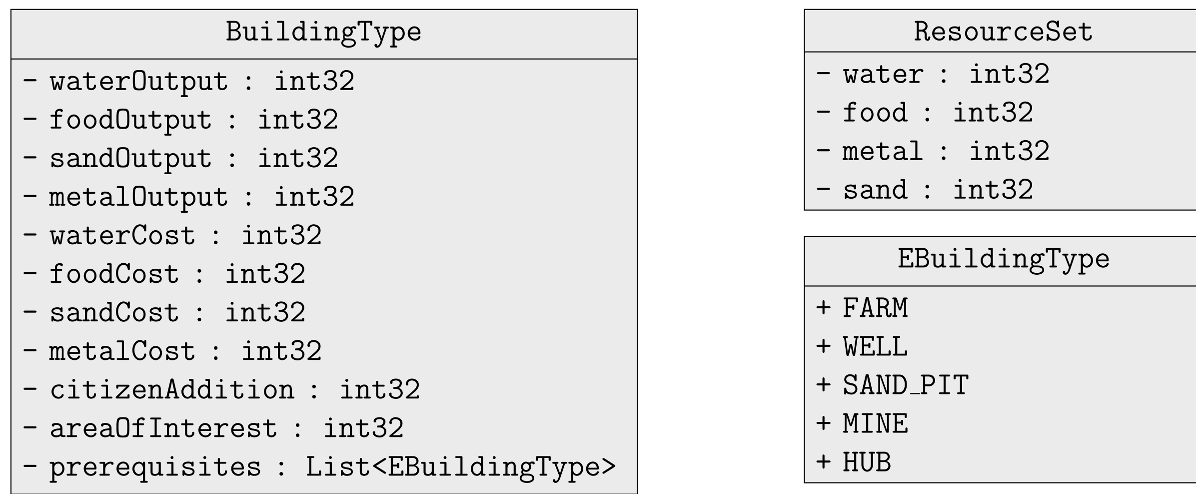


Figure 5.1: Custom classes defined in Mars Pioneer.

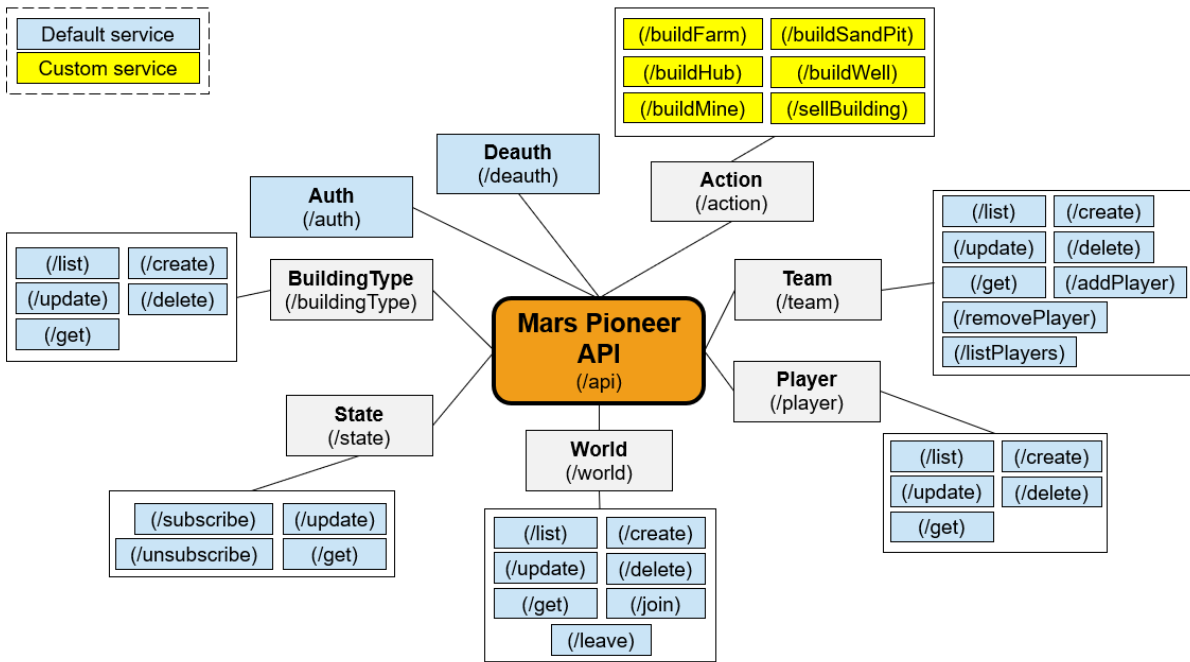


Figure 5.2: The API defined for Mars Pioneer.

API, the Firestorm and Objectis libraries are used. The use of these persistence options and tools is demonstrated through an example in Appendix 9.D where the implementation of the world session DAO is included in listing 9.1. At other points, customizations are necessary to make sure that additional game properties are included within the communicated state – such as in the retrieval of state snapshots. Listing 9.2 in Appendix 9.D shows how a snapshot is modified to include a resource set, which is a game-specific property. Meanwhile, player actions are handled by their corresponding WebSockets in a method called `handleMessage()`, as shown in Appendix 9.D, listing 9.3. In this example, several logical steps are implemented when a build action is taking place. Firstly, the session is verified through the framework's internal Authentication API. Once the session is verified, the state API is used to retrieve a snapshot of the game state at the location of the action. Using the retrieved state, a set of resource and terrain-building rules are enforced. If all of these conditions are passed, resources are deducted and the building is constructed by creating an entity in the game state, which is subsequently updated to reflect the action made. Finally, the state-update mechanism functions within the state API are used to define and disseminate a state update. On the client side, communication with the backend is implemented through the generated service stubs. An example in Appendix 9.D, listing 9.4, shows how the `SellBuildingStub` is implemented to communicate with its corresponding WebSocket to handle any responses by printing messages on the client's UI. The client is also responsible for visualizing the game's state using simple 2D graphics on a canvas, as shown in figure 5.3. The client also allows players to interact with the game by moving through the world, selecting positions, and constructing buildings using keyboard and mouse inputs. Finally, it plays an instrumental role in the simulations described in section 6.3, as it manages bots that act as players to record synthetic benchmarks.

5.2.2 Impact on framework

The development of Mars Pioneer as the first case study has had a significant impact on the development of the framework. Firstly, its successful development and deployment signal the suitability of the framework in creating scalable MMOG backends that can be deployed and executed on commodity clouds. Secondly, it proves, albeit to a limited extent, that commodity clouds can be viable options for the deployment of such applications. This experience has also led to a better understanding of many design problems that were defined previously. For

Figure 5.3: A screenshot of the Mars Pioneer client program, presenting a visualization of the game state to the client.

example, the implementation of the case study led to the adoption of a de-coupled service architecture, mentioned in section 4.4.6. During the development of this case study, a problem was found with regards to this specific approach: WebSocket-specific implementations were incorrectly designed because service logic was included within the service containers – i.e. the WebSocket classes themselves – thereby fusing logic to the used technology and going against the design principles of the framework. This design flaw was discovered relatively late during the development of the case study and therefore was intentionally kept to avoid errors, but was later fixed by making the necessary amendments to the framework's code generator. The experience of developing Mars Pioneer has also led to the discovery of many previously-unknown problems, causing the proposed approach to take a generational leap forward. Several methods presented in section 4.4 such as the concepts of snapshots and modifiables, and the definition of various state-update mechanism stages were introduced because of problems that arose during this case study. Such solutions offered the ability to improve the performance, scalability, and code maintainability of projects developed using the proposed methodology. The development of Mars Pioneer also helped to shape the project editor and generator as these software tools were still at their very primitive stages at this point, with only a trivial set of features and only supporting a very limited set of approaches. Most of the supporting libraries were also developed during this initial case study, out of the necessity to expedite development.

While many improvements were introduced, some other, more complex problems that were identified still remain unsolved. During the development of Mars Pioneer, 125 project code generations were made, most of which involved small, incremental changes to the game model and services. Since neither the project editor nor the generator can automatically merge newer code generations with prior implementations, code had to be merged manually where necessary by keeping game-specific code written in older versions and manually adding components from newer code generations. Fortunately, this was not a big problem as most new components were independent of existing ones. The generation of newer versions on top of existing implementations is a natural process, as it is very likely that developers will either need to adjust existing functionality or implement additional features after their initial attempt. However, the current approach of manually merging different versions is somewhat problematic as it may lead to mistakes and possibly accidental loss of code. In a more complex project that involves collaboration between multiple developers, the chances of making such mistakes can increase significantly due to miscommunication. This presents a new challenge and invokes a new question: *How can the*

generation and merging of new code with previously implemented game-specific logic be better handled – or ideally completely automated – to ensure a consistent code base between versions of the same project?. This question is not specific to MMOG backends and is related to the broader area of software engineering and version control. As this problem is out of the scope of this research, it is reserved for further, future work.

5.3 Case study 2: aMazeChallenge

The second case study is aMazeChallenge, an existing educational programming game (Kasenides & Paspallis 2021). aMazeChallenge is a turn-based, multiplayer, maze-solving game that was initially created as part of an undergraduate research project. The game aims to teach basic programming concepts (i.e. conditionals, loops, functions, etc.) to high school and early university students by first training them in these concepts and then having them program an avatar to escape a maze using a block-based language. The main objective of the game is to escape the maze with the most points possible.

5.3.1 First version

The first version of aMazeChallenge, originally developed in 2018, used a backend hosted on Google's App Engine Standard and featured an Android-based client. Communication between these two components was achieved using JSON-formatted messages that are communicated through web services implemented using Java Servlets. This was based on the client-server model, where a request received by the backend is executed to make adjustments to the game state. To persist the state, the project used Memcache, a cloud-based cache that is integrated with App Engine. Other pieces of information, such as game sessions and player information were persisted using Google's Cloud Datastore. Finally, the game uses a long-polling approach to update the state of the clients in 1-second intervals. Once connected to a game, clients would utilize this method to individually request updates to their states using HTTP requests.

aMazeChallenge is quite different compared to conventional multiplayer games. Its gameplay does not entail the direct control that players have over their characters or entities in commercial online games. Instead, players must create code that is uploaded before the start of each game.

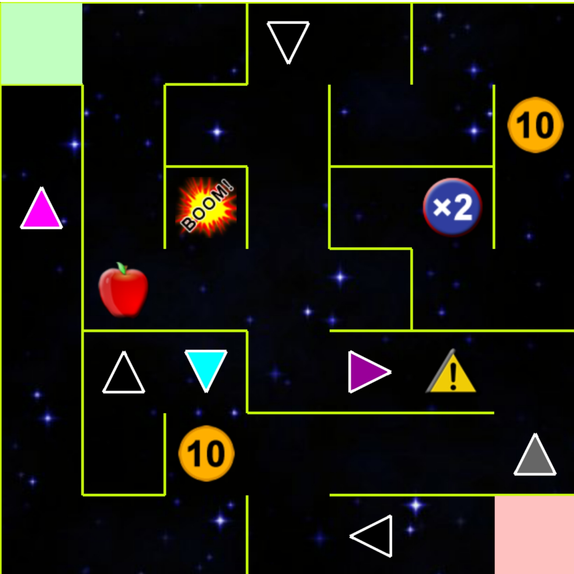


Figure 5.4: A screenshot of the aMazeChallenge client during a student competition, held at UCLan Cyprus in 2021.

The code written by the players determines how their avatar will behave during the game – i.e. what type of action will be executed based on the conditions within the maze. This type of indirect control requires the use of various facilities, such as join and play queues to accommodate gameplay. When players submit their code, they are placed at the back of a queue, and their code is executed in order of submission. For each turn in the game, the runtime executes the code of all players sequentially, applying an action based on their code, and then updating the game's state. Figure 5.4 shows an example of a simple maze, in which players (indicated by colored triangles) have to traverse the maze from the start position (red) to the exit (green) while interacting with various types of objects such as bombs, traps, fruits, and coins. These objects may affect the player's state once they are interacted with, by adjusting their points or health status. The square-grid world type is a natural fit for aMazeChallenge's worlds as entities can only exist in specific, cell-based locations within the maze. These game worlds are generated randomly using various pathfinding algorithms to create the maze walls. The objects within the world are also generated automatically at random times based on a seed and the level of difficulty. This case study aims to deconstruct the main components of aMazeChallenge to determine if such a game can be developed by using the proposed methodology. This explores the applicability of Athlos to an existing platform and investigates whether Athlos has the potential to satisfy the requirements of mobile and web-based games.

5.3.2 Development

The project editor was used to create a new game project. Various custom types are defined within the game model, the most important of which is the `Challenge`. The challenge type defines the properties of a maze challenge – such as its name, difficulty, maze wall, and background colors, the state of the grid, and more. The challenge itself is not playable and only describes the properties of a game level that will facilitate that challenge. Game worlds are instantiated using a specific challenge's properties, and can then be joined by players. Within the challenge is another custom class called `Grid`, which models information about the grid of the challenge – its starting and finishing locations, width, and height in terms of cells, and the data representing the walls of the grid. Another major type is the `Game`, which includes attributes that help the game manage players in queues and execute turns. The `EventQueue` type is used to manage events that may occur during the game and communicate their occurrence to the players. The game model also features a large set of enumerators for identifying maze generation algorithms, colors, audio, images, difficulty, language, and more. Two types of entities are defined within the model. The `PlayerEntity` represents a player's avatar, whereas the `PickableEntity` represents objects that may be randomly generated within the grid with which the avatars can interact. No actions are defined for this particular game, as the players do not have direct control over their avatars. Various other classes within the default model are further customized to enable gameplay, the most notable of which are partial states. In this type, the state of the grid, other players, and events are included in addition to the default attributes.

The game's API consists of mostly default services, as shown in figure 5.5, but also several custom services that enable the submission of code by the players and of post-game questionnaire responses. A special runtime service is also defined but cannot be accessed by the clients. This special service handles turn-based gameplay in all game worlds and is executed automatically using CRON jobs every second. The size of the maze grid is limited from 5x5 up to 30x30 cells as smaller mazes would be too easy to solve and larger mazes would not be ideal to visualize. In addition, the number of players joining the same maze is limited to avoid congestion in these limited spaces. The limited scale of the game state in aMazeChallenge offers an opportunity to move away from the default State API defined by Athlos and towards a more efficient, customized solution. To avoid additional state retrieval queries, the state of the grid and entities

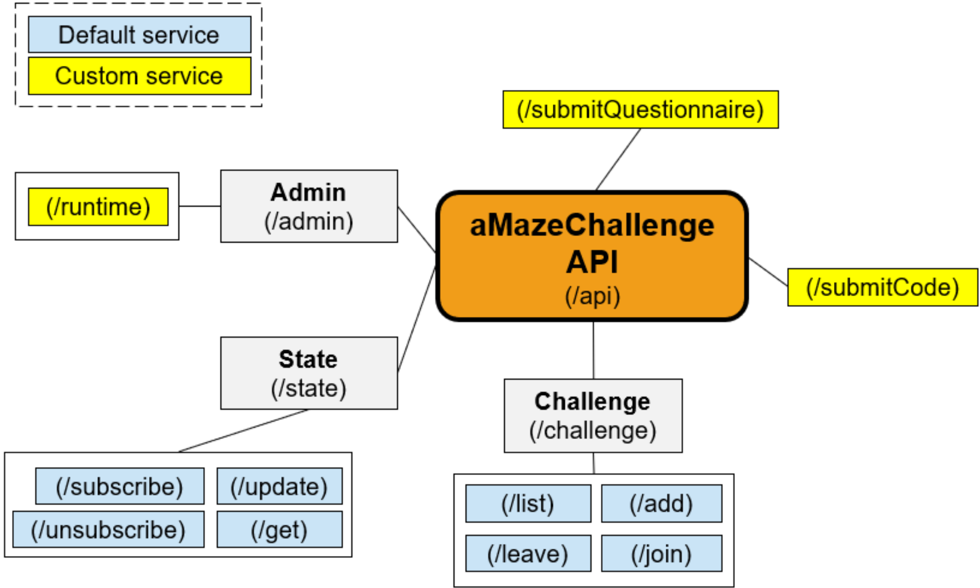


Figure 5.5: The game API defined in the new version of aMazeChallenge.

are included within the game objects themselves. This is shown in appendix 9.E, listing 9.5, which shows the code for retrieving the state of the game. For all tested scenarios, which will be explored in section 6, this approach managed to achieve good performance. The logic included in-game services was copied from the original version of aMazeChallenge and adapted to work with Athlos-based features like Protocol Buffers within a newer App Engine project. In terms of persistence, the new, Athlos-based version of aMazeChallenge is upgraded to utilize Google's Firestore which enjoys lower latency, while the default caching option is kept. The new version also makes extensive use of the persistence API by setting DAO policies and implementing various DAOs for each of the objects defined. This is illustrated in line 23 in listing 9.5, as well as in listing 9.6. While the upgraded version still uses long polling for the state updates, it also implements an alternative pub/sub messaging system using Ably that is disabled by default to avoid extra charges. In the future, Firestore's real-time update mechanism may be used to update the state without the need for third-party services. The state update mechanism itself is underutilized within this project due to the limited scale of the terrain and the number of players. The project's source code is hosted on GitHub[2].

[2]https://github.com/nkasenides/aMazeChallenge2.0

5.3.3 Impact on framework

The second case study also has significant effects on the framework. Firstly, it confirms that the proposed approach does not only work with its conceptualized MMOG but can model and realize other games as well, including ones that already exist. The implementation of aMazeChallenge as an Athlos-based MMOG was challenging at first because of the game's unique gameplay style and mechanics – which are very different from those normally encountered in commercial MMOGs. However, the modular architecture of the proposed methodology, in conjunction with a dynamic model allowed for the implementation of aMazeChallenge with relative ease. This case study also confirms the framework's compatibility with web and mobile technologies. In addition, it opens up opportunities to also explore other types of technologies in games, such as the Internet of Things (IoT), Augmented, and Virtual Reality (AR, VR).

The impact of this case study on the framework comes from several challenges that had to be faced during its development. Firstly, there was a need to facilitate player-independent events occurring within the game world. This includes events that generate in-game pickable objects, as well as audio events that signal actions made during multiplayer sessions. The latter presented a major challenge as multiplayer audio events were a new feature that was not included in the original version. This challenge led to the design and adoption of the *event model* and *mechanism*, which are used in aMazeChallenge to launch game-specific events at specific times, as well as to disseminate multiplayer audio events to players during an online session. This event mechanism was then abstracted to provide an interface with which events can be created, managed, and executed at specific times without the involvement or participation of players, not just for aMazeChallenge but for any type of game.

This upgraded version of aMazeChallenge also raised the need for tools with which MMOG backends can be administered. For instance, aMazeChallenge could benefit greatly by using an administration panel, which offers tools for managing the backend's runtime and player sessions, exploring the data layer, offering content distribution, creating, exporting, and importing challenges, and more. Such operations are very useful during the game's runtime and especially during competitions as they can enable the resolution of various issues by providing easy access to information. The implementation of the aMazeChallenge administration panel also offers a better understanding of their uses and common features and serves as potential groundwork to offer them in other types of games. aMazeChallenge has also raised the need for various other

commercial services, such as scoreboards and match-making, which are important features of many MMOGs. The standardization of such services and their adoption within the framework may add extra value to the proposed methodology, raising the level of abstraction, and further expediting the development of these applications.

5.4 Case study 3: Minesweeper

The third case study is an implementation of the Minesweeper game first presented in section 3. During the feasibility study, this particular game was selected for several reasons including its simple set of modeling requirements and rules. Despite its simplicity Minesweeper still presents some challenges. The first challenge, which was fully addressed in the feasibility study, was the conversion of the original, single-player game to run as a multiplayer game. This uncovered many of the challenges and requirements of multiplayer online games and laid the groundwork that would be later used to develop the proposed methodology. The second challenge, which is addressed within this case study, is how to convert the state of such a game so that it can be scaled to massive sizes and support as many players as possible. In the feasibility study implementation, the state of the game (aka. the board) is modeled using a matrix structure that includes the states of each cell. Players can join a game world and play cooperatively with others on the same board by issuing various actions. The architectural components are rather simple and involve the use of web containers to serve client requests. To serve a request, JSON-formatted text is communicated from the backend to the client whereas the state is persisted using various cloud-based data stores. Meanwhile, state updates are sent to the clients using Ably, but there are no standardized facilities for defining, filtering, composing, or disseminating them. The aim of this third case study is the reconstruction of the initial version of Minesweeper, by first defining the elements of the game within an Athlos project and then attempting to leverage the proposed methods and tools to scale the state of the game well beyond the limitations that were encountered in the feasibility study. Furthermore, it aims to evaluate the feasibility and applicability of the IaaS/Dedicated approach, as the new implementation is based on this type of infrastructure.

5.4.1 Development

As with the two previous case studies, the project editor is used to define various game elements. Several custom types and enumerators are defined, mostly related to the state of cells within the game (i.e. the `RevealState`), as well as the `GameState` and `Difficulty`. In this implementation, the players can change their partial state size which allows an exploration of how the size of the partial state affects performance and scalability in an MMOG backend. This is implemented by varying the size of the partial state across various configurations. For Minesweeper, no entities are identified as it is a game featuring only board-based states. Several services and actions are defined within its API, which are summarized in figure 5.6. These are used to manage worlds, allowing players to join them, subscribe to them for updates or issue actions to reveal or flag a cell within the board. As the project is based on the IaaS/Dedicated approach, the game's API is implemented as gRPC services which can be called by the client using their respective stubs. These services are unidirectional and follow the request-response model, with only one of them (state update) being bidirectional. The state update service, which is a default bi-directional Athlos service, uses gRPC streams to distribute the game state to the clients in real time. This case study also utilizes many components that implement the proposed methods. For instance, the Persistence API is used as an interface between the data layer and the logic and presentation layers. In combination with a Redis cache within GCP's Cloud Memorystore and the methods proposed for the scalability of game states, this enables the creation and storage of very large game boards. As the state of the game reaches very large sizes, the State API is also used extensively to manage the game state efficiently, while the state update mechanism is given an opportunity to show its true potential under heavier loads. Finally, the client program is adapted to include the graphics from the feasibility study implementation – shown in figure 5.7 – while also offering the ability to issue player-specific partial state sizes, communicate with the gRPC server, and utilize PB classes as established by the proposed methodology. The code for this project is open-source and hosted on GitHub[3].

[3]https://github.com/nkasenides/minesweeper-athlos

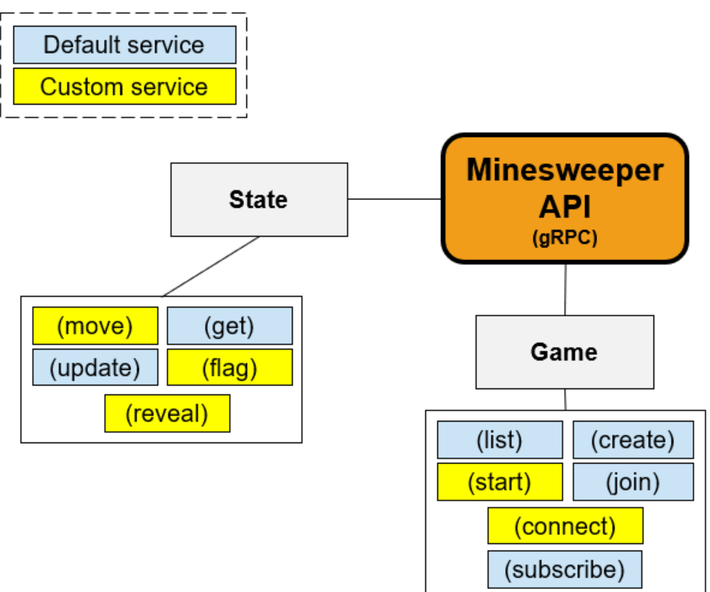


Figure 5.6: The game API defined for the Minesweeper case study MMOG.

Figure 5.7: The GUI presented by the Minesweeper client during a simulation using a 10×10 partial state size.

5.4.2 Impact on framework

Despite being the simplest out of the three case studies presented, Minesweeper's implementation using Athlos further establishes the ability of the proposed approach to handle different types of games through its default model, as well as the tools that it provides. This case study did not present any significant challenges during its development, which is perhaps due to its relatively simple set of features and gameplay. Nevertheless, it enables a comparison between approaches that do not offer any special facilities for improved performance or scalability – such as the ones used in the feasibility study – and the proposed methodology which claims to feature such support. Therefore, experiments can be designed to evaluate whether the Athlos framework truly meets the expectations set forth by the research objectives of this thesis. The design of these experiments and their results are reported in section 6.

5.5 Conclusions

The process of developing the case studies described in this chapter is an initial evaluation of the proposed approach, as it provides key insights into its feasibility and applicability. As observed from these case studies, Athlos is capable of modeling, designing, and producing projects for three different games and allows their deployment on public cloud environments with relative ease. The experience of developing these case studies establishes the suitability of the proposed approach in developing MMOG backends with different requirements and utilizing different technologies. These case studies have had a profound impact on the proposed framework itself as the valuable insights they provided led to major improvements in terms of features and fixes. This was achieved mainly due to the discovery of several weak points during development that led to amendments in the proposed approach. Some of these issues are still unresolved and may be handled in future research. Ultimately, the case studies presented in this section enable the investigation of several challenges that were posed in sections 1.3 and 3.7. The reflections made upon the hypotheses and these technical challenges are discussed in the analysis presented in chapter 7.

Chapter 6

Evaluation

> *"No amount of experimentation can ever prove me right; a single experiment can prove me wrong."*
>
> Albert Einstein

6.1 Introduction

In chapters 4 and 5 a novel software development methodology is presented and then utilized to develop scalable MMOG backend prototypes running on commodity cloud platforms. The development of these case studies has had a profound impact on the methodology itself and enabled an observational evaluation of the proposed approach. While these case studies prove the usefulness of this approach in a variety of contexts, they do not study important aspects of MMOG backends, such as performance, scalability, development effort, and code design, which have an important role in their effectiveness as software systems.

In this chapter, the proposed software development framework and its related tools are quantitatively evaluated using the case studies implemented and described in chapter 5. While these case studies have already explored some challenges to a limited extent, the hypotheses of this research are still left unexplored. The evaluation of the proposed methodology through several experiments aims to explore these hypotheses, and ultimately provide quantifiable evidence that MMOG backends can (a) be developed for a variety of environments in commodity clouds, (b) sustain a satisfactory level of performance, (c) achieve the necessary scalability that may allow them to be hosted as commercialized products, and (d) be developed efficiently and effectively.

6.2 Evaluation strategy

The evaluation of the proposed approach is based on four aspects – *performance*, *scalability*, *development effort*, and *code maintainability*. These aspects are used to evaluate the models, methods, and tools presented in chapter 4 within the context of the three case studies discussed in chapter 5 as well as other isolated, targeted experiments.

The first part of the evaluation first identifies the operations involved during the execution of various game services and evaluates their performance. This aims to obtain useful information about which of these operations and stages in the processing pipeline are the most performance-intensive, with the aim of guiding further improvements in the proposed methodology as well as concrete game implementations towards optimizing these services. Furthermore, it studies the effects of various backend and experimental configurations on *latency* as a function of the number of active players. This can reveal how latency – by far the most important performance indicator in MMOG backends – is affected by different parameters, and how the MMOG backends developed using the proposed approach perform under different loads. The boundaries between the evaluation of performance and scalability in such systems are blurry, as these two aspects are intricately related and have direct effects on each other. It is argued that performance in MMOG backends is synonymous with runtime scalability, as these systems must be inherently scalable. However, the experiments designed to study *state scalability* within this research are different and are therefore described separately. In terms of state scalability, the proposed methodology is evaluated based on how well the Athlos framework can produce MMOG backends that can support massive, expandable states that can be managed efficiently and distributed effectively. State scalability experiments are designed to run in isolated environments, aiming to study how the case studies respond to increasing state sizes, and to measure their absolute size and performance. The Athlos framework is also evaluated with respect to the effort needed to develop MMOG backends and compared with other development methodologies. Finally, the evaluation explores the readability, maintainability, and design quality of the code produced by the framework using various software design metrics. In each of the described experiments, the targeted hypotheses and challenges are identified with the aim of explicitly addressing them in chapter 7. This chapter is further divided into five sections. The first four sections evaluate the framework through the case studies described previously, whereas the last section attempts to evaluate the usefulness of various tools using isolated experiments.

6.3 Performance and runtime scalability

This section describes experiments that aim to measure the performance and scalability of MMOG backends developed using Athlos, mainly addressing hypotheses H1-H4.

The related works have established *global response latency* as the single most important factor related to the performance of an MMOG backend, as it has direct effects on the QoE perceived by the players and encompasses a large set of other factors like processor and memory usage, network speed, and more (GauthierDickey et al. 2004, Jardine & Zappala 2008, Burger et al. 2016, Dhib, Boussetta, Zangar & Tabbane 2016, Dhib, Zangar, Tabbane & Boussetta 2016). In the context of this research, *global response latency is defined as the time elapsed from the moment a player's input is received until the reception of an update that assimilates that input.* This delay includes the time taken for a request to be communicated from the client to the server, processed, a response to be received by the client, and then parsed into an actionable format. This communication process comprises a variety of steps that add up to form the global response latency. For instance, the network distance and speed greatly impact global response latency as they determine how fast – or slow – a message can be communicated back and forth. In addition, the processing power and load of the computing nodes in the network, including both clients and servers directly affect this metric. Slower or busier client devices may take longer to serialize a message into a format that can be communicated to the server, and conversely, to de-serialize a message coming from a server into a format that can be used to present feedback to the player. These factors are circumstantial, as the performance of client devices or the network can greatly vary based on the types of devices being used – i.e. smartphones vs personal computers – or the network's conditions.

Global response latency is undoubtedly useful and plays an important role in determining the performance of a specific deployment approach, such as those studied in chapter 3, or a specific implementation of an MMOG system. However, in the context of evaluating the MMOG backend development methodology described in chapter 4 and all the methods and tools it entails, using global response latency may skew data based on the performance of individual client devices or the network, potentially yielding less useful, or even invalid results. To solve this problem a different metric is used – *backend processing latency* – which is defined *as the time taken for an MMOG backend to process a request upon receiving it, create, serialize, and*

send a response back to the client. Given the context of this thesis, this is stated hereafter as *processing latency*. Processing latency is used to eliminate factors that are not associated with the performance of the backend – such as the network or client device performance or conditions, or even other unforeseen external factors. The elimination of the delays incurred by these factors allows a more isolated evaluation of the framework's methods and tools and ultimately increases the usefulness of the data recorded.

The first experiment in this evaluation uses the Athlos implementation of Mars Pioneer – which is the most complex out of the three case studies – to measure the performance of various services employed by its backend. For comparison, the same backend is deployed on a relatively powerful local machine as well as GCP's App Engine Flexible environment. Reflecting on hypothesis 1 from a purely theoretical standpoint, it is expected that the locally-hosted backend will have significantly lower latency compared to the cloud-hosted backend at low numbers of players, mostly due to the employment of less powerful instances in the cloud at these small scales. Nevertheless, this trend is expected to reverse as the number of players increases, eventually causing the locally-hosted backend to run out of resources, become overwhelmed by the increasing demand, and eventually suffer from a huge spike in latency that renders the system non-operational. On the other hand, the cloud-based backend is expected to scale by employing more computing nodes, therefore allowing it to deal with increasing demand for much larger numbers of active players.

To carry out this experiment a simulation harness is developed to enable large numbers of players to be simulated as bots that can join and play concurrently within the same world. The harness can also simulate various configurations with changing variables and conditions during the experiments and thus enables synthetic data to be recorded. For the purposes of this experiment, the game assigns a very large number of in-game resources to these bot players allowing them to carry out operations at a much faster pace than with human players, which in turn allows experiments to be expedited. Player bots are programmed to pick actions randomly based on a predefined set of available actions and construct buildings at locations where they determine fit – i.e. where there is available space for construction based on their locally-perceived states. As the game is developed in Java, the simulation harness uses the facilities of the same language to create various types related to the simulation. For example, the behavior of bots, as well as the data recorded by each bot during the simulation are encapsulated within

the `Bot` class. This class is programmed to run as an independent thread of execution, thus allowing multiple bots to run concurrently on the same device. These bots are managed by the `Simulation` class, which defines simulation-specific attributes and how the bots will be instantiated, managed, and stopped gracefully so that meaningful data can be retrieved at the end of each simulation. The simulations conducted using this harness can support a variety of configurations and variables. The `SimulationConfig` class, which encapsulates the different parameters of each simulation, allows the definition of variables such as the time limits of the simulation, the delay between actions by each player, and a list of events that may occur during the simulation. Events are subsequently modeled using `SimulationEvent` and can either signal a player joining or leaving. Theoretically, these could be extended to also include other types of events in more complex simulations. Simulation events allow different configurations in simulations by allowing them to define when a player will join or leave the game. For example, many types of configurations can be created, such as a *linear* configuration where players continuously join the game at a stable pace, a *spike* configuration where players will suddenly join the game at approximately the same time, or a *flat* configuration where a certain number of players joins the game but remains constant throughout the experiment. The simulation harness also allows several variables to be adjusted, such as the total number of players to join the game, their joining rate (in players/second), the delay between player actions (in milliseconds), the duration of the simulation, and more, all of which may affect the results. For the purposes of this experiment, the flat configuration is selected as it allows the inspection of the latency as a function of the number of players by keeping the number of players constant in each run. Multiple runs can then be executed with different numbers of players, allowing the observer to determine how the number of players affects latency.

Several factors must be kept in control to improve the validity of the results obtained from these experiments. Firstly, the device and network utilization are kept as similar as possible and at or near idle conditions. Even though the performance of these two does not directly affect the backend, it may do so indirectly as the simulation is initiated from a client device and calls have to travel across the network to reach the backend. The location of the data center used for the cloud-hosted approach is also kept the same throughout all runs. Furthermore, the database, cache types, and data policies used are also kept identical, and the App Engine deployment configuration and server environment remain unchanged. Between the two deployments, the same server environment is utilized and both backends deploy services featuring identical logic.

The locally-hosted experiments used a computer running Windows 10 with a 3rd generation Intel Core i7 processor and 16GB of RAM, out of which 10GB are allocated to the backend. The cloud-hosted backend runs on Google Cloud's App Engine Flexible, using the default F1 instances and with a configuration that employs 4 vCPUs and 4GB of memory on each instance, and uses automatic scaling with a target utilization of 65%. This configuration also limits the number of instances to 4, due to budget limitations. To run the simulation clients, the experiments used a computer running Windows 10 with a 7th generation i5 processor and 6GB of RAM, of which 4GB are allocated to the simulation.

Various factors are kept constant throughout these experiments. For instance, the joining rate of players is kept constant at 1 player per 500ms, and the duration of each player's gameplay is calculated based on the total number of players to join so that all players are given a chance to join the game with time left to spare for the last joining player to execute meaningful interactions with the backend. This duration is calculated using the formula $d = p \times i + 10000$ where d is the duration, equal to the number of players p, times the joining interval i, with 10,000 milliseconds added at the end to allow actions from the last joining player. Lastly, a random delay between player actions is used, which ranges between one and two seconds in order to prevent the players from taking actions simultaneously.

Base latency

As a prelude to the main phase of the experiment, the *base latency* of each approach is measured using inert services as initially described in section 3.6. Through this preliminary experiment (E0), a baseline is established in the form of global response latency, helping to negate the influence of network-induced factors from this experiment as well as to evaluate the performance of the two infrastructures without any implementation overheads. The locally-hosted service is deployed on a machine within the same network as the requesting device, whereas the cloud-hosted service is deployed on Google's Europe West 6 data center in Zurich, approximately 2,500km away from the client device. Out of 20 measurements taken 10 seconds apart, the locally-hosted service yielded an average latency of 4.9 milliseconds, whereas the App Engine Flexible (cloud-hosted) service took an average of 63.95 milliseconds to respond.

Global response latency components

The first experiment (E1) attempts to investigate the different stages at which latency is introduced in the system in the form of global response latency. The complicated nature of MMOG backends, and by extension the Athlos framework itself entails various components that together make up the global response latency of a system, including:

1. **Input latency** (client) – the amount of time taken for user input to reach the system. For example, the polling rate of a physical keyboard can affect input latency. For the experiments described herein, input latency is ignored as it is not related to the backend's latency.

2. **Local rule processing** (client) – the time taken for the client software to validate a subset of the game's rules that are checked locally. This is ignored as none of the case studies in this thesis feature any local rule validation.

3. **Request formation** (client) – the time taken for the client device to form a request upon receiving an input.

4. **Request serialization** (client) – the time taken to convert a request object into a serialized format so that it can be communicated across the network.

5. **Ingress network latency** (network) – the time taken for the information to be transmitted from the client to the server.

6. **Request de-serialization** (server) – the time taken to convert a serialized request back to an object that can be used to perform an operation.

7. **Action processing** (server) – the time taken for the backend to process an action described by the request.

8. **Response formation** (server) – the time taken for the backend to create a response object.

9. **Response serialization** (server) – the time taken to serialize a response object into a communicable format to be sent to the client.

10. **Egress network latency** (network) – the time taken for the serialized response to be transmitted from the server back to the client.

11. **Response de-serialization** (client) – the time taken for the client to de-serialize a response into an actionable object.

12. **Presentation** (client) – the time taken for the client software to render the state of the game based on the response it received. The presentation step is ignored in the simulations as none of them render any graphics.

To determine the time taken to run through each of these steps, an experiment is designed using the `GetState` service within the Mars Pioneer case study. The retrieval of the state is a uni-directional service that allows the inspection of all of the steps described previously. This service is meant to be called once, after the player joins a world, in order to retrieve a base partial state for the player. By contrast, state updates are called continuously and are bi-directional services. Their nature makes the inspection of these components harder, as they involve server-initiated events. The state retrieval service is also considered more useful than other single-call services for this experiment because it carries the state of the game, making it the most heavyweight out of the uni-directional services. In this process, steps 1, 2, and 12 are ignored as they are irrelevant, whereas steps 6 through 9 are identified as the *backend processing latency* defined above. The network-induced latency (both ingress and egress) is also ignored in this experiment, as it is deployment-specific and does not offer any insights into the performance of the system. To determine the time taken for the system to go through each of the defined stages, the client and backend of Mars Pioneer are adjusted to output time measurements on each of these elements. In addition, other processes which are not relevant to this particular experiment, such as the subscription of clients to worlds, and state updates are removed. The backend is deployed on a local server as described previously and tested using a single player bot requesting state information. The experiment is repeated 5 times, with averages calculated out of all runs.

The results, shown in table 6.1, indicate that the average total time taken for the system to go through the cycle for this specific service is 175.40ms. The stages studied in this experiment can be combined into four groups: *request/response object formation*, *serialization*, *de-serialization*, and *action processing*. As shown in figure 6.1, the most time-consuming stage in this pipeline

	Run 1	Run 2	Run 3	Run 4	Run 5	Average (ms)
Request formation	2	2	1	2	2	**1.80**
Request serialization	0	0	0	0	0	**0.00**
Request de-serialization	1	1	1	1	1	**1.00**
Action processing	141	94	148	95	93	**114.20**
Response formation	2	2	2	1	2	**1.80**
Response serialization	3	3	4	3	3	**3.20**
Response de-serialization	52	53	54	52	56	**53.40**
Total (w/out network)	201	155	210	154	157	**175.40**

Table 6.1: Results showing the time taken to run through each of the identified stages in the backend's request-response cycle.

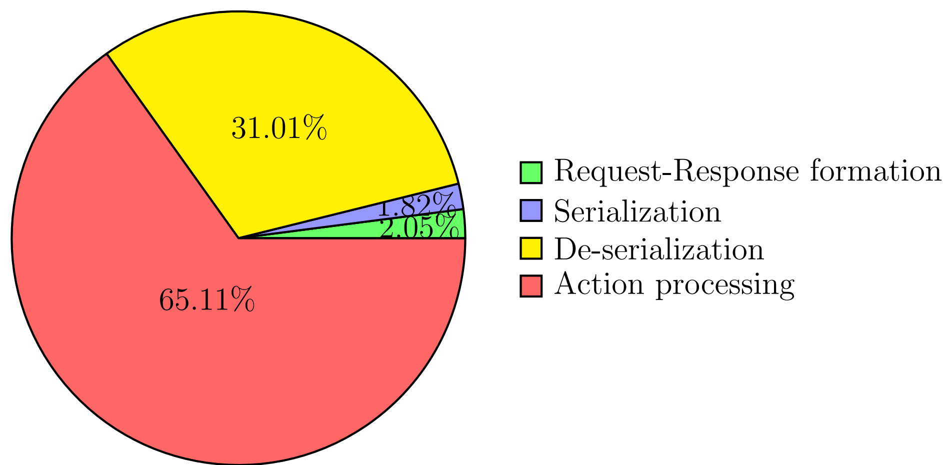


Figure 6.1: The percentage of the total global response latency taken by different stages in the request-response cycle

of processes is the *action processing* stage, taking 65.11% of the total time. This is followed by the de-serialization of the state at 31.01%. These two stages together take up more than 96% of the total time taken in the cycle, and are therefore the most important steps — deserving the attention of the framework's methods and tools, and subsequently of the developers of MMOG backends. Action processing, which is by far the most time-consuming of the two, is a game-specific process that involves various operations itself – which must be explored further to analyze their impact on performance. Consequentially, this highlights the importance of performance-oriented code design in the services of MMOG backends, and the use of efficient serialization and de-serialization tools.

Processing latency

The second performance experiment (E2) divides the measurement of processing latency into five different groups based on the sub-processes carried out during the action processing stage of various action services in the game – e.g. `BuildFarm`, `BuildMine` etc. The first process is *session validation*, which verifies that the world session used to make the request is valid and active. This step mostly entails an interaction between the backend and a datastore used to retrieve the world session. Following this is *state retrieval*, in which the backend first computes the elements of the partial state it must retrieve, and then interacts with a cache or datastore to retrieve it. Based on the state retrieved, the rules of the game are applied to the action made by the player, in a process defined as *rule processing*. Once the action is validated against the rules, the necessary *state modifications* occur to reflect the action made by the player. During this step the runtime interfaces with the datastore or cache to update the persisted state. Meanwhile, the state is distributed to the players through the state update mechanism based on the action's AoE – known as *state dissemination*.

This experiment aims to determine the time taken for each of these operations to complete, giving insights into how the action processing stage can be optimized. The experiment entails the use of the Mars Pioneer's action services to update the state of the game, which is then disseminated back to the clients through the state update service using persistent web socket connections.

To conduct the experiment, several configurations are created with varying numbers of players starting from 5 players, 10 players, and then incremented by 10 up to a maximum number of players that is determined by the capabilities of the system. Each configuration is executed three times for both local and cloud-hosted approaches and measurements are recorded for each of the action processing operations identified above. The data was recorded by adding code in the services and measuring the time elapsed in each sub-process using timestamps. The average processing latency is calculated for each sub-process individually as well as for all of the entire service combined.

The results, shown in table 6.2 and graphically demonstrated in figure 6.2 for the locally-hosted dedicated backend and in table 6.3 and figure 6.3 for the cloud-based backend can be used to identify which of these sub-processes are the most performance-intensive. Both approaches

appear to yield similar results, with the most demanding sub-process within the gameplay of Mars Pioneer being state sending/dissemination. In the locally-hosted deployment, this sub-process takes an average of 66% of the total service latency, whereas in the cloud-hosted deployment, it takes a slightly lower percentage of the total time (64%). The second most demanding sub-process appears to be state retrieval. In the locally-hosted backend, this took an average of 29% of the total service latency across various configurations. It also took an average of 32% of the total time in the cloud-based backend. These two processes appear to be the most demanding sub-processes in this particular implementation with the most likely cause for this intensiveness being their interactions with the persistence layer. For instance, the state distribution sub-process has to retrieve the state based on each client's perspective, whereas the state retrieval process uses queries to retrieve a partial state of the backend necessary to carry out an operation. Furthermore, the time taken by the state distribution sub-process might further be increased by the need to serialize data before it is sent to the clients. Similarly, during state retrieval, the system must organize the retrieved state data into a *contextual*[1] partial game state. Out of the other sub-processes, session validation contains interactions with the database that are minor in terms of data size compared to those in state retrieval and distribution, and state modifications only make minor updates that require relatively lightweight operations. Meanwhile, other operations like rule processing, session validation, and state modification take a significantly lower portion of the total time as they are much simpler in complexity and involve mostly logical operations that are less detrimental to performance. These results underline the need for both MMOG development frameworks such as Athlos and game developers to focus on optimizing these two specific sub-processes, which appear to be the cause of most processing latency.

In terms of absolute performance, the results presented in table 6.2 show that the average total processing latency of the locally-hosted approach grows by an average of $\times 2.07$ for each 10-player increase. In most of the sub-processes involved, latency increases by an average factor of about $\times 2$ for each increase in 10 players, except for rule processing which increases at a lesser rate ($\times 1.4$). This can be explained by the fact that the rule processing operation is far less resource-intensive and scales more efficiently mostly because it does not require any interactions with the database, whereas all other operations do. From the results of the locally-

[1] A contextual item is defined be an item with a certain context. In this particular example, it is the context of a game world, which helps provide perspective for a particular action being made.

Processing Latency (ms) - Locally-hosted Mars Pioneer backend						
Number of players	**Session validation**	**State retrieval**	**Rule processing**	**State modification**	**State send**	**Total**
5	1.03	24.11	0.16	2.89	43.97	72.01
10	0.91	41.26	0.04	2.39	101.48	146.08
20	1.25	51.63	0.05	2.95	113.08	168.97
30	3.94	141.36	0.04	7.33	399.62	552.29
40	16.97	359.08	0.03	31.99	808.74	1216.81
50	39.17	614.30	0.17	87.93	1334.77	1137.00
60	64.20	942.85	0.02	133.98	2091.96	3233.01

Table 6.2: The processing latency in terms of milliseconds, recorded for various sub-processes of a play service in a locally-hosted version of Mars Pioneer.

Processing Latency (ms) - Cloud-hosted Mars Pioneer backend						
Number of 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

Table 6.3: The processing latency in terms of milliseconds, recorded for various sub-processes of a play service in a locally-hosted version of Mars Pioneer.

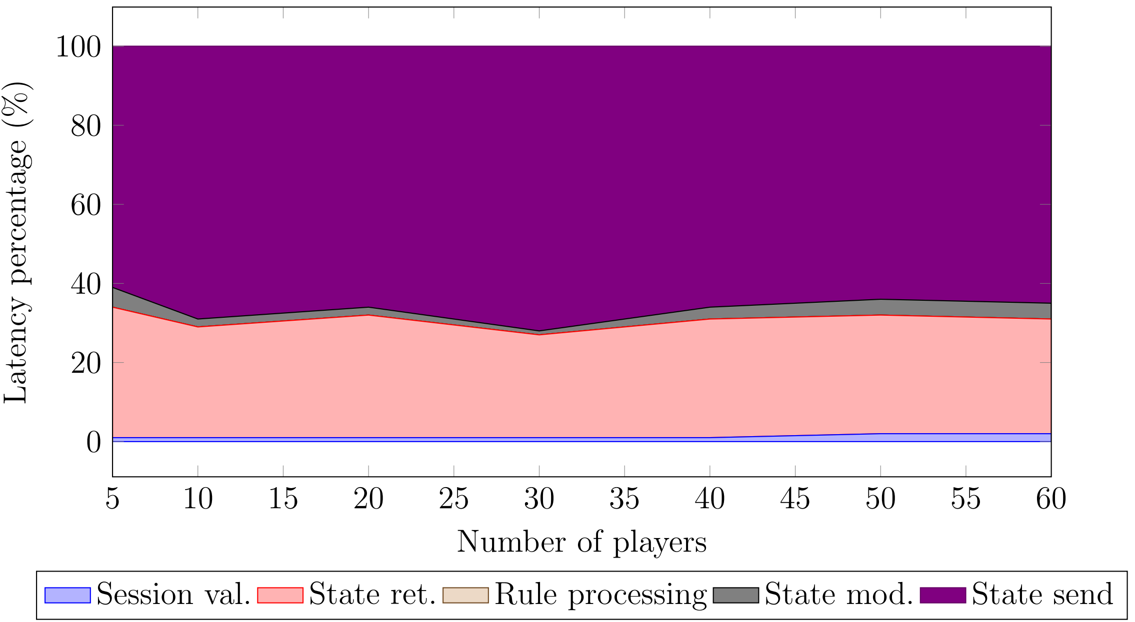


Figure 6.2: Sub-process latency as a percentage of the total service latency in the locally-hosted version of Mars Pioneer.

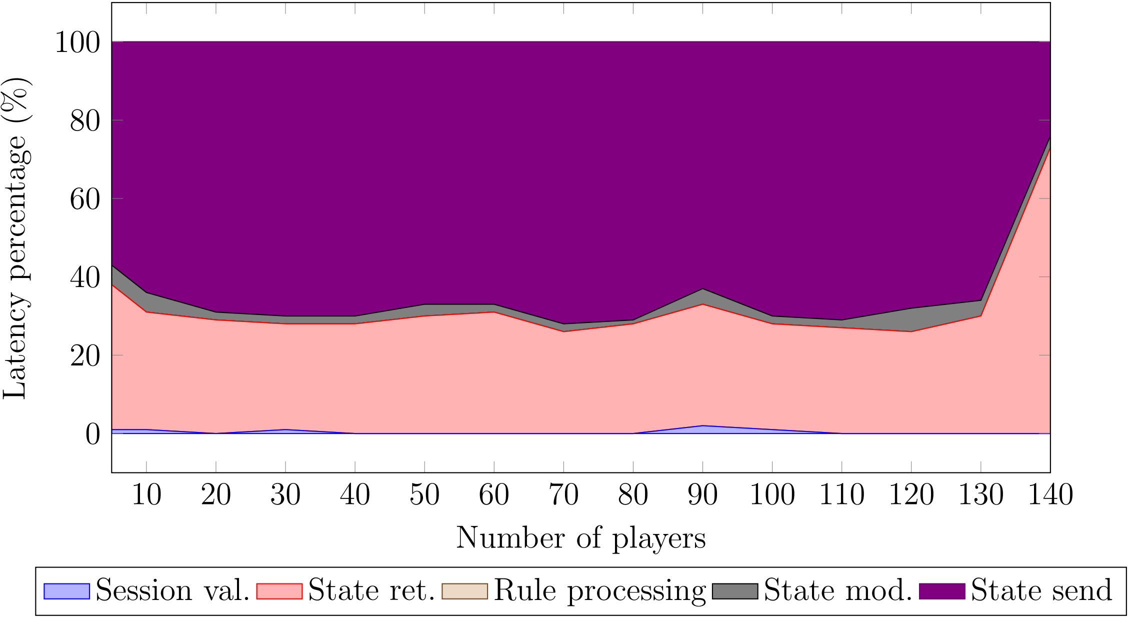


Figure 6.3: Sub-process latency as a percentage of the total service latency in the cloud-hosted version of Mars Pioneer.

hosted simulation, it is determined that the 1000ms latency threshold established in hypothesis 2 is reached at 36.72 players. Based on this figure, the locally-hosted backend can sustain approximately 3.67 players per GB of memory at this threshold latency. Finally, it should be noted that the recorded data includes simulations of up to 60 players, as the local system did not manage to sustain more than 60 players simultaneously. In all of the attempts made to reach higher numbers of active players, the system reached its memory capacity, causing the backend to either take too long to respond or crash before any useful data could be obtained.

The processing latency of these operations is also measured for the cloud-hosted backend, the results for which are shown in table 6.3. From these results, the state retrieval operation is distinguished as the one with the highest average growth, at $\times 2.05$ per 10-player increase. Despite expecting database-heavy processes like this to have a higher impact on performance, this is an unexpected result because the persistence option being utilized in this approach is being hosted on the cloud. When compared to its locally-hosted counterpart, the cloud-hosted state retrieval operation has a higher average growth – albeit at a much higher number of players due to the more efficient use of resources in the cloud-based version. Consequently, further analysis is required for this specific type of operation to understand its impact on performance. Nevertheless, this is the only operation in which the local approach fares better in terms of average latency growth. Across all other operations, the cloud-based backend fares significantly better with an average total latency growth of $\times 1.41$ compared to the locally hosted backend's $\times 2.07$. This growth rate brings the cloud-hosted backend close to an ideal constant growth in latency as the number of active players increases. It also enables it to sustain 131.10 active players below the threshold latency of 1000ms, which is significantly higher than those supported by the locally-hosted backend. In addition, the cloud-based backend employs 16GB of RAM through all its instances and thus supports $\sim$8.19 players per GB of RAM, which is more than double that of the locally-hosted approach – ultimately making it a much more efficient option to deploy an MMOG backend.

The processing latency data of these two deployments are compared in figure 6.4, which shows the processing latency attained by each approach as a function of the number of active players. In the secondary vertical axis, the figure also shows the instances employed by the cloud-based backend, which provides more context for the interpretation of the data. As seen from this figure, the latency of the locally-hosted approach is initially lower than that of the cloud-

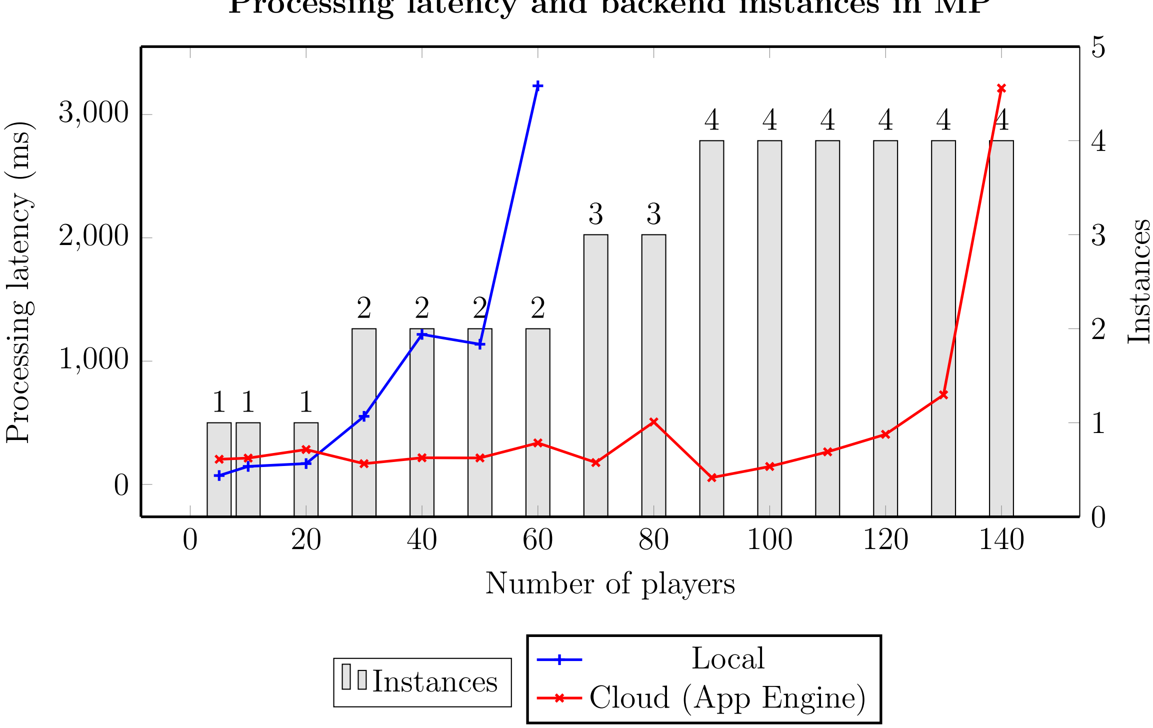


Figure 6.4: Total processing latency of local and cloud-hosted approaches as a function of the number of active players and the number of instances launched.

based approach. The local approach manages to sustain better latency than the cloud-hosted backend for up to 20 players but starts to increase at a higher rate after this point. In contrast, the cloud-hosted backend has a slightly higher processing latency at first but leverages the automatic scaling configuration of the App Engine runtime to spin up a second instance to handle the increased workload during the 30-player simulation. Based on the data, the elastic nature of the cloud-based approach reverses the increasing trend in latency and reduces it to a satisfactory level. These trends continue, with the locally-hosted approach suffering from an ever-increasing latency up to its breaking point of 60 players, whereas the backend hosted on Google's App Engine spins more instances to handle the demand and keep latency below the threshold. Finally, the cloud-hosted backend gives in to the resource limitations set in the deployment configuration at 140 players.

Using the parameters established in the experiment described, a similar attempt is made to compare the processing latency of MMOG backends which are deployed on serverless computing environments, but which are not developed using Athlos. This aims to isolate the two so that the contributions and limitations of the proposed methodology can be identified. To undertake this exploration, the original (non-Athlos) version of aMazeChallenge is deployed

Number of players	Processing latency (ms)	Network latency (ms)	Instances
1	38.73	88.31	1
2	120.81	92.80	1
4	222.83	109.41	1
8	283.22	179.16	1
16	618.23	186.32	3
32	1067.89	185.06	6
64	1884.60	218.36	9

Table 6.4: Processing, network latency, and initiated backend instances in the original version of aMazeChallenge under various number of players.

on Google's App Engine Standard. This specific implementation does not utilize any of the proposed methods and tools but is deployed on a serverless computing platform – allowing the differentiation of the two main factors of scalability: App Engine providing infrastructure scalability, and Athlos providing runtime and state scalability. Using a similar approach as the experiment described previously, the backend of aMazeChallenge is evaluated using a bot simulation. The simulation instantiates varying numbers of bots which all employ the same maze-solving algorithm (left-wall follower) and play within the same maze. The bots join the game in a flat configuration, aiming to provide consistency with the experimental configuration used to evaluate Mars Pioneer. Several trials are executed, with player numbers ranging from 1 to 64. In each of the trials, the processing and network latency is recorded separately, by using markers at specific locations within the source code of the customized client and backend software. In addition, the number of instances initiated by App Engine is also recorded, but this time no limitations are configured in terms of the maximum number of these instances.

The results, shown in table 6.4 and figure 6.5, indicate that aMazeChallenge had to initiate a larger number of instances to deal with the workload compared to Mars Pioneer, despite having to support fewer players. The aMazeChallenge backend initiated a total of 9 instances at peak capacity to support just 64 players, whereas the Mars Pioneer backend managed to support 140 players with just 4 instances below the latency threshold of 1000ms. The main reason for this is the heavier workload per player in this specific backend compared to that of Mars Pioneer – most likely due to the compilation and execution of players' codes which are both very performance-intensive operations. Although the data obtained across these two experiments are not comparable, they offer insights into the impact each game's complexity has on performance.

Figure 6.5: Processing and network latency in aMazeChallenge under varying numbers of players and numbers of instances launched by App Engine.

More interestingly, it is observed that the processing latency in aMazeChallenge increases in direct relation to the number of players under each trial, while the network latency remains constant. This result shows that the serverless environment on which the original version of aMazeChallenge is being hosted in this experiment is capable of scaling to meet the demands of the game, offering an acceptable network latency despite an exponential increase in the number of players. On the other hand, the processing latency of the same backend follows a different trend, increasing in direct relation to the number of players. Based on these results, it appears that in this specific scenario the infrastructure scales, thereby keeping network latency at a low level, whereas the runtime and state of the backend do not. This causes processing latency – a metric directly related to the implementation of the backend – to rapidly increase. The software components used in this approach lack the facilities which may allow game states and runtimes to leverage scalable infrastructures. This is by design, as the original version of aMazeChallenge did not explicitly consider the potential of scaling beyond a classroom-sized number of players. In direct contrast to these results, Mars Pioneer, which is implemented using the proposed methodology and hosted on a similar serverless environment, is capable of scaling its runtime and state more efficiently – thus keeping processing latency in control until the computing resources are exhausted. These results further motivate the development of the

proposed methodology as they provide proof of the importance of software-based methods for managing scalability. They also provide developers with guidance towards the adoption of both scalable infrastructures as well as scalable runtimes and states. The combination of these two can enable MMOG backends to efficiently leverage the resources allocated to them, leading to more competitive and economical MMOG backends.

The usefulness of these results is limited by the fact that they cannot be generalized for MMOG backends at significantly larger scales because these simulations do not reach the numbers of players seen in commercial MMOG backends. Due to the high costs associated with running multiple, continuous experiments that utilize cloud resources, it was not possible to scale the Mars Pioneer backend beyond the aforementioned capacities. However, it is argued that these results are useful because they (a) demonstrate the ability of the Athlos framework to develop a scalable MMOG backend, (b) provide evidence that serverless computing environments can be used to deploy these MMOG backends, (c) prove that cloud-hosted backends have the potential to serve much higher numbers of active players under certain latency thresholds compared to dedicated backends, and (d) reveal a trend in terms of the ratio of latency to the number of active players that may be extrapolated to much higher scales when cloud resources are available. While showing conclusive evidence for the above, further research is required to further evaluate the performance of the Athlos framework at larger scales, isolate its contributions in terms of performance and scalability, and explore the capabilities of serverless computing environments in terms of hosting commercialized MMOG backends.

6.4 State scalability

To investigate the ability of the Athlos framework to develop MMOG backends that can attain, efficiently manage, and distribute scalable states, this section outlines a set of experiments based on the case studies described in section 5. State scalability is defined as the ability of an MMOG to grow or shrink its world states according to the demands of the interacting players. For example, games with scalable states may upscale their states when new players join a world or level, mostly by extending the state of the terrain or instantiating new entities. Conversely, games may downscale their states when players leave, by either removing entities or identifying them as inactive, or shrinking the terrain where needed. State scalability is evaluated based on

four different aspects: *absolute size*, *sub-state loading time*, *query-loading time equilibrium*, and *serialization time*. The experiments designed to evaluate these aspects are related to hypotheses H3 and H4, as well as other challenges discussed in section 1.3.

6.4.1 Absolute state size

The first experiment for state scalability (E3) measures the maximum possible game state of the Mars Pioneer backend when developed using the proposed methodology and utilizing a cloud-based NoSQL datastore. The feasibility study implementation of Minesweeper (MS) is also used for comparison and control. The experiment measures how many chunks of terrain can be generated and stored for a single game world, making it possible to observe how many of these chunks can be created before the persistence option being used becomes overwhelmed. The generation of the terrain is achieved through the use of an algorithm designed to request parts of the terrain at random locations within the game world. After the creation of all the chunks or when the persistence option used reaches its limitations, the number of chunks possible in each approach and game is recorded.

The first stage of this experiment measures the absolute size of each terrain cell for both games, which allows meaningful comparison between the proposed methodology and non-Athlos approaches. For the Mars Pioneer implementation, the absolute size of a single terrain cell stands at 16 bytes, whereas for Minesweeper it is 5 bytes. The state of MP cells is naturally larger than those of MS because the game features more complex gameplay and rules. During the experiment, a terrain generation algorithm is used to obtain parts of the terrain at random locations in both games. Given enough time and the maximum resources available in each approach, this creates as many chunks of terrain as possible, allowing the measurement of the maximum possible state in both approaches. The MP backend is developed using the Athlos framework and utilizes Google's Firestore for persistence, whereas the non-Athlos implementation of Minesweeper uses Google's Datastore. For the Athlos-based backend (MP), the algorithm managed to generate a total of 110,240 cells. The same algorithm was used to generate 52,441 cells in the backend that did not utilize Athlos (MS), despite those cells having a significantly smaller size. To put these numbers into perspective and allow a more meaningful comparison, the Athlos-based backend generated a total of 1,763,840 bytes across the generated cells, which eclipses the non-Athlos backend's 262,205 bytes by $\sim$6.73 times.

These two case studies use two different data stores to cover a broader variety of persistence options and this can make the results of these experiments less comparable relative to using the same datastore in both case studies. However, it is argued that the data is still comparable to a certain extent because these persistence options are very similar in terms of characteristics and resource availability as they evolved from the same initial product. From this experience, it is reported that the terrain generation process in the Athlos-based backend stopped because of quota limitations in write operations for the persistence option used and not because of method limitations. Contrastingly, the non-Athlos implementation reached its capacity much faster as it did not provide any way of efficiently managing the state. The ability of the Athlos-based backend to attain a much higher absolute state size is attributed to the chunk-based method described in section 4.4.7 which allows a significant reduction in database operations by grouping cells together and makes it possible to efficiently distribute the state of the game across different computing nodes. While more research is needed to generalize these results to more complex MMOGs or a broader range of genres, they nevertheless indicate a significant improvement in terms of state scalability over other approaches such as using relational databases or object-based datastores while not employing this method. While it remains unknown how the Athlos-based backend would have behaved had state expansion may have continued unhindered, a trend is established which allows us to predict the ability to support even larger states when more resources were available. Ultimately, this points towards the capability of developing MMOG backends that leverage the chunk-based method employed by the Athlos framework to sustain very large, expandable states.

This experiment is further extended to explore how the proposed methodology's terrain identifier type is utilized within the framework and to evaluate its overheads. As mentioned in section 4.3.6, the terrain identifier type is used to allow the efficient indexing and management of chunks by detaching these objects from the states of cells included within them – thereby reducing the performance costs of retrieving information about chunks. Despite their usefulness, terrain identifiers add an overhead to the backend as they require an extra write operation. The absolute size overhead of terrain identifiers can be calculated using their attributes. This calculation assumes the standard 16×16 chunk size (256 cells) and UTF-8 string formatting, with each terrain identifier containing 3 ID strings composed of 36 basic characters (108 bytes) and a matrix position (8 bytes) totaling 116 bytes. The absolute size overhead is calculated as $116 \div 256$, which equals to $\sim$0.453 bytes per cell – a negligible amount of data considering

the performance improvement they can offer. In the Athlos-based Mars Pioneer backend, the generated cells were grouped in 6,890 chunks, meaning that as many identifiers are needed to manage the game state more efficiently. This doubles the number of documents being written to the datastore to 13,780. Massive states like those tested in this experiment can greatly benefit from the use of terrain identifiers as many of the used datastores and caches feature limited or no support for queries and filtering. Despite the performance improvement they offer, terrain identifiers add a significant write operation overhead. Furthermore, the worst-case scenario of using 4×4 chunk sizes dramatically increases the size overhead to 29 bytes per cell. Considering these overheads, the proposed methodology allows developers to utilize terrain identifiers as an option in games that are expected to reach massive scales or those typically modeled using larger chunk sizes. Alternatively, they can be avoided in other games which are not expected to grow significantly in state size or which are typically modeled using smaller chunks.

6.4.2 Sub-state loading time

Building on the experience of the previous experiment, the second scalability experiment (E4) aims to explore how the time taken to retrieve a sub-state of the world changes with respect to its full size. In this experiment, the Mars Pioneer implementation is used to create a game world that is initially empty. Using specialized code the world is then populated with a number of chunks that comprise the world's full state. The number of chunks contained in each world is changed in each experimental configuration, which allows the time taken to load a single 16×16 chunk to be measured. To make results more consistent the world state always contains chunks within a certain range of positions for each configuration and chunks are generated prior to their retrieval to negate the impacts of terrain generation. In addition, these chunks are not previously loaded into the backend to make sure that this always entails the execution of the same operations. To increase the effective range of this experiment, the full state size starts at a single chunk for the first configuration and is increased by a factor of two in each run, up to 1024 chunks. Trials for each configuration are repeated three times, with averages taken for each number of chunks in the full state.

The results, tabulated in table 6.5 and graphically presented in figure 6.6, show that the average time taken to load a single 16×16 chunk remains constant as the size of the full state increases exponentially. This constant order of growth is attributed to the chunk-based method employed

Full state (Number of chunks)	Average loading time (ms)
1	34.67
2	34.00
4	37.33
8	31.66
16	40.33
32	33.66
64	30.00
128	34.33
256	31.33
512	30.33
1024	33.33

Table 6.5: Time taken to load a single, pre-generated, not previously loaded 16x16 chunk as the full size of the state increases.

by the Athlos framework, which uses map data structures and hash values to retrieve parts of the terrain in constant time regardless of the total number of chunks in the state. Consequently, these results confirm the known theoretical underpinnings for the use of this type of data structure. It is therefore argued that MMOG backends developed using Athlos can inherently support massive, expandable game states that can be accessed and managed with a minimal performance cost.

6.4.3 Queries vs loading time

The success of the chunk-based method in experiments E3 and E4 in terms of enabling large-scale, persistent, efficiently-manageable game worlds inspires further exploration that could potentially lead to more optimizations. Experiments E1 and E2, which studied the performance of various operations in MMOG backend services have established that the most performance-intensive operations involve heavy or frequent interactions with the persistence layer. These results motivate a study of the effects of frequent interactions with the persistence layer on performance, as well as the monetary costs of frequently running such operations. As described in section 4.4.7, the chunk-based method is flexible in allowing developers to customize the maximum size of chunks in each game, within the range 4-64 (i.e. ranging from square chunks of 4×4 to 64×64). This range was selected intuitively, and it is expected that the number of queries required to fetch a sub-state of the world will greatly vary based on the chunk size.

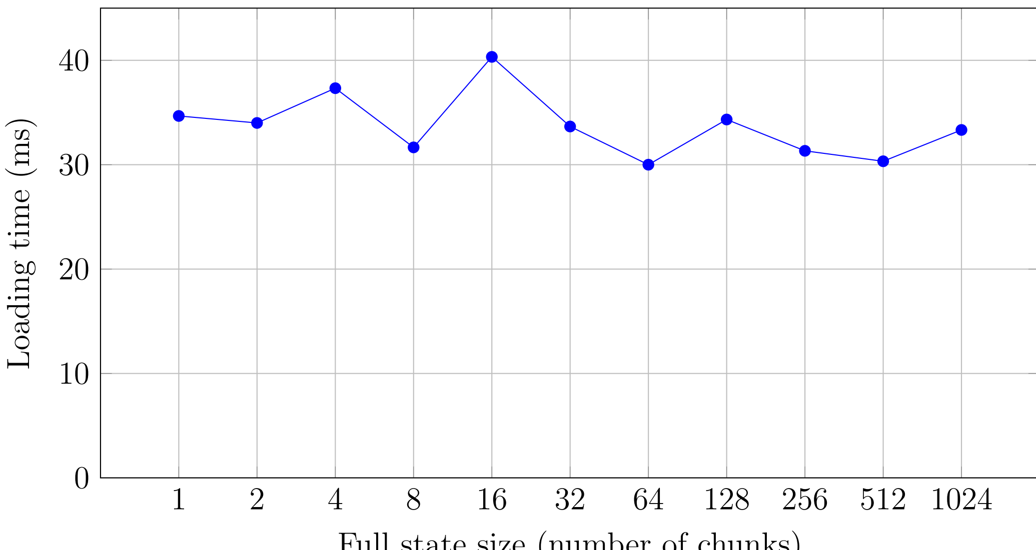


Figure 6.6: The amount of time taken to load a single 16x16 chunk as the size of the full state increases.

Chunk size (cells)	Number of queries	Retrieval time (ms)	Generation time (ms)
4x4	250	4.80	19.18
8x8	125	3.03	24.26
16x16	63	2.20	34.94
32x32	32	2.37	74.13
64x64	16	4.85	303.38

Table 6.6: Results showing how different chunk sizes affect the number of queries, average retrieval time, and average time per generation.

To shed light on which chunk size is the most efficient, experiment E5 explores the effects of chunk size on the number of queries needed to fetch a sub-state of the world with a fixed size, and the loading time of chunks. The goal of this experiment is to determine which chunk sizes achieve (a) the best performance in terms of state loading time, and (b) the lowest number of database queries. The experiment uses the Mars Pioneer case study with previous experimental setups to create a world in which terrain will be generated and retrieved. A specialized program is then used to request 1000 individual cells within the range (0,0) to (0,999). This experiment is conducted across 5 different configurations: 4×4, 8×8, 16×16, 32×32, and 64×64, recording the number of queries required to fetch the state, the average time taken to retrieve cells, as well as the average time taken to generate them. Various other factors such as the size of the chunks, the implementation of the tools used to run this experiment, the datastore location, and policies, as well as backend, network, and device utilization, are kept in control.

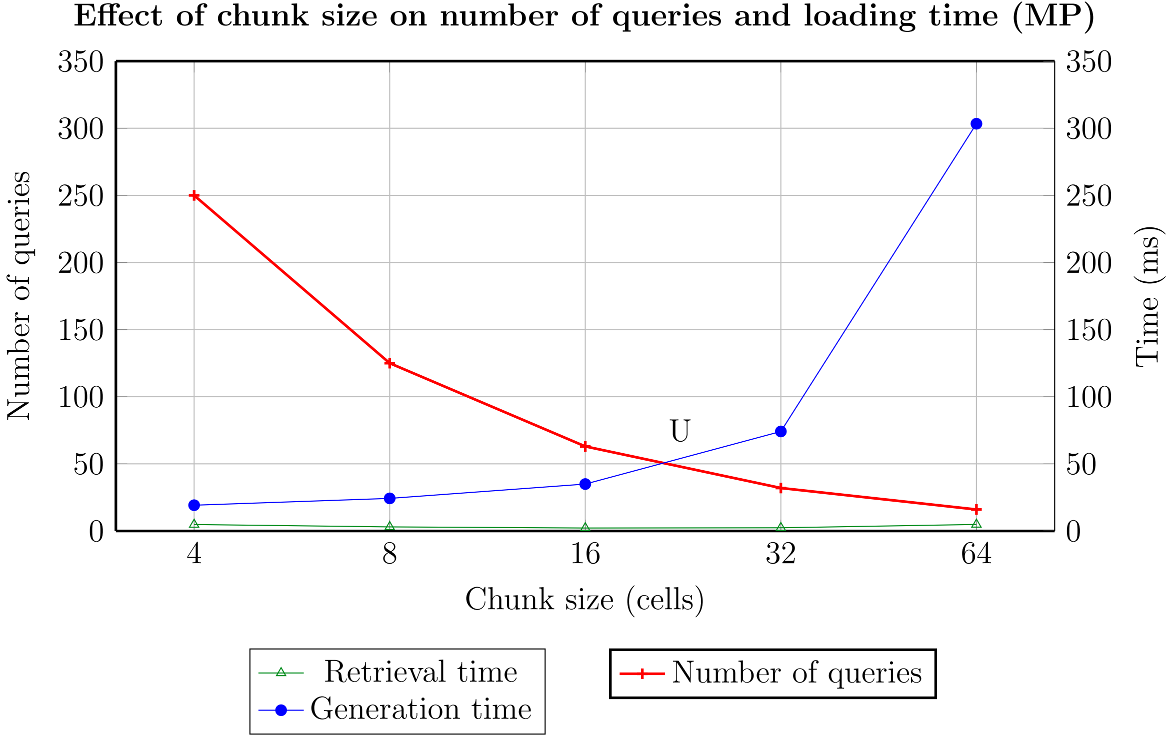


Figure 6.7: The effect of chunk size on the number of queries required to fetch a part of the game state and the time taken to generate the chunks.

The results, shown in table 6.6 and graphically presented in figure 6.7 show that as the size of the chunks grows, the number of queries required to retrieve the partial state is reduced. It is also observed that retrieval time remains stable and at negligible levels throughout all chunk sizes. From the data collected, it is not possible to conclusively determine a trend in the relationship between retrieval latency and chunk size. Assuming that no such trend is visible, it is deduced that the chunk size does not affect the retrieval time of the partial state. On the other hand, generation time is strongly affected by chunk size. The data shows that generation time increases in direct relation to the size of the chunks. A noteworthy fact that aids the interpretation of the data is that the number of cells included in the chunks tested in this experiment increases in powers of 4 – making this an exponential growth in terms of the size of the data being encapsulated in the chunks. It is also observed that there is an intersection point between the number of queries required to retrieve the partial state and the time taken to generate the state. Since the retrieval time remains unaffected, the evaluation of the scalability and performance of the backend can be mainly based on these two factors. The intersection point of these two factors may be regarded as the *equilibrium between the number of queries and generation time*, which is marked by U in figure 6.7. This point, valued at $\sim 22.4^2$ cells per chunk might provide the optimal balance between the number of queries required to retrieve

the state and the time taken to generate it. Despite neither of the two values being at their lowest at this equilibrium, both of them are rather close to their optimal values. For example, at this point, the number of queries required to retrieve 1,000 cells is about 50, which means each query has to retrieve 20 cells. The size of such a query is neither too large nor too small, also allowing for a balance in terms of serialization time. Similarly, the average generation latency to retrieve 1,000 cells is about 50 milliseconds at this equilibrium, which means that the retrieval of a single cell induces a latency of only 0.05ms.

Chunk sizes smaller than the identified equilibrium may reduce the time taken to generate the cells at the cost of increasing the number of queries required to retrieve them. On the other hand, larger chunk sizes may increase the generation time but require fewer queries to retrieve the state. While these may sound inefficient at first glance, it is entirely possible that different types of MMOGs may benefit from larger or smaller chunk sizes based on their gameplay. As an example, some games may differ in terms of their terrain retrieval requirements, only loading the state once or even not at all, while others may have to constantly update it. For games that are not expected to retrieve terrain at a fast pace, it is believed that a larger chunk size (e.g. 32 or 64) will improve cost efficiency by reducing the number of queries and entities being written in the database. Alternatively, games that are expected to retrieve terrain at a relatively high frequency may opt to use smaller chunk sizes (e.g. 4 or 8) to achieve lower latency. Finally, games that are known to require a balance between the two may utilize the equilibrium point identified above (22) or simply select to use the closest alternative of 16×16 chunks.

6.4.4 Serialization time

Another important aspect that determines the scalability and performance of an MMOG backend is bandwidth consumption, as identified in the analysis of related works in chapter 2. Bandwidth requirements in an MMOG backend can be measured in terms of bytes and are mostly affected by the state of the game rather than other, occasionally communicated information. Experiment E6 attempts to measure the bandwidth requirements of an MMOG backend developed using the Athlos framework. To achieve this, the aMazeChallenge case study is used to create different sizes of mazes and record how many bytes are required to serialize and transmit their state. The aMazeChallenge case study is used in this particular experiment because of two reasons. Firstly, this game features a non-expandable game state, which is more

Maze size (cells²)	Non-Athlos state size (JSON) in bytes	Athlos state size (Protocol Buffers)
5	713	53
10	792	128
15	917	253
20	1092	428
25	1317	653
30	1592	928

Table 6.7: State size requirements for the non-Athlos and Athlos implementations of aMazeChallenge for various maze sizes.

manageable and less complex to experiment with compared to the expandable states found in Mars Pioneer. Secondly, aMazeChallenge's original version, which is implemented to use the JSON format for serialization can be used for comparison and control. In the original version, JSON is used through Google's GSON library for Java, whereas the Athlos-based version uses Protocol Buffers. In this experiment, the comparison is made exclusively between these two serialization/deserialization methods.

The comparison between these two approaches is made in the context of aMazeChallenge, a case-study MMOG backend. The communication of the state, found in aMazeChallenge as a `Grid` object, is the most frequently occurring operation, and therefore measuring the size of objects of this data type can be a direct indicator of the backend's bandwidth consumption and serialization overheads. To measure this consumption, different sizes of mazes are created ranging from 5×5 to 30×30 cells — the largest state possible in the game. Several parameters are kept in control, such as the types of mazes and their contents and the entities being generated in each game. The state size of the non-Athlos (original) implementation of the game is measured by recording the size of the JSON-formatted string produced by a Grid object in bytes. To produce JSON-formatted strings, this version of the game uses the GSON library. The size of the state in the Athlos-based implementation is recorded by measuring the size of the Grid as a serialized Protocol Buffer object.

As the results in table 6.7 and figure 6.8 show, the Athlos implementation of aMazeChallenge achieves a significantly lower state size across all maze sizes compared to the non-Athlos implementation. The data also shows that the Athlos-based approach using PB can serialize the game state of aMazeChallenge in as little as 7% of the total size needed by the JSON format in the 5×5 maze. The Athlos-based approach appears to be more efficient, with the Protocol

Figure 6.8: A comparison between the serialization formats used in the non-Athlos (JSON) and Athlos (Protocol Buffers) implementations of aMazeChallenge, across a range of state sizes.

Buffers averaging 33% lower size requirements across all state sizes. Despite that, a decreasing trend is observed in this efficiency as the state size increases. For instance, in the 30×30 maze, the Athlos approach used 58% of the size of the state in the non-Athlos approach, which is far more than the 7% observed in the 5×5 maze. Data shows that there is no case in which the Protocol Buffers mechanism, which is employed by the proposed methodology, is less efficient than JSON.

A similar experiment is conducted using the Mars Pioneer case study to explore whether this trend applies to another type of game state. In this scenario, the experiment involves the creation of multiple Player objects which are then separately serialized using the JSON format and Protocol Buffers. To keep this experiment fair both approaches serialize the same data objects containing the same attributes and values, which are created and initialized prior to their serialization. The experiment uses various numbers of objects, starting from 10 and moving up by a factor of ten, up to 1 million objects.

The results of this experiment, shown in table 6.8 and figure 6.9 confirm the earlier results obtained from the aMazeChallenge case study. In both cases, the Protocol Buffers mechanism manages to be more efficient than JSON. In the case of Mars Pioneer, the PB mechanism is approximately 33% more efficient than JSON, which is identical to the average value found in the previous experiment. However, in MP this improvement in efficiency does not vary as much

Number of objects	JSON size (bytes)	Protocol Buffers size (bytes)
10	4,513	1,510
100	45,013	15,100
1,000	450,013	151,000
10,000	4,500,013	1,510,000
100,000	45,000,013	15,100,000
1,000,000	450,000,013	151,000,000

Table 6.8: The size of the state in bytes, as serialized by both JSON and Protocol Buffers in Mars Pioneer.

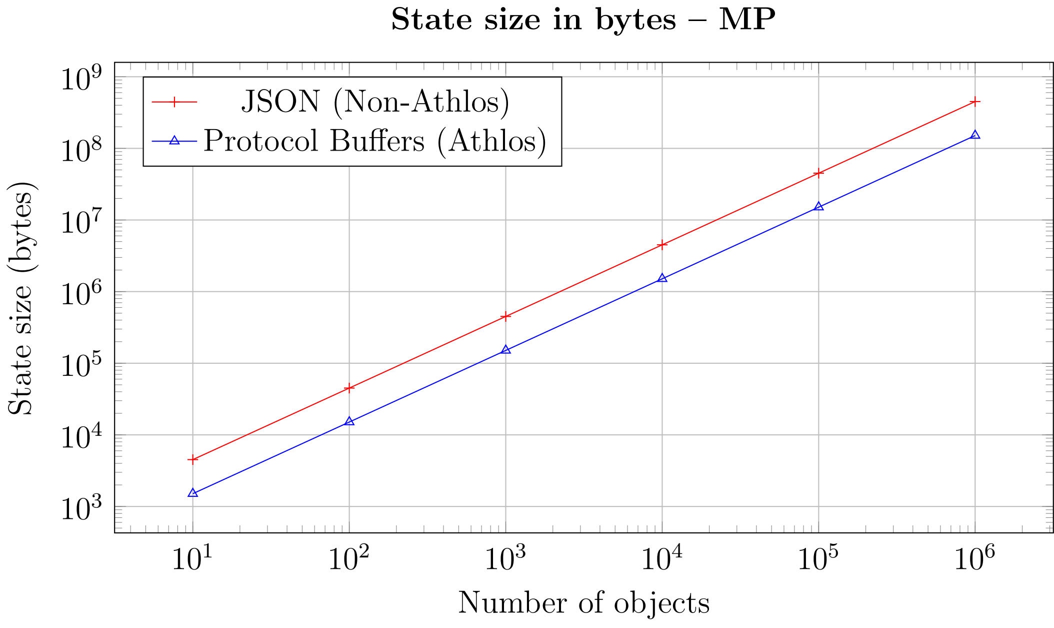


Figure 6.9: A comparison between the size of the state when serialized using JSON and Protocol Buffers in Mars Pioneer, across a range of object numbers.

Number of objects	**JSON serialization time (ms)**	**JSON de-serialization time (ms)**	**PB serialization time (ms)**	**PB de-serialization time (ms)**
10	1.00	0.33	0.00	0.33
100	1.67	1.00	0.00	0.33
1,000	6.00	4.00	0.67	1.00
10,000	30.67	18.33	1.33	2.33
100,000	289.33	144.33	3.33	19.67
1,000,000	2147.67	1241.33	33.33	192.67

Table 6.9: Results obtained from the serialization and de-serialization of objects in Mars Pioneer using JSON and Protocol Buffers, for various numbers of objects.

as it does in aMazeChallenge, only fluctuating in the thousandths of these percentages – perhaps signaling a less deviant data set. These results confirm that MMOGs that are developed using Athlos and which enjoy its facilities and abstractions in terms of utilizing Protocol Buffers can leverage this approach to benefit from reduced state sizes and bandwidth, and ultimately achieve better economy over time.

Bandwidth consumption is only one side of the coin in terms of evaluating serialization mechanisms and formats. Another important characteristic is their performance, which can be measured by the time taken to serialize or de-serialize data objects. To explore the performance of the tested serialization options, experiment E7 attempts to measure the time taken by JSON and Protocol Buffers to serialize/deserialize many identical data objects. This setup utilizes a specific MMOG backend implementation that was developed using Athlos. Trials start from 10 objects, incrementing the number of objects by a factor of ten up to 1 million. Each of these trials is repeated three times for every number of objects and averages are recorded. The objects being converted are kept identical across runs and during each run, with both JSON and PB using identical data. The same experimental design is followed for both the serialization and de-serialization of objects and the time taken to run through these operations is recorded.

The data obtained from this experiment are tabulated in table 6.9 and graphically illustrated in figures 6.10 and 6.11. A noticeable pattern in these results is that Protocol Buffers are an order of magnitude more efficient than JSON in terms of the time taken to serialize and deserialize data. In the context of an MMOG backend developed using Athlos, it is expected that moderate amounts of data may be communicated by the services in fast, continuous bursts. For example, even games that employ the maximum chunk size possible (64×64) will need to transfer 4,096 cell objects per chunk, which is a moderate amount. For up to 10,000 objects

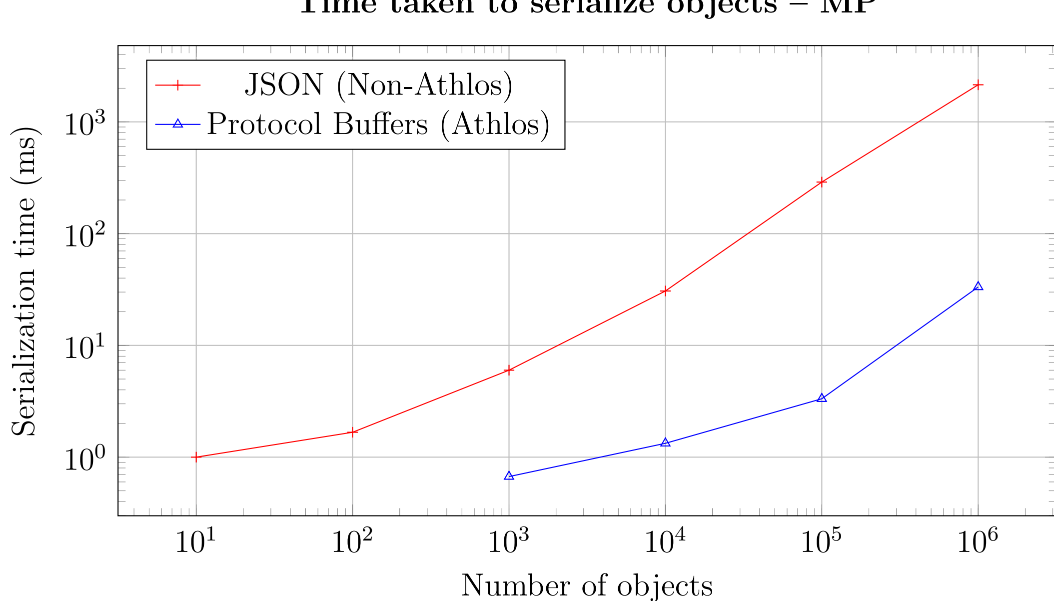


Figure 6.10: A comparison between the time taken to serialize identical objects when using JSON and Protocol Buffers in Mars Pioneer, across a range of object numbers.

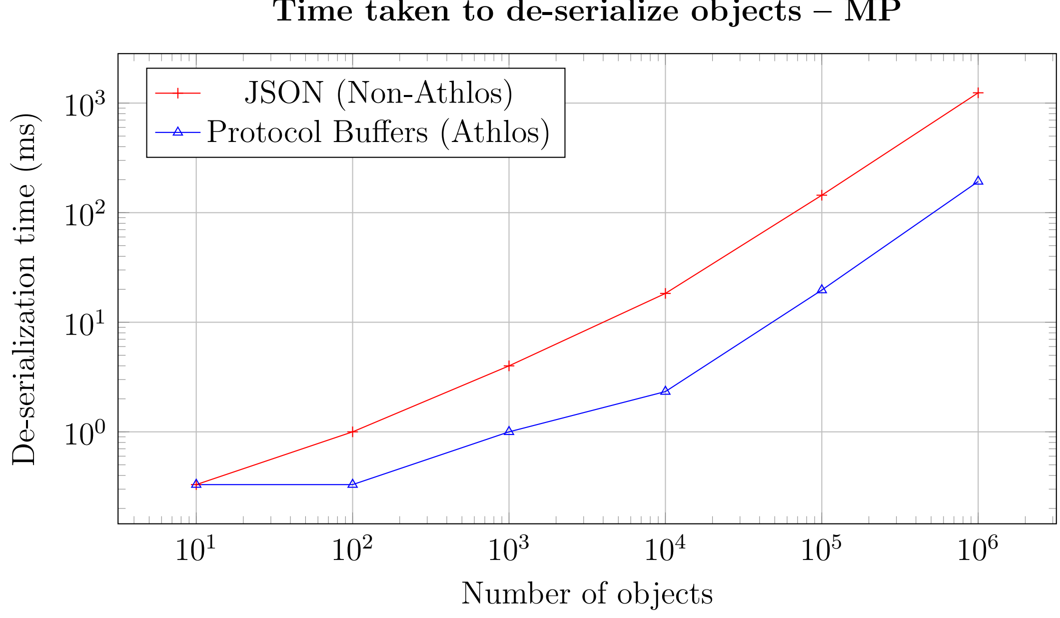


Figure 6.11: A comparison between the time taken to de-serialize identical objects when using JSON and Protocol Buffers in Mars Pioneer, across a range of object numbers.

the PB approach offers negligible performance overheads – 1.33ms for serialization and 2.33ms for de-serialization. By contrast, the JSON approach takes much longer to serialize (30.67ms) and de-serialize (18.33ms) as many objects. This is 23 times longer than PB for serialization and about 8 times longer for de-serialization. Given that games may need to continuously carry out this task at a rapid pace, and taking into account the performance overheads of other operations within the services of MMOG backends, the use of PB appears to be a significantly more advantageous option for serialization.

6.5 Development effort

The second aspect of the evaluation explores how much effort is needed to develop MMOG backends by using the proposed methodology. From a software engineering perspective, Athlos aims to reduce the development effort through the use of various abstractions included within its default model, a modular architecture that divides code into independent components, as well as various methods and tools that aim to expedite development through the use of software design patterns or other software engineering principles. This aspect of the evaluation is guided by and attempts to answer hypothesis 5 (H5), introduced in section 1.3.

Different metrics can be used to measure the software development effort. In most projects, regardless of how large or small they are, a decomposition technique can be used to estimate the effort required to develop a certain product. Given a certain project scope and a good estimation of the size of a software product, it is possible to generate an estimation for the amount of effort required to develop it. This evaluation is not concerned with studying the scope of a project or the size of each case study. Instead, different metrics can be used to estimate the effort required to develop each case study. One approach to estimating effort divides the activities undertaken into tasks, for which certain timeboxes can be assigned, and thus an estimation of the effort undertaken can be measured. A second estimation method involves the calculation of person-hours or person-days required to complete a certain task. While these types of estimation work well in larger, collaborative projects, the case studies developed and showcased in this thesis were developed by a single person, thus making these estimations relative to one's experience in utilizing a certain approach, programming language, or in general, their technical expertise. Another method of evaluating the effort required to develop an MMOG backend is by measuring

the Source Lines of Code (SLOC) written. SLOC is a relatively simple software metric used to measure the size and complexity of a computer program, as well as to estimate development effort, productivity, and to some extent its maintainability.

The impact of the proposed methodology in terms of software effort can be evaluated – albeit to a limited extent – using SLOC, and by using the two implementations of the Minesweeper case study in experiment E8. The initial version of MS, which was presented in chapter 3, is developed without the Athlos framework and makes no effort to provide any additional facilities in terms of improving scalability and performance. On the other hand, the implementation of the same game using Athlos presented as a case study in section 5, includes many additional features. Despite that, these two versions feature identical code in terms of their service logic and are thus considered very similar. The SLOC measured for these two projects are separated into two categories: those which are *efforted* by the developer (i.e. directly written code), and those which are automatically *generated* by software tools. This separation helps understand which percentage of the code is written by the developer, and for which effort was required. In these measurements only source code files are included – omitting any project, configuration files, resources, or Athlos project definitions. Source code that is not related to the game's implementation and core functionality, such as simulation harnesses, data, and configurations are also excluded. In addition, source code produced by the Protocol Buffers compiler is also excluded even though these files are actively being used in the game. It is argued that the omission of these files makes the comparison fairer as PB is an external library that could be used in either approach. The measurements for SLOC in both approaches are measured using a plugin in IntelliJ IDEA called Statistic.

The results, shown in figure 6.12, show that the Athlos-based project contains 3,628 SLOC – about 1.5 times more than that of the non-Athlos implementation of Minesweeper (2,355 SLOC). This difference in lines of code can be attributed to the fact that implementations based on the Athlos framework contain a more diverse set of functionalities than that required by a specific game and especially a simple game like Minesweeper. The SLOC measured for each of the two implementations can also be considered in terms of the two categories identified above. By separating the SLOC in each of these two categories, it is possible to observe that the number of lines generated in the Athlos-based implementation greatly exceeds that of those which were manually efforted. Comparing just the efforted SLOC between the two implementations reveals

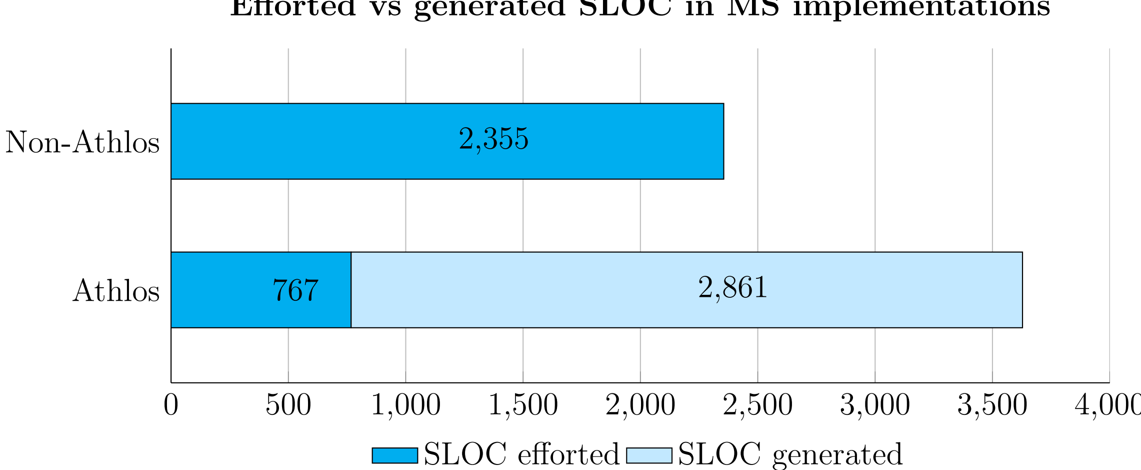


Figure 6.12: A comparison between the Athlos and non-Athlos implementations of Minesweeper in terms of source lines of code efforted.

that the Athlos-based project required only a third (32.6%) of the lines of code efforted in the non-Athlos implementation. Even though the Athlos-based implementation results in a significantly larger project, most of the source code in that project is automatically generated by the software tools described in section 4.5. These results are isolated to a specific, relatively simple MMOG backend implementation, and therefore further research is required to fully comprehend the impact of the proposed methodology on the development effort required to realize MMOG backends.

Although SLOC is a popular software metric used to evaluate development effort, the results it provides are of limited reliability. Firstly, there are varying definitions of what constitutes a SLOC, and this is problematic for two reasons: (a) the definition of this metric is circumstantial, or at least subject to interpretation based on the approach used to calculate it (Rosenberg 1997), and (b) it makes it significantly harder to compare SLOC across different programming languages, making the results of this experiment valid only to the approach being used. Secondly, the SLOC metric only looks at a single part of the development lifecycle – the implementation – thus ignoring other development processes which are also time-consuming and thus important to the effort undertaken. Processes like requirements and risk analysis, design, testing, and deployment, also have an impact on the development effort of a system, especially with regard to designing efficient and scalable MMOG backends that are deployed on commodity clouds. Finally, it is conceded that other, more effective metrics can be used to evaluate development effort and cost estimation with a higher degree of precision, which are further discussed in section 8.3.

Despite the disadvantages of using the SLOC metric in providing a development effort estimation, it is a relatively simple technique that can be employed quickly in simple projects like the Mars Pioneer prototype. The fact that the implementations compared in the experiment described above use the same programming language, environments, tools, APIs, and follow the same conventions and programming style makes it possible to use this metric with higher reliability than otherwise. From these limited results, it can be argued that the Athlos framework has a positive impact on development effort by significantly reducing the number of lines of code required to implement an MMOG backend. However, these results cannot be generalized for larger and more complex projects, where more reliable cost-estimation techniques would be more useful in providing a better picture of the framework's usefulness concerning the reduction of development effort.

6.6 Code maintainability

Apart from the effort undertaken to develop MMOG backends, the evaluation studies the quality of the code produced by the software tools mentioned in section 4.5 and within the Athlos API. This aims to explore hypothesis 6 (H6) by measuring the quality of the code within the case studies. The quality of source code relates to many different aspects, including its *reliability*, *maintainability*, *testability*, *portability*, and *reusability*. Furthermore, other measures can be used to identify code quality, such as the Mean Time Between Failures (MTBF), the number of defects detected, the defect density, and more. The evaluation of code quality is a relatively complex topic that warrants attention to detail and deep knowledge of various aspects of code design. While there is a large set of methods and tools which can be used to evaluate code quality, this thesis attempts to simplify this procedure by using a suite of criteria known as Chidamber-Kemerer (CK) metrics (Kumar & Kaur 2011). This suite can be used to test software against a variety of aspects that are mainly related to object-oriented design.

The experimental procedure (E9) to evaluate code quality involves the use of the aMazeChallenge and Minesweeper studies, which are implemented both as non-Athlos and Athlos-based projects. This makes it possible to compare the two implementations for each case study and determine how the proposed methodology impacts the code's quality. The measurements for various CK metrics are taken using a plugin for JetBrains-based IDEs called MetricsReloaded.

This plugin automatically measures CK metrics in a project and then produces statistics and reports which can be either analyzed directly or exported. In a similar manner to the evaluation of development effort, several items are excluded from these measurements. For instance, source code related to simulations or generated by the Protocol Buffer compiler is excluded from this evaluation. The plugin is then used to measure the values of the following CK metrics, the values of which are inversely proportional to the quality of the code in a project – i.e. a lower value is considered more 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.

The results obtained from this process are shown in tables 6.10 for Minesweeper and 6.11 for aMazeChallenge. For Minesweeper's implementations, it is observed that the non-Athlos approach offers better code quality by a relatively wide margin compared to the Athlos-based implementation as it manages to obtain lower (better) scores for 4 out of 6 CK metrics. Based on the measurements for LCOM, RFC, and WMC, the non-Athlos implementation has a higher degree of cohesiveness in its class methods, makes fewer procedure calls per message received in each of the classes, and has reduced complexity compared to the framework implementation. On the other hand, the framework-based implementation offers less inheritance depth and associations between classes (DIT, NOC). For the aMazeChallenge case study, the comparison

between the two implementations is inconclusive. Both of these approaches seem to have their own advantages and limitations in their respective areas. For instance, the Athlos-based implementation scores better in terms of DIT, LCOM, and NOC, meaning that its class hierarchy is slightly less complex and that it offers better cohesiveness in terms of its class methods. On the contrary, the code in this implementation scores worse than its non-Athlos counterpart in terms of cyclomatic complexity (WMC), spawns more procedure calls for each message received per class (RFC), and appears to be more intertwined (CBO).

While being relatively limited in scope and depth, this evaluation offers a glimpse into the quality of the code produced by the proposed methodology. From the results obtained, it is not possible to determine a clear pattern. It appears that in some metrics and game implementations the Athlos framework fares better, whereas in others it makes the code quality worse. Meanwhile, it is possible that code quality may be influenced by other factors which could not be controlled in this experiment, such as personal preferences in code style and design. For instance, during the development of the case studies, no provisions or considerations were made regarding code quality. The code produced by the framework, as well as its APIs is also designed to provide developers with various convenience methods for accessing different features and components of the architecture, which are aimed at expediting development rather than achieving better quality. The suite of metrics used in this evaluation, as well as the plugin used to measure them, are also specific, and may therefore limit the usefulness and generalizability of the results. Consequently, a more thorough study and evaluation of the quality of the code produced by the Athlos framework is warranted, which may help achieve clearer and more generalizable results. Despite these limitations, it is possible to use these results to obtain better knowledge in terms of producing better code quality, as well as guidance for future improvements in the framework. It is therefore acknowledged from these results that the framework could be improved to produce higher quality code, making the code in MMOG backends produced with this approach more readable, maintainable, testable, reliable, and portable.

6.7 Tools

The final part of the evaluation attempts to assess the usefulness and performance of various software development tools which are part of the proposed methodology. These tools are created

	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 6.10: CK metric measurements for the two implementations of Minesweeper.

	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**

Table 6.11: CK metric measurements for the two implementations of aMazeChallenge.

specifically to support scalable MMOG backends, but are independent of the Athlos framework and may be used in other types of applications. Thus, the experiments to evaluate them are designed to be independent of the framework or the produced case studies, even though they may contain related models. The usefulness of these tools is evaluated based on their impact on performance and development effort, which are identified as significant properties for MMOG backends. As these tools are of secondary importance, their evaluation is described in Appendix 9.G.

6.8 Conclusions

This chapter presented an evaluation of the proposed development methodology through several experiments. The case studies described in section 5 were used to evaluate the performance, scalability, code maintainability, and development effort required to develop MMOG backends using the Athlos framework. While not representative of all types of games, technologies, and circumstances, they provide insights into the behavior of MMOG backends and establish patterns which may be extrapolated to much larger scales. The results of these experiments can provide answers to the research questions and hypotheses posed in section 1.3, and be used as

a starting point for more elaborate future work. These are further discussed in the subsequent sections.

Chapter 7

Analysis

> "*The good thing about science is that it's true whether or not you believe in it.*"
>
> Neil deGrasse Tyson

7.1 Introduction

Chapter 6 presented the strategy to evaluate the proposed approach and various experiments to determine its usefulness with regard to various aspects of performance, scalability, development effort, and code quality. Various tools which are utilized within the framework are also evaluated. While analyses are offered for each of the results presented in this chapter, these are specific to each experiment and do not offer a look at the bigger picture. This chapter attempts to interpret the results from a different perspective, attempting to offer an analysis at the framework level, based on the hypotheses and technical challenges introduced in sections 1.3 and 3.7. The results of the evaluation are combined with the findings from other chapters to answer many of these questions. Ultimately, this chapter aims to provide a high-level view of various patterns and behaviors established from this research and to explain how the proposed models, methods, and tools can be leveraged to develop scalable MMOG backends and enable their deployment on commodity clouds.

7.2 Addressing the hypotheses

7.2.1 Hypothesis 1

MMOGs that are hosted in serverless cloud environments and utilize the proposed 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 the framework.

This hypothesis is mainly addressed in section 6.3, experiment E2. This experiment first attempts to provide an understanding of how different operations impact the performance of an MMOG backend, and then explores how performance is affected by various deployment approaches. In the second part of this experiment, a prototype MMOG backend was deployed on a relatively powerful dedicated machine as well as on a serverless environment (Google's App Engine Flexible), with the aim of exploring how the latency of the backend is impacted with respect to the number of active players. To make sure that the data recorded is not affected by external factors that are irrelevant to the backend, a new metric called processing latency is used, which helps negate the impact of network latency and client device performance. The results of this experiment show that cloud-based serverless computing environments enable MMOG backends to scale to accommodate more active players, while the Athlos framework allows the runtime and state of the game to effectively leverage the resources of these environments. Consequently, it is shown that MMOG backends hosted on serverless environments and using the proposed methodology can achieve a much better (lower) ratio of latency to the number of active players compared to other approaches that use dedicated architectures. The average ratio of latency to the number of players for the non-Athlos dedicated backend is about 23.27, whereas the Athlos-based, serverless backend scores a significantly lower average ratio of about 9.50 – about 41% better.

A second, similar experiment involves a different MMOG backend which is hosted on a similar serverless infrastructure (App Engine Standard) but does not utilize Athlos. This aims to explore whether the methods and tools proposed have a meaningful impact on how MMOG backends scale in these types of infrastructures. As the data illustrates, this specific deployment managed to maintain its network latency to low levels despite exponential increases in

the number of players, showing that the underlying infrastructure could successfully scale to accommodate these increases. The measurements for processing latency indicate that the backend's performance decreases as more players join the game. The disparity between these two measurements shows that this particular backend does not leverage the scalable infrastructure it is deployed on, and as a result, it is constrained by limitations that are not related to the availability of resources. Based on these insights, it is argued that utilizing only one of the two components – either a serverless environment or the proposed framework – does not suffice to provide the necessary scalability for MMOG backends and that a combination of the two is needed to achieve it. While more diverse experiments that leverage more computing resources are needed to generalize these results, the data obtained from these experiments seem to confirm hypothesis 1.

7.2.2 Hypothesis 2

MMOGs based on the proposed framework and hosted on serverless clouds can sustain a higher total number of active players than single-machine, non-framework approaches, under the threshold latency of 1000ms.

Data from the same experiment (E2) can be used to explore the second hypothesis. The results show that the non-Athlos, dedicated deployment of the MMOG backend prototype only managed to support up to 36.72 players before reaching the threshold latency. The Athlos-based, serverless deployment of the same backend managed to sustain up to 131.1 players below the same threshold – about 3.57 times more. The threshold latency of 1,000ms is arbitrary and is assigned as such for experimentation purposes. It is not specific to any MMOG or game genre, but it attempts to capture the latency requirements of many types of MMOGs, such as MMORPGs and MMORTSs. Based on the related works (Nae, Iosup & Prodan 2010, Shea et al. 2013), higher latency negatively impacts gameplay experience in such genres, which is the major factor influencing the value chosen for this threshold. Based on the results of this experiment, it is evident that Athlos-based MMOG backends deployed on serverless clouds can sustain higher numbers of active players than non-Athlos dedicated approaches, even though more research is required to generalize this outcome.

7.2.3 Hypothesis 3

MMOGs that utilize the proposed framework are able to feature very large and expandable game states (within the limits of the hardware resources being utilized).

This hypothesis aims to explore if and how the proposed methods enable the creation and management of very large, expandable, and theoretically infinite game world states. Experiment E3 measures the maximum game state possible in an MMOG backend prototype that utilizes a scalable NoSQL datastore. A specialized algorithm is used to simulate requests for state retrieval at various points in a game world, aiming to observe how many terrain cells can be created before the resource or approach limitations are reached. An Athlos-based backend was able to generate and store a total of 110,240 cell objects before reaching daily quota limitations, while another backend which did not utilize Athlos only managed to generate 52,441 cells before reaching object size limitations in its utilized datastore. The usefulness of these results is limited by the fact that only a bounded set of resources were available for this experiment. Further research is required to determine how results may have differed if more resources were available during the experiment and if the daily quota was not in place. The scalability limitations of the non-Athlos backend stem from its inability to provide methods with which game states can be distributed among multiple data objects – a problem that is described in detail within section 4.4.7. It was thus impossible for this backend to scale its world state beyond the 52,441 cells it had reached due to datastore object size limitations, even though the resource limitations had not been reached. On the other hand, the backend which utilized the proposed methodology can leverage the methods described in section 4.4.7 regarding chunks and game state distribution. This enables it to scale beyond the object size limitations of the utilized datastore, as seen by the results of the experiment. Backends that are based on the Athlos framework reach state limitations only when subjected to resource limitations or daily database operation quotas. It would be interesting to see how the Athlos-based backend would have behaved with an increased quota. Nevertheless, the data obtained from this experiment shows that the methods proposed within the Athlos framework allow MMOG backends to feature very large, expandable game worlds, and nurture the possibility of supporting theoretically infinite game states.

7.2.4 Hypothesis 4

When using the proposed framework, the time taken to retrieve a sub-state of a game world remains constant regardless of the world's full size.

The fourth hypothesis is explored in section 6.4.2, which describes an experiment (E4) that studies how the time taken to load a single chunk is affected by the size of the full state. A specialized algorithm is used to generate the game state in the form of terrain chunks of various sizes, forming the full state of the game world. In each of these game world sizes, a single chunk is requested as a sub-state, recording the time taken to load it. The results of this experiment show that the time taken to retrieve a sub-state of the game world remains constant despite exponential increases in its size when using the chunk-based method proposed by the framework. While being limited to a certain MMOG implementation and to relatively low full state sizes, the results establish a trend that could continue to larger game state sizes and which may be extrapolated to other types of games, ultimately confirming this hypothesis.

7.2.5 Hypothesis 5

The development of scalable MMOG backends using the proposed framework simplifies the development process and results to lower effort and time taken to develop an MMOG.

This hypothesis is studied in section 6.5, where different metrics are considered regarding the evaluation of software development effort. As mentioned, there is a large variety of metrics that can help determine the effort undertaken to develop a software product. Many of these are complex metrics that involve several parameters and which entail task management and collaboration principles. Most of these metrics are overly complex and hence not suitable for the case studies developed. A simpler, yet popular indicator for estimating effort in software engineering is source lines of code (SLOC).

A case study MMOG backend is used for this evaluation, which is implemented using both a non-Athlos and an Athlos-based methodology and can therefore be used to compare the effort undertaken between the two projects. To better comprehend the effort required to build the two versions of the backend, the SLOC measures are divided into two groups – efforted lines (manually written by the developer) and generated lines (automatically generated by the tools

of the framework). This makes it possible to distinguish which of these lines required an effort to write and thus allows the estimation of the effort required to produce each version. The results show that even though the Athlos-based implementation of the backend included more lines of code, the vast majority (79%) of these were automatically generated by the framework's code generator. On the other hand, the non-Athlos implementation required three times as many SLOC to be manually written. Even though the experiment's scope is limited to a specific MMOG backend, its results illustrate the potential of the proposed methodology to reduce development effort – and subsequently the time taken to realize an MMOG backend.

7.2.6 Hypothesis 6

The proposed approach produces high-quality, readable, maintainable, and re-usable code.

Experiment E9 addresses the sixth hypothesis by measuring the quality of the code in two different MMOG backends that were developed with the Athlos framework, as well as another approach. The quality of the code in the two versions of either backend is measured using CK metrics, which can evaluate many object-oriented aspects of the code's design. The results obtained from this experiment are mixed at best, showing that the quality of the code produced by the framework is in fact worse compared to an alternative approach in one of the two backends. In the second case study, the results obtained are mixed and inconclusive. The usefulness of these results is limited by the relatively narrow scope of the tests conducted. Despite these limitations, an interpretation of the results is still possible and must be made because it can significantly aid future improvements and developments in the proposed methodology. The Athlos-based implementations scored worse compared to their alternative approach in terms of object coupling (CBO), response for class value (RFC), and cyclomatic complexity (WMC). Given this knowledge, efforts can be undertaken to improve the quality of the produced code with regard to these metrics. Having proven that the tested alternative approach can fare better in terms of code quality compared to the solutions constructed using the proposed framework, hypothesis 6 remains unproven. It is believed that improvements to the code generation tools provided by Athlos could eventually enable improvements in code quality.

7.3 Addressing the technical challenges

Apart from the hypotheses, several other technical challenges are addressed by this research. As there are many challenges related to many aspects of development, the discussion in this section focuses only on the most important of these challenges.

7.3.1 Challenge 1

Can a generic model be created and used for all types of games and game genres?

In section 4.3 a generic model is proposed which can be used to handle various data modeling requirements in several types of MMOGs. As demonstrated through the development of multiple case studies, during which no significant data modeling issues emerged, this model is applicable to a broad set of games. This is mainly owed to the separation and categorization of world types, which is the main driver behind adapting the model to various games. While this model is generic enough to be applied to various use cases and MMOG genres, it is impossible to predict the modeling requirements of all existing games – or even worse, of games that have not yet been invented. Furthermore, there are no known evaluation methods or metrics with which the success and suitability of such models can be quantified.

With these in mind, a different path is taken to solve this challenge. An attempt is made to circumvent the limitations of fully static data models by leveraging a multi-layered model which provides static abstractions at one level but also allows dynamic adjustments to be made based on each game's implementation. Using this approach, it is first possible to customize, and ultimately extend the model to ensure that it captures the requirements of as many games as possible. The methodology proposed by this thesis uses a generic model to capture the requirements of all the case studies described, but also enables dynamic adjustments to be made in each backend implementation. It is impossible to say with certainty that even such a dynamic model will be able to handle the modeling requirements of all existing and future games. In the end, it would be impractical to provide a simple yes or no answer to this question given its complex nature and the infinite possibilities that may exist. With these parameters in mind, it is argued that to the extent of current knowledge and technology, it is possible to utilize a generic, *expandable*, and *dynamically adjustable* model to fulfill the data requirements

of most MMOGs.

7.3.2 Challenge 2

How can the technical limitations of serverless environments be dealt with and what types of methods and tools must be developed to enable MMOG backends to run on these environments?

One of the most prominent technical limitations of serverless systems is the bounded execution time of the services they provide (Donkervliet et al. 2020). This limitation is introduced in these systems by design, as it enables them to scale more efficiently by employing stateless, containerized web services, often provided through HTTP. In the context of MMOG backends, other types of services have been utilized, mostly involving remote procedure calls or the use of direct connections which also offer lower latency. While these technologies may be more suitable for MMOGs, they are more challenging to scale compared to containerized services. This thesis provides methods with which online gameplay can be provided through these stateless, containerized services. Firstly, they are de-coupled from their corresponding containers, allowing the use of a single model for services that can be deployed in containers or using other technologies. Moreover, code generation tools are used to automatically create technology-specific code components linking these services with the underlying infrastructure. This allows developers to focus on service and gameplay logic rather than the implementation of individual technologies used to containerize the services – in turn expediting and simplifying their development, introducing modularity to the system, and promoting code reusability. As demonstrated in the case studies, this approach can enable the deployment of services for different types of MMOG backends and on different types of technology. Within the proposed framework, several serverless technologies have been implemented as proof-of-concept: Google's App Engine Standard and Flexible, and Google Cloud Functions. Furthermore, it is assumed that the standardized nature of these technologies can accommodate the introduction of more serverless technologies in the future.

A second technical limitation of serverless environments is their lack of support for bi-directional communications. Services employed in these environments typically have a limited lifespan, as discussed above, to accommodate efficient scalability. They must therefore execute their processing operations during this lifespan, without offering the ability to establish long-term con-

nections between the client and the server. On the other hand, multiplayer games can greatly benefit from the use of bi-directional, persistent connections, as these enable a seamless, continuous stream of state updates to reach the players and generally offer lower latency. While the properties of serverless systems appear to conflict with the requirements of MMOG backends, there are ways in which they can be converged. Within the proposed methodology, it is possible to define bi-directional, streaming services for serverless technologies which support them. Some serverless technologies, such as App Engine's Flexible, and Amazon's Lambda Functions include support for utilizing bi-directional communications over HTTP through the use of specific APIs – such as App Engine's or AWS's WebSockets APIs. While these are relatively new additions, there seems to be a growing trend toward the adoption of such technologies in serverless systems. By using various abstractions provided within the proposed methodology, such as the definition of platform-agnostic services and the state update mechanism, developers may be able to leverage the advantages of bi-directional connections within serverless environments.

Another issue that arises when working with certain serverless systems is their lack of support for concurrent operations and the lack of standardization of this feature across different commodity cloud platforms. While these environments are designed to support concurrent requests, they typically lack the facilities to enable the straightforward instantiation and execution of disjoint threads within the same request. Some of these environments have been improved to allow concurrent operations in the last few years. For instance, Amazon's Lambda Functions enable concurrency through the definition of concurrency controls, which associate a certain function with a number of reserved and provisioned instances that can execute operations in parallel. Similarly, Google's App Engine, Firebase, and Cloud Functions allow the creation of new threads using language-specific provisions, such as the `ThreadManager` API for Java or `asyncio` for Python. All of these facilities are relatively new features for these environments. As it seems, cloud providers have given more attention to providing support for concurrency within such services. Despite the non-standardized nature of running concurrent operations across different public clouds, the proposed methodology includes tools that leverage their benefits, and which may be adapted in the future to work in conjunction with specific concurrency APIs.

Finally, the use of serverless environments is typically associated with higher latency due to the increased number of layers – and thus overheads – they incorporate. This contradicts the

requirements for low latency in multiplayer games. Through the experience of developing the feasibility and case studies in this thesis, it is reported that this association is purely circumstantial. For instance, using low-level sockets on a dedicated backend is guaranteed to provide lower latency than using HTTP functions in a serverless backend within the same data center, mainly due to lower overheads. However, this difference in latency is minimal – in the order of single millisecond digits. While a few milliseconds of additional latency may make or break the performance of some high-performance games, the experiments conducted in section 6.3 show that more attention should be placed on optimizing game-specific logic and services. These aspects have a much more severe impact on performance if implemented inefficiently compared to the latency induced by the additional overheads of serverless technologies. Regardless, the proposed methodology enables efficient state and runtime management through various methods and algorithms, as demonstrated in the case studies and proven using the results of various experiments in chapter 6. In the worst-case scenario, it allows the definition of platform-agnostic MMOGs, which can be easily modified to work in dedicated or IaaS environments, should the additional latency of serverless technologies not suffice for specific high-performance games.

7.3.3 Challenge 3

How can consistency and performance be balanced to ensure a good QoE?

Consistency and performance are two properties that are at constant odds with each other, especially in terms of persistence, because higher consistency is typically associated with slower operations, and vice versa. The key to providing a good balance between the two is the utilization of manageable consistency, or different levels of consistency, as argued by several related works in section 2.8.5. In practical terms, this may entail the use of an architecture such as the one proposed in section 4.4.3, which employs both higher (i.e. cache) and lower-performance (i.e. database) persistence options. The combination of these two types of systems can enable MMOGs to attain the necessary consistency without significantly hindering performance. It is strongly suggested that game developers utilize caching systems for continuous, rapidly-occurring operations such as those typically seen when players issue actions that alter the game's state. Such systems can offer strong consistency at high performance. Specifically for commodity clouds, the use of distributed caching systems can provide such strong consistency at massive scales. Meanwhile, for other less-demanding, less frequently occurring operations, or

operations which do not require strong consistency – such as updating user information or running backup operations – MMOG backends can utilize more persistent options like datastores or databases. The use of distributed caching systems is so important to attain the necessary balance in terms of consistency and performance at a large scale that they are embedded as a component within the proposed architecture. At the same time, the Athlos framework offers developers ways with which data can be easily managed and interacted with depending on the persistence option being utilized (i.e. the Persistence API), thus making it easier to adapt or make changes to the persistence layer without having to deconstruct service logic.

7.3.4 Challenge 4

Can cloud-friendly persistence options be adapted for gaming workloads, and if so, do they need to be complemented with new methods and tools?

The development of the feasibility and case studies provides proof that cloud-based persistence options, such as Google's Cloud Datastore, Firestore, Amazon's DynamoDB, and Microsoft's CosmosDB can be utilized in multiplayer games. However, the feasibility study has uncovered a major hurdle in terms of creating and managing scalable game worlds within cloud-based NoSQL datastores. As seen in the feasibility study, these datastores each have their own object size limitations which makes it impossible to expand game worlds beyond a certain size that is dependent on the datastore's policy and the game's implementation. To circumvent this limitation, this thesis proposes (a) a disjoint entity-terrain/level state to reduce interdependency between the two, and (b) the use of chunks. Chunks divide the world into sections of adjacent cells that can be grouped together and managed as a single datastore item, thus offering a way to create and store massive game states without reaching the object size limitations of datastores. Furthermore, it provides the facilities and algorithms with which these chunks can be efficiently retrieved and formed into *contextful* states, without the need for developers to explicitly program such behavior. Based on the results of various experiments, it is proven that these methods enable the use of such persistence options to create massive and expandable game states.

7.3.5 Challenge 5

Are built-in tools such as load balancing and resource provisioning mechanisms provided in serverless environments adequate for developing and servicing MMOG backends with a good QoE?

Insights into the performance of load balancing and resource provisioning mechanisms of serverless environments can be inferred from the results obtained in section 6.3. Some serverless environments offer the opportunity for developers to customize the load balancing and resource provisioning configurations of their MMOG backends. For instance, Google's App Engine Flexible allows developers to select a certain type of scaling, target certain thresholds of performance, and so on, allowing developers to tailor their backend based on each specific game. The results obtained from the aforementioned experiments show that the algorithms employed by these environments operate efficiently, scaling MMOG backends to higher numbers of instances to meet the demand, balancing the workload among these instances, and keeping the latency at a manageable level until resources are fully exhausted. In the future, more research is needed to analyze the performance of these tools and determine their overheads, or any potential limitations in the context of real-time distributed systems like MMOG backends.

7.3.6 Challenge 6

Is it possible to design a framework that utilizes the same architecture for a variety of games regardless of differences in gameplay and architectural components?

The requirements and challenges faced in each game can be quite different. Especially across different types of games, requirements could be so different that it would be impossible to design a framework that can provide facilities and support for all the features implemented in such games. Nevertheless, this thesis proposed the use of a dynamic game model that distinguishes the game world and entity types based on how they can be managed in terms of the game state. For instance, the use of grid-based worlds can enable many types of games in which entities can only operate inside a grid of coordinates, whereas uniform worlds allow such entities to move freely. It is believed that using the three proposed world types, as well as an expandable and customizable game model, many types of games can be supported. This

is certainly true for all the case studies developed so far. Even though these are targeted to a specific group of games, they demonstrate that it is feasible to support different types of worlds, with wildly different characteristics and gameplay requirements. Furthermore, the proposed architecture and all of its components were used in all three case studies without encountering issues. This demonstrates that it is generic enough to handle a variety of MMOGs. Even in unforeseen cases where this architecture is not adequate, it can still be expanded to introduce additional components which connect to the backend's runtime or other components to handle specific workloads. As an example, it may be possible to expand this architecture in the future to accommodate the use of Edge Computing technologies, which may offer improved performance. Without discovering any evidence pointing to the contrary, it is argued that the models, methods, and tools proposed in this framework are capable of handling a large variety of games, regardless of their differences, even though there is no quantifiable way to fully substantiate this claim.

7.3.7 Challenge 7

Is it possible to support the development of MMOG backends on both IaaS and serverless environments with the same models, methods, and tools?

The different methodologies used during the feasibility study described in chapter 3 posed the question of whether a single methodology can be utilized to develop MMOG backends regardless of where they are to be deployed. Using a unified development methodology for MMOG backends can provide many advantages, including (a) using common models, methods, tools, and development practices across all implementations that may simplify and expedite their development, (b) using a unified software scalability approach, (c) offering chances to compare, analyze, and optimize implementations regardless of their deployment target. Consequently, this is explored within this thesis through an attempt to unify the development of MMOG backends that can be deployed on different commodity cloud layers – or even off the cloud altogether.

Firstly, games are abstractly defined using the platform and technology-agnostic definitions which include a game model and API. These components can be defined by forming relationships between data and declaring services that can be used to manipulate them – thus allowing

players or other types of users to interact with the game. The game editor discussed in section 4.5.2 ensures that these game definitions remain consistent with the selected deployment environment, while also allowing developers to change their selected environment by adjusting the defined services to meet the requirements. The definitions are subsequently used by the code generation tool described in section 4.5.3 to create boilerplate projects. These implement many of the technology-specific components of each deployment approach. In addition, many components of the architecture are abstracted to allow the definition of logic and relationships without tying them to concrete technologies. Some important examples of these abstractions are the state update and event mechanisms and the game and persistence APIs. The proposed methodology also offers the use of many standardized technologies and tools to solve important problems which are not specific to particular games. For instance, the chunk-based method standardizes the retrieval and management of scalable states regardless of the game being implemented and allows developers to focus on what to do with the state once it is retrieved, rather than having to implement the facilities to retrieve it.

Another example of such standardization is the use of Protocol Buffers as a serialization medium, which helps reduce development overheads by employing a proven and battle-hardened technology to help solve the problem of serialization. The use of PB remains relatively hidden within concrete implementations and does not require developers to delve into game-specific serialization mechanisms or even obtain an in-depth understanding of how Protocol Buffers work.

Lastly, the proposed methodology offers a degree of customization to several aspects. For instance, the proposed model can be customized to include additional types of data or be expanded to meet the modeling requirements of different MMOGs. The code produced by the code generator for the concrete MMOG backend implementations can also be customized at any level, allowing full development control. In fact, many of the functions produced by the code generator include markers that provide suggestions and hints for developers to encourage them to customize their code. Through the combination of abstraction, standardization, and customization, this thesis describes a novel methodology including a pipeline of processes that can produce MMOG backends that can be deployed on various types of layers in commodity clouds, using a single, unified set of methods and tools.

7.4 Limitations

This section discusses the limitations of this thesis in terms of the development methodology used and the research practices employed to evaluate the proposed methodology.

7.4.1 Development methodology

While the software development framework presented in this thesis provides a plethora of advantages for the creation of scalable MMOG backends, several drawbacks can also be identified. Firstly, the use of specific technologies for some aspects of development is debatable. For instance, the Athlos framework works by utilizing core principles included within a specific serialization mechanism – Protocol Buffers. Both empirical and quantitative evidence suggests that among various serialization options, Protocol Buffers offers the best performance and message sizes. However, more efficient options may exist now or in the future, putting to question the use of this serialization mechanism within the Athlos framework. For the time being, Protocol Buffers are considered to be one of the best options for serialization due to their widespread use and support, their provision for generic data messages, low message sizes, and high performance. In the future, Athlos may evolve to provide the option to utilize a variety of serialization mechanisms, including JSON or others, to ensure that it becomes independent from a specific serialization scheme.

Meanwhile, Protocol Buffers are also selected because of their support for creating platform-agnostic services within Google's RPC system (gRPC). These two tools work in concert to provide ways with which services and their subsequent requests and responses can be abstractly modeled and then generated as code in select programming languages. While gRPC provides an efficient, high-performance method of networking for MMOG backends that are deployed on IaaS or dedicated environments, the framework's dependence on this tool is seen as a limitation. Consequently, other networking alternatives may be provided as options alongside gRPC in the future. To this end, other networking tools have also been explored but are not mentioned as they remain experimental and outside the scope of this thesis.

The projects developed using the IaaS approach could also be improved significantly. Despite supporting these types of backends, this thesis focuses exclusively on serverless environments

and therefore does not explore any methods and tools with which MMOG backends can be scaled by utilizing IaaS services. Various commodity cloud services exist which can support the scaling of MMOG backends on non-serverless technologies. At this moment, the Athlos framework does not support any of these cloud services, even though it is compatible with them.

One of the hardest challenges is how to enable the automatic integration of past, implemented versions of a game project with new boilerplate versions. Currently, there are no tools within the framework, or within any other known framework which utilizes code generation which supports this feature. The reliance of the Athlos framework on modeling information within game definitions makes it susceptible to this kind of problem, for two main reasons. Firstly, the creation and modification of data types and services rely on declaring them within definitions and then generating their code. While this improves abstraction, it complicates their management as different versions of the same type, featuring different data attributes or relationships, can exist across different versions of the same project. Secondly, developers may find it challenging and time-consuming to integrate newly-generated code with their previous implementations because some versions may include additional elements and some may not. As a result, this may cause friction during the development process and especially at iterations of the development cycle where many changes are made between versions. While developers can manage this by making small, incremental changes in each version, this is not an ideal solution and further work is needed to improve this process.

Finally, the results of code quality experiments reported in section 6.6 show that the code produced by the framework's tools as a boilerplate MMOG backend could be significantly improved in terms of quality. Such improvements may improve the readability, maintainability, and reusability of the code, thus leading to higher-quality MMOG backends.

7.4.2 Research methodology

The research methodology followed and the results obtained from the experiments in various studies are also susceptible to several limitations. First and foremost, the performance and scalability experiments which are described in chapter 6 do not reach the scales seen in commercialized MMOGs due to resource and budget limitations. Unfortunately, it was not possible

to work around these limitations without exceeding the budget for this project. Even though the results present the potential outcome of what would happen had these resources been available and established a trend that would be expected to continue at larger scales, it is not possible to predict the outcome of experiments conducted at such scales with certainty.

Secondly, the presented models, methods, and tools are validated using a limited number of case studies. It remains unknown whether the proposed methodology can handle other, different types of games in terms of both development practices, as well as providing the necessary performance and scalability. Due to time constraints, it would be impractical to go beyond a certain number of studies in an attempt to validate the framework with as much accuracy and detail as possible. In this thesis, three different cases are studied, with which a variety of MMOGs can be associated. Despite that, further exploration is needed to determine the feasibility of the proposed methodology for a more diverse set of games.

The use of a single commodity cloud in the case studies, as well as in the deployment approaches of the framework itself are also research limitations of this project. These limitations make the results obtained from the experiments conducted in chapter 6 less generalizable to other types of clouds, even though cloud providers tend to offer similar services. Due to time and resource constraints, it was deemed impractical to utilize services from multiple commodity clouds. To an extent, the feasibility study presented in chapter 3 provides the underpinnings of such a generalization. However, the results presented in this thesis must be cautiously interpreted, as little information is available about the behavior and usefulness of the proposed methodology when it is coupled with other commodity clouds.

Another limitation in terms of the research methodology is that the measurements taken from the experiments described in chapter 6 are of limited precision, as the trials were repeated a specific number of times – usually three to five times for each trial. The precision of the simulation harnesses discussed in section 6.3 is also limited. These simulations are conducted using specific configurations which are preset to study specific conditions and to determine the behavior of MMOG backends under certain circumstances. While the simulation harnesses enable the definition and use of various configurations – e.g. linear or spike demand configurations, larger or shorter play delays, shorter join times, and more – these are not utilized within this thesis to their full extent, and it is therefore impossible to conclude how MMOG backends would behave under these conditions.

Perhaps one of the most important limitations of this thesis is the fact that the proposed methodology is not evaluated by actual game developers. This makes it impossible to know whether the intended end users of the framework would find it useful and relevant, be able to understand how it works, and leverage its benefits to develop MMOG backends. This aspect of the framework's evaluation would also shed more light on the requirements of MMOG backends from the perspective of game developers, allow an understanding of how they perceive the process, and ultimately lead to improvements in the proposed methodology.

Such an evaluation would entail face-to-face tutorials, interviews, participant observation, and surveys with a specific target group. In addition, the human resources required to carry out such an evaluation are limited, putting to question the extent of its usefulness compared to the risks and costs associated with its execution. The evaluation of Athlos is expected to continue in the future when resources are made available.

Chapter 8

Conclusion

> "*Computers are useless. They can only give you answers.*"
>
> Pablo Picasso

This chapter summarises the thesis' contributions, discusses its potential impact, and discusses the plans for future work.

8.1 Contributions and content

This thesis introduced the research topic in chapter 1 by describing the general concepts and enumerating various problems that exist in the area of MMOG backend development. This chapter further motivated the development of MMOG backends on commodity clouds by introducing their characteristics and peculiarities, which have hindered their deployment on such infrastructure in the past, and by establishing the need for such research to take place given the requirements of modern, commercialized MMOGs. The scope and research objectives of the thesis were also outlined, which were expanded and explored in the thesis, alongside several hypotheses and technical challenges.

Chapter 2 presented previous works in the area of MMOG backend development. It discussed the state of the art in the development of such applications and attempted to identify important aspects that could facilitate their deployment on commodity clouds. These aspects are used as criteria to perform a systematic review of past research works. This aims to analyze the different

approaches, exposing their advantages and limitations, and ultimately the opportunities and challenges arising from each approach. Through the analysis of past works, trends in research were established by revealing existing patterns in the methods utilized by researchers and developers. Such trends further motivated this thesis by identifying gaps and potential research directions that have yet to be explored.

Based on the insights of the analysis in chapter 2, the potential of utilizing commodity clouds for the deployment of MMOG backends remains relatively unexplored. Chapter 3 reports an exploration of the potential of commodity clouds for hosting MMOG backends. A feasibility study is conducted where the facilities of various commodity clouds are explored, and a simple prototype MMOG backend is deployed and tested on these environments. This uncovers various challenges and limitations of these platforms and motivates their exploration by enumerating additional challenges that are explored in later sections.

Chapters 2 and 3 contributed new knowledge in this area by offering an in-depth analysis of the state of the art, and by making the first steps in studying the suitability of commodity clouds for MMOG backend deployment. These contributions were further enhanced in chapter 4, which presented a novel framework encompassing models, methods, and tools which can be used to develop scalable MMOG backends on commodity clouds. This chapter first motivated the creation of such a framework by discussing the limitations of existing methods and tools, and by presenting a conceptual MMOG case study. This aids the presentation of various elements, such as a common model and architecture to be used by various types of MMOG backends, and the development of various methods to improve their performance and scalability. These are incorporated and utilized through various tools which aim to standardize the development of MMOG backends hosted on commodity clouds, ultimately leading to a more efficient software engineering process.

In chapter 5, the framework was empirically evaluated through the development of three prototype MMOG backends. This aimed to investigate the suitability of the proposed methodology, report the experience of developing MMOG backends which are hosted on various types of commodity cloud environments and explore the challenges mentioned in chapters 1 and 3. The implementation of these prototypes establishes a proof-of-concept for the proposed methodology and helps address many of its weak points in its early stages of development.

Chapter 6 further advanced the contributions of this thesis by quantitatively evaluating the

proposed methodology. While this evaluation was mainly centered around the hypotheses mentioned in chapter 1, other exploratory experiments were also conducted to explore the usefulness of specific methods or tools, or to uncover useful information about the behavior of MMOG backends. Furthermore, the proposed methods and tools are evaluated in terms of the development effort required to produce MMOG backends and the quality of the code generated by the framework. Even though they are limited by various factors, the results presented in this chapter showed that MMOG backends developed using the proposed framework and which are deployed on serverless commodity cloud environments can sustain larger player numbers while satisfying certain latency thresholds, can allow game states to be scaled beyond the normal capabilities of the underlying technologies, and be managed more efficiently. The results also showed that the proposed framework helps reduce development effort while pointing out various weaknesses with respect to the quality of the code produced.

In chapter 7, the results of the evaluation were discussed with respect to the hypotheses established in section 1. This chapter also analyzed various technical challenges associated with the development of MMOG backends and their deployment on commodity clouds. The contributions of the proposed methodology with respect to each of these challenges were analyzed, suggesting possible solutions to these problems. Within this chapter, the limitations of the development and research methodologies were also identified.

Finally, the present chapter concludes the thesis by summarizing its contributions and content, presenting a general discussion on its impact, and ultimately offering glimpses into future work which may address its limitations.

8.2 Impact

The research presented in this thesis aspires to have an impact in the areas of software engineering, cloud-based, and real-time distributed systems. Despite initiating research towards a new direction, the contributions of this project barely begin to scratch the surface in terms of deploying scalable, non-parallel software systems with real-time constraints – such as MMOG backends – on commodity clouds. The peculiar nature of these applications has hindered their deployment on serverless commodity clouds, with almost all research focusing either on dedicated or non-elastic options. Using the Athlos framework, a novel suite of models, methods, and

tools proposed in this thesis, software engineers can develop scalable MMOG backends which can be deployed on commodity clouds, on various layers and services. This framework aims to improve the development of these applications by defining them as abstract entities which can ultimately be implemented in specific technologies chosen by the developers. This can standardize the process of development, especially with regard to leveraging cloud resources – something for which very little has been accomplished so far. In terms of software engineering, the results obtained in this thesis show that standardization can lead to faster, more efficient, and more convenient software development.

At the same time, MMOG backends can enjoy seamless scaling by leveraging cloud resources within serverless environments. This can have an impact on how software engineers design systems, and the effort required to realize, maintain, and support them. Many of these advantages can resonate at the business level. For instance, game development studios and companies may leverage the elasticity of serverless clouds to create MMOGs that are substantially more economical than their predecessors. Furthermore, start-up game companies may find it easier to enter an already congested market due to lower barrier to entry requirements – mostly associated with infrastructure costs. This is achieved by eliminating the need to take economic risks to cover basic investments in other types of dedicated infrastructures and by providing the ability to quickly respond to fluctuations in demand. In addition, start-ups may have to seek out a lower number of employees as serverless clouds reduce the need for infrastructure specialists and system analysts, especially at the early stages of a project.

The impact of resuming research in this direction may also have positive effects on the environment. Despite the fact that data centers have a relatively large carbon footprint, it is argued that the elastic nature of serverless clouds may reduce power consumption as machines adjust their resource utilization based on demand – helping cut down extra costs associated with the overprovisioning of cloud resources.

Furthermore, the contributions of this thesis make it possible to envision the use of abstraction, standardization, and customization through dynamically modeled elements to solve various other kinds of problems related to software engineering and distributed computing. The framework discussed in this thesis can easily be adapted to work with most types of online applications, including enterprise applications. This may be especially useful for other types of distributed, soft real-time systems which have similar constraints (e.g. real-time, bi-directional

communications, non-parallel execution, scalable software architectures, etc.), such as virtual collaboration environments, video conferencing tools, instant messaging, and social platforms. In addition, it may be possible to adapt large-scale simulations, of which MMOG backends are a subset, to be deployed on serverless commodity clouds. This may include simulations covering various sectors and industries: medical, physical, chemical simulations, and more, as well as other areas of human interest, such as disaster relief, socioeconomic, geopolitical, or military simulations.

Finally, the contributions of this research may facilitate the design of higher-quality multiplayer online games which can be used as educational tools. Such games may be used in computer science education or other areas to educate students and other individuals on specific skills through interactive environments.

8.3 Future work

Amid various contributions, this project has also established the foundations for future research to take place in the same domain. The Athlos framework which incorporates the proposed approach is still in the early stages and can be significantly improved. The development methods described within this framework may be adapted, or even substituted for better alternatives. Perhaps the most imminent work in terms of the development methodology is the improvement of the quality of the code produced by the framework's tools. The results obtained in section 6.6 illustrate the need to improve the quality of the generated code in Athlos-based projects. To this end, more methods of evaluating code quality must also be explored, as this may make more information available which would help address this problem more efficiently.

This also applies to evaluating development effort and estimating project costs. Even though the results obtained in section 6.5 demonstrate that Athlos-based MMOG backends require less effort to develop, more research is needed to generalize these results and to further optimize the development process. To this end, a broader variety of cost estimation techniques could be used to improve the accuracy of these results. For instance, development effort can be derived from the number of requirements of a project or the features that have to be implemented. Another approach is to divide a project into activities or tasks that are assigned a certain amount of time to complete. This allows developers to estimate the effort and time required to complete

a project. This can be combined with human-centered estimation techniques that calculate the person-hours or person-days needed to complete tasks or the entire project. Such techniques (e.g., backlogs with user stories, planning poker, etc) can be combined with a risk assessment and time loss estimation to provide a clearer picture of the effort required to develop a software project.

The framework is currently available as a beta version which only supports Java-based projects on a select number of serverless environments. In future versions, Athlos can be expanded to include support for a variety of other languages and deployment options, allowing developers to leverage the framework to its full potential. Furthermore, it is unknown whether specific programming languages or environments may affect the costs associated with the use of this approach. Future studies may focus on evaluating the suitability of the framework in incorporating new technologies with a relatively low cost and assessing the ethical impacts they may have on the development and use in commercial MMOG backends.

The potential of the proposed approach can also be further explored in terms of performance and scalability. The experiments conducted in chapter 6 can be extended to facilitate larger scales using more computing resources in the future. These can be combined with a larger variety of experimental setups, involving different case studies, simulation configurations using different parameters, and the inclusion of other variables which may help improve the reliability of the data and therefore help better understand the internal behavior of MMOG backends. In terms of isolated experiments, a more thorough evaluation of specific methods and tools can be conducted, with more data samples over a broader variety of circumstances – ultimately improving the accuracy of the data and reducing the uncertainty of the measurements. The interpretation of data in these experiments may benefit from a statistical analysis which may help infer relationships and patterns and ultimately lead to more objective conclusions.

New additions may also help enhance the framework's usefulness and direct new research into this area. For instance, the versioning and code integration problem mentioned in section 7.4.1 is a general problem that may be faced by different frameworks employing code generators. Further study is needed to map this area of software engineering and utilize appropriate methods and tools to optimize this process. The proposed approach may also benefit from the creation of a domain-specific language for game definitions. At the moment, the Athlos editor requires the use of graphical components to define entities, which is a rather slow process. For more

advanced developers, the use of such a language and additional facilities to utilize it within the existing tools may expedite the definition of MMOG backends, their models, and APIs. Another domain in which this research can be extended is the evaluation of a greater variety of MMOG types. The presented thesis investigates a relatively simple set of games that have diverse gameplay requirements, but which could be considered similar in terms of state representation. Athlos can also support the development of a greater variety of games – for instance limited-scale, fast-paced environments like First-Person Shooter games or games rendered in 3D virtual worlds. A future study may attempt to test the suitability of the framework in developing a more diverse set of games, benchmark their performance, and ultimately provide more substantial proof of its usefulness for developing commercial MMOG backends. Lastly, the use of containerization systems like Docker can enable MMOG backends to be deployed on top of underlying containers. In conjunction with orchestration platforms like Kubernetes, it may be possible to orchestrate MMOG backends to be deployed on multiple instances, scale seamlessly, and without supervision while leveraging the advantages of serverless computing as well as the customizability possible in dedicated servers. Such advances in software engineering and cloud computing may be coupled with products like Google's Agones or Amazon's GameSparks, opening a new frontier in research for scalable MMOG backends on commodity clouds.

Chapter 9

Appendices

9.A Feasibility study data

The following tables present data recorded during the feasibility study experiments discussed in section 3.6.

	AWS EC2	GCP App Engine	Azure VMs
1	166	94	143
2	167	97	145
3	180	92	143
4	166	94	144
5	165	93	144
6	166	92	143
7	165	106	145
8	166	107	143
9	166	102	146
10	166	95	146
Average	**167.3**	**97.2**	**144.2**

Table 9.1: Base latency data in the feasibility study experiment.

	Maximum board size in cells^2
AWS DynamoDB	98
GCP Cloud Datastore	158
Azure CosmosDB	229

Table 9.2: Maximum board size for each datastore, in cells^2

	AWS EC2	GCP App Engine	Azure VMs
1	249	799	698
2	243	873	385
3	409	470	355
4	224	544	315
5	227	391	290
6	411	457	315
7	416	367	305
8	384	331	260
9	359	323	278
10	405	411	262
Average	**332.7**	**496.6**	**346.3**

Table 9.3: Latency data for the `/create` service in the feasibility study experiment.

	AWS EC2	GCP App Engine	Azure VMs
1	243	384	285
2	190	310	282
3	226	730	168
4	192	755	170
5	185	556	296
6	222	634	251
7	214	550	232
8	186	319	281
9	180	621	198
10	180	687	185
Average	**201.8**	**554.6**	**234.8**

Table 9.4: Latency data for the `/join` service in the feasibility study experiment.

	AWS EC2	GCP App Engine	Azure VMs
1	343	914	951
2	790	1414	445
3	338	1484	766
4	326	1060	296
5	816	1338	369
6	333	786	498
7	325	806	541
8	801	1412	398
9	815	1397	745
10	798	921	541
Average	**568.5**	**1153.2**	**555**

Table 9.5: Latency data for the `/list` service in the feasibility study experiment.

	AWS EC2	GCP App Engine	Azure VMs
1	179	214	821
2	184	141	188
3	175	253	188
4	173	249	151
5	177	155	169
6	176	127	195
7	175	134	178
8	177	143	182
9	174	194	186
10	173	158	196
Average	**176.3**	**176.8**	**245.4**

Table 9.6: Latency data for the `/getState` service in the feasibility study experiment.

	AWS EC2	GCP App Engine	Azure VMs
1	177	335	179
2	183	132	174
3	179	125	180
4	177	406	181
5	175	141	176
6	179	190	187
7	176	209	152
8	181	204	188
9	172	126	150
10	170	144	191
Average	**176.9**	**201.2**	**175.8**

Table 9.7: Latency data for the `/play` service in the feasibility study experiment.

9.B Model

The following sections describe various items within the game model which are of secondary importance to this thesis.

9.B.1 Players (NX)

Perhaps one of the most prominent types are the players themselves, which are arguably one of the most common types found in all games. A *player* is defined as an actor or character that takes part in a game. Players may either be human or be controlled by an artificially intelligent script (*Non-Player Characters* - NPCs). In most games – but not all – they may have control over one or more entities and can issue actions that affect the game's state. In this particular model, the player type records information that is not gameplay-specific, such as personal information. This means that no game state is stored within this type. The default attributes of this type can be used to identify and authenticate the player, with the possibility of extending this model to include more data according to the requirements of each game.

9.B.2 Teams (NX)

A *team* is a collection of players who usually work together towards a common goal. Teams may or may not be formed according to each game's mechanics, and therefore this data type can be ignored in cases where it is not needed. While the default team type features only basic information, it can be expanded to include game-specific data like a team flag, color, collective resources being gathered by the players of the team, and much more. The attributes and relationships for these two data types are shown in figure 9.1.

9.B.3 Positioning and direction (NX)

Position and Movement

The positioning of an entity within a game world depends on the world type. In grid-based worlds, the `MatrixPosition` type is used, which models a position using integer-based row

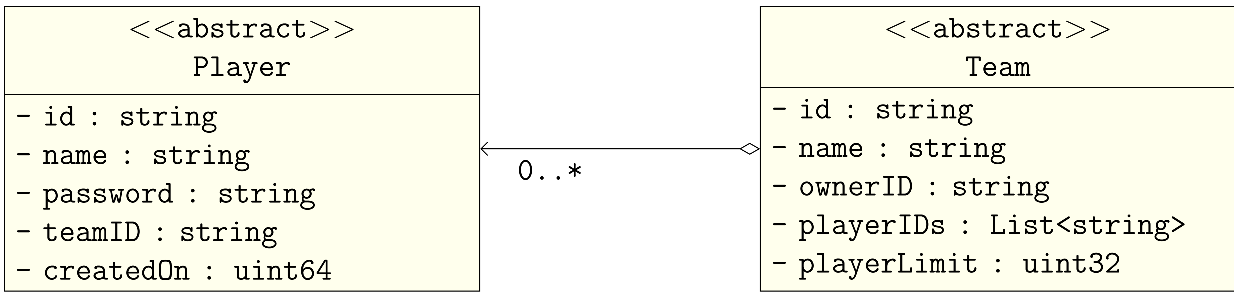


Figure 9.1: The player and team models.

and column coordinates. Matrix positions are also utilized to manage terrain, as terrain is also modelled using a grid structure. Conversely, entities within uniform worlds are positioned using the `GeoPosition` data type, which models positions in a Euclidian space, using floating-point x, y, and z coordinates. These two types of positioning are the inverse of each other when the height axis is excluded, which makes it very easy and efficient to convert between them: $(x, y) = (col, row)$. The use of these different types is motivated by the fact that both of them incorporate numeric data types, making it very easy to accidentally provide incorrect parameters to methods. Based on experience, this can lead to hard-to-detect logical errors in code, which can be easily avoided by using specifically-typed parameters. To make it easier to work with these types, the framework includes support for converting between them, as well as performing various useful space calculations like distance, area inclusion and exclusion, and more.

A change in position is modeled by movement data types. In uniform worlds this is trivial, as an entity can move in its facing direction by a specific distance. In square-grid worlds, movement is limited to `FORWARD`, `BACKWARD`, `LEFTWARD`, and `RIGHTWARD`, as modelled by the `Movement4` enumerator. Similarly, `Movement6` enumerates the possible movements in a hexagonal-grid world: `FORWARD`, `FORWARD_RIGHT`, `BACKWARD_RIGHT`, `BACKWARD`, `BACKWARD_LEFT`, and `FORWARD_LEFT`.

Direction and Rotation

To model the direction of an entity that exists in a uniform world, the framework uses an integer number representing the angle at which the entity is facing, relative to the north. The absolute angle of an entity (θ) can be found by dividing the relative angle value ($angle$) by 360 and using its remainder to find direction ($\theta = \text{angle} \ \% \ 360$). While a simpler, ranged system could be used

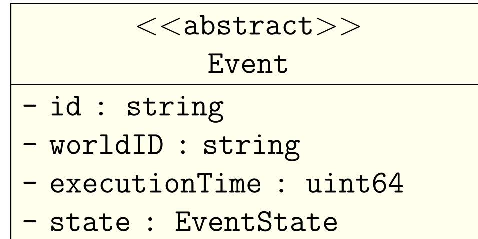


Figure 9.2: The default event model.

to find the relative angle, such as forcing the value to stay in the range 0-360, the used approach allows games to store the change in angle, which enables more complex simulations involving angular velocity. For square grid-based worlds, the direction is represented using `Direction4`, an enumerator type that includes the values `NORTH`, `EAST`, `SOUTH`, and `WEST`. For hexagonal grid-based worlds, the `Direction6` enumerator is used, with the possible values being `NORTH`, `NORTH_EAST`, `SOUTH_EAST`, `SOUTH`, `SOUTH_WEST`, and `NORTH_WEST`. A change in direction for grid-based worlds is modelled by the `Rotation` enumerator, which has two possible values: `CLOCKWISE`, and `COUNTER_CLOCKWISE`.

9.B.4 Events (X)

An *event* is something that occurs during the game, at a specific point in time and within a specific world. Events may alter the state of a world and can be scheduled to run at specific points in time or at intervals apart. Events make it possible to alter the game's state without the explicit involvement of players. The default model for events is shown in figure 9.2. Events are utilized within an event mechanism described in section 4.4.3 which manages their execution and state.

9.B.5 Actions (X)

An *action* is an event carried out by a player inside a specific world and may or may not be related to an entity. Actions can have an effect on the game state, including its entities and terrain, but are typically *stateless*. This provides an opportunity to provide them as stateless, RESTful services in an MMOG backend (Doglio 2015). Actions do not have a default model, as each concrete action is considered game-specific. They are, however, a common feature that is found in all games.

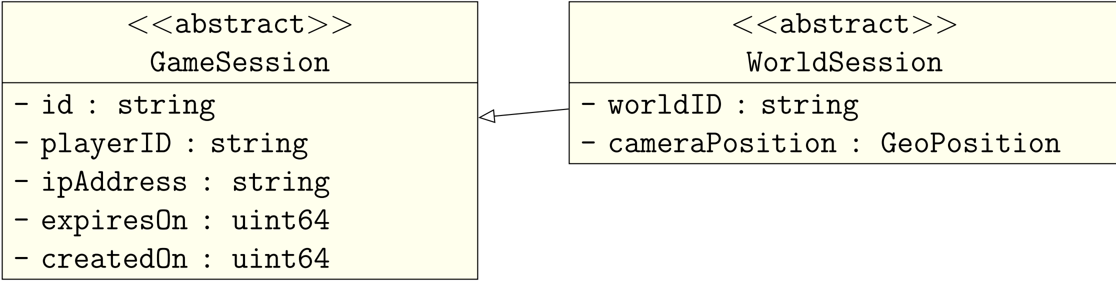


Figure 9.3: The default game and world session models.

9.B.6 Game sessions (NX)

A *game session* is a user authentication and management data type which identifies a player who has connected to a game's backend. Game sessions are created by the backend to track players who have been authenticated. In this context, this can mean authentication using security credentials like a username or password, or simply connecting to the backend using a valid game client. Use cases of game sessions include operations that are not related to a particular world, like changing account information, sending text messages, and so on.

9.B.7 World sessions (NX)

A *world session* is an extension of the game session and enables the identification of a player who has joined a specific world. World sessions can be used to track player progress in worlds, verify their actions during gameplay, or to compose partial states. The default models for game and world sessions are shown in figure 9.3.

9.B.8 Services (X)

Developers can define various types of services that enable them to manage their MMOG backends, or provide extra functionality on top of actions to clients. A *service* is similar to an action, but is not related to a specific action made by the player and is meant to serve a specific request to access or manage information. For instance, a client may need to retrieve the list of available worlds so that the player can explore them. This is not directly related to neither a specific world nor its state and therefore cannot be considered an in-game action. To handle

such types of interactions services can be defined, consisting of their corresponding request and response models. The default service type does not include any attributes or functionality.

9.B.9 Requests and Responses (X)

Services can be modelled using a pair of request and response models, where the *request* type models the expected input to the service, and the *response* type models output expected from the service. Requests are instantiated by clients and sent to the backend to access or manage information, whereas responses are instantiated by the backend to provide this information to the clients. Services can have unique pairs of request-response models, or can utilize common models that are also used by other services.

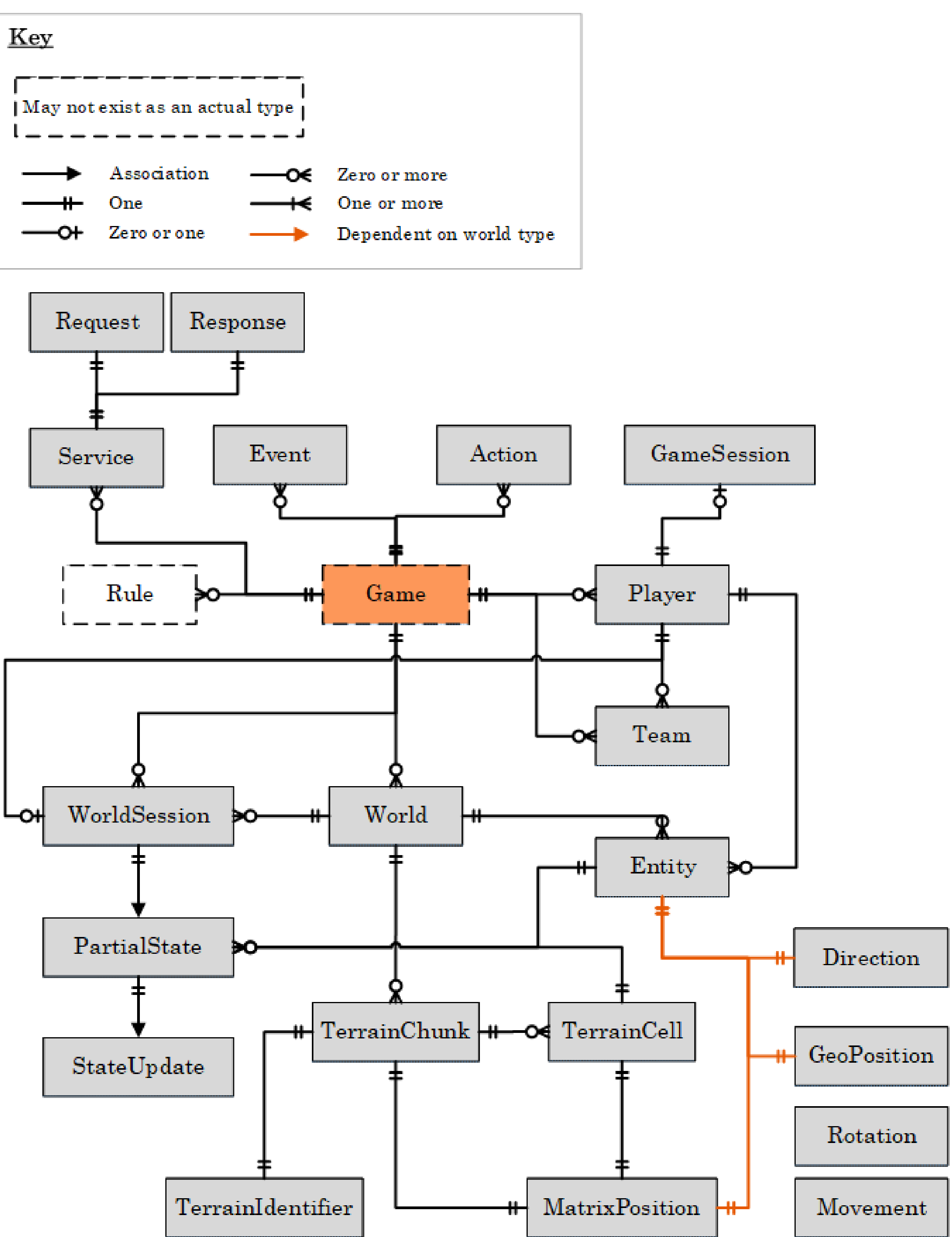


Figure 9.4: A Crow's foot diagram presenting the data model of the Athlos framework and the relationships between various types. Attributes are omitted for brevity.

9.C State API diagram

The diagram in the next page presents the Athlos State API and its components, which is described in section 4.4.7.

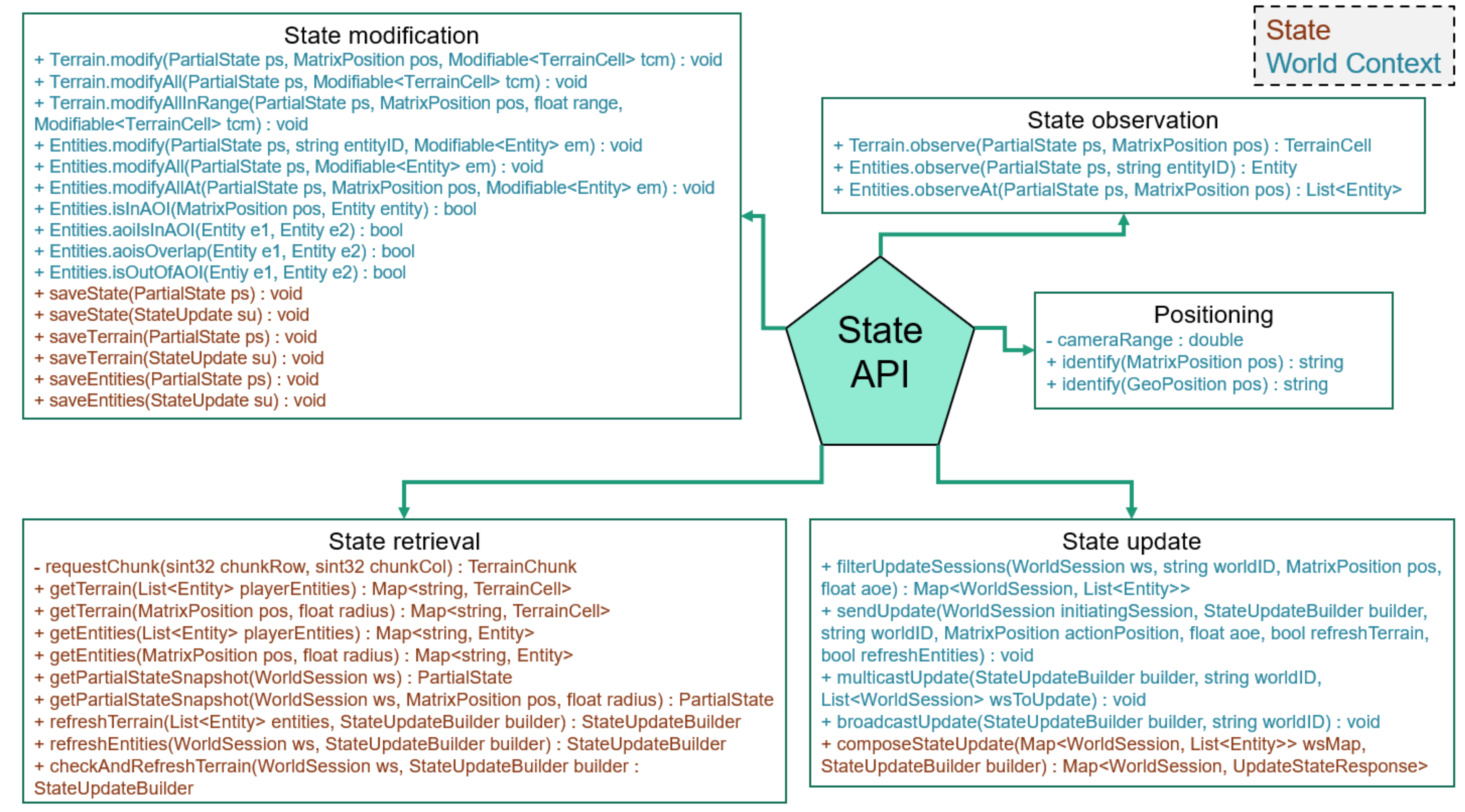


Figure 9.5: The state API, including methods from the State and World Context classes.

9.D Mars Pioneer case study code

This appendix contains code from the Mars Pioneer case study discussed in section 5.2.

```
1 public class MPWorldSessionDAO implements WorldBasedDAO<MPWorldSession> {
2     @Override
3     public boolean create(MPWorldSession object) {
4         String uuid = UUID.randomUUID().toString();
5         object.setId(uuid);
6         Objectis.create(object, uuid);
7         new Thread(() -> Firestorm.create(object, uuid)).start();
8         return true;
9     }
10
11    @Override
12    public boolean update(MPWorldSession object) {
13        Objectis.update(object);
14        new Thread(() -> Firestorm.update(object)).start();
15        return true;
16    }
17
18    @Override
19    public boolean delete(MPWorldSession object) {
20        Firestorm.delete(object);
21        return true;
22    }
23
24    @Override
25    public MPWorldSession get(String s) {
26        return Objectis.get(MPWorldSession.class, s);
27    }
28
29    public List<MPWorldSession> getMany(List<String> ids) {
30        return Objectis.getMany(MPWorldSession.class, ids);
31    }
32
33    @Override
34    public MPWorldSession getForWorld(String worldID, String itemID) {
35        final List<MPWorldSession> items = Objectis.filter(MPWorldSession.class)
36                .whereEqualTo("worldID", worldID)
37                .whereEqualTo("id", itemID)
38                .fetch().getItems();
39        if (items.size() == 0) {
40            return null;
41        }
42        return items.get(0);
43    }
44
45    @Override
46    public Collection<MPWorldSession> listForWorld(String worldID) {
47        return Objectis.filter(MPWorldSession.class)
48                .whereEqualTo("worldID", worldID)
49                .fetch().getItems();
50    }
51
52
```

```
53
54      public MPPlayer getPlayer(final String worldSessionID) {
55          final MPWorldSession worldSession = Objectis.get(MPWorldSession.class,
        worldSessionID);
56          if (worldSession == null) {
57              return null;
58          }
59          return Objectis.get(MPPlayer.class, worldSession.getPlayerID());
60      }
61
62      public Collection<MPWorldSession> listForPlayer(final String playerID) {
63          return Objectis.filter(MPWorldSession.class)
64                  .whereEqualTo("playerID", playerID)
65                  .fetch().getItems();
66      }
67
68      public MPWorldSession getForPlayerAndWorld(final String playerID, final
        String worldID) {
69          final List<MPWorldSession> items = Objectis.filter(MPWorldSession.class)
70                  .whereEqualTo("playerID", playerID)
71                  .whereEqualTo("worldID", worldID)
72                  .limit(1)
73                  .fetch().getItems();
74          if (items.size() == 0) {
75              return null;
76          }
77          return items.get(0);
78      }
79
80  }
```

Listing 9.1: The implemenation of the WorldSession DAO in Mars Pioneer – MPWorldSessionDAO.java

```
1       public MPPartialStateProto getPartialStateSnapshot(MPWorldSession
        worldSession, MatrixPosition position, float radius) {
2           final MPPlayer player = DBManager.player.get(worldSession.getPlayerID())
        ;
3           return MPPartialStateProto.newBuilder()
4                   .putAllEntities(getEntities(position, radius))
5                   .putAllTerrain(getTerrain(position, radius))
6                   .setTimestamp(System.currentTimeMillis())
7                   .setWorldSession(worldSession.toProto())
8                   .setResourceSet(
9                           ResourceSetProto.newBuilder()
10                                  .setFood(player.getFood())
11                                  .setMetal(player.getMetal())
12                                  .setSand(player.getSand())
13                                  .setWater(player.getWater())
14                                  .build()
15                  )
16                  .build();
17      }
```

Listing 9.2: Customizations made to the `getPartialStateSnapshot()` method – WorldContext.java

```
1     public void handleMessage(BuildFarmRequest request) throws IOException {
2
3         long t = System.currentTimeMillis();
4         long start = t;
5
6         //Retrieve the session:
7         final MPWorldSession worldSession = DBManager.worldSession.get(request.
      getWorldSessionID());
8
9         //Verify session:
10        if (worldSession == null) {
11            send(BuildResponse.newBuilder()
12                    .setStatus(BuildResponse.Status.INVALID_WORLD_SESSION)
13                    .setMessage("INVALID_WORLD_SESSION")
14                    .build());
15            return;
16        }
17
18        final MPPlayer player = Auth.verifyWorldSessionID(worldSession.getId());
19        if (player == null) {
20            send(BuildResponse.newBuilder()
21                    .setStatus(BuildResponse.Status.CANNOT_BUILD)
22                    .setMessage("NOT_AUTHORIZED")
23                    .build());
24            return;
25        }
26
27        Objectis.create(new SessionValidationResult(System.currentTimeMillis()-t
      ));
28        t = System.currentTimeMillis();
29
30        final MatrixPosition actionPosition = request.getPosition().toObject();
31
32        //Get state:
33        final MPPartialStateProto partialState = State.forWorld(worldSession.
      getWorldID()).getPartialStateSnapshot(worldSession, actionPosition, 20);
34
35
36        Objectis.create(new StateRetrievalResult(System.currentTimeMillis()-t));
37        t = System.currentTimeMillis();
38
39        //+++++++++++++ Resource rules ++++++++++++++
40        //Check resources:
41        if (player.getSand() < BuildingType.FARM.getSandCost() ||
42                player.getFood() < BuildingType.FARM.getFoodCost() ||
43                player.getWater() < BuildingType.FARM.getWaterCost() ||
44                player.getMetal() < BuildingType.FARM.getMetalCost()) {
45            send(BuildResponse.newBuilder()
46                    .setStatus(BuildResponse.Status.INSUFFICIENT_FUNDS)
47                    .setMessage("NOT_ENOUGH_RESOURCES")
48                    .build());
49            return;
50        }
51
52        //+++++++++++++ Terrain-building rules: ++++++++++++++
53
54        //Cannot build anything on lava:
```

```
55         final MPTerrainCellProto cell = State.Terrain.observe(partialState,
       actionPosition);
56         if (cell.getType() == CellType.LAVA_CellType) {
57             send(BuildResponse.newBuilder()
58                     .setStatus(BuildResponse.Status.CANNOT_BUILD)
59                     .setMessage("CANNOT_BUILD_ON_LAVA")
60                     .build());
61             return;
62         }
63 
64         //Cannot build a farm on rock or ice:
65         if (cell.getType() == CellType.ROCK_CellType || cell.getType() ==
       CellType.ICE_CellType) {
66             send(BuildResponse.newBuilder()
67                     .setStatus(BuildResponse.Status.CANNOT_BUILD)
68                     .setMessage("CANNOT_BUILD_FARM_ON_ROCK_OR_ICE")
69                     .build());
70             return;
71         }
72 
73         //Cannot build too far away from a hub:
74         boolean hubWithinDistance = false;
75         for (MPEntityProto e : partialState.getEntitiesMap().values()) {
76             if (e.hasBuildingEntity()) {
77                 if (e.getBuildingEntity().getBuildingType() == EBuildingType.
       HUB_EBuildingType && e.getPlayerID().equals(player.getId())) {
78                     double distance = e.getPosition().toObject().distanceTo(
       actionPosition);
79                     if (distance <= 20) {
80                         hubWithinDistance = true;
81                     }
82                 }
83             }
84         }
85 
86         if (!hubWithinDistance) {
87             send(BuildResponse.newBuilder()
88                     .setStatus(BuildResponse.Status.CANNOT_BUILD)
89                     .setMessage("TOO_FAR_FROM_HUB")
90                     .build());
91             return;
92         }
93 
94         //Cannot build on a cell that already has a building:
95         for (MPEntityProto e : partialState.getEntitiesMap().values()) {
96             if (e.hasBuildingEntity()) {
97                 if (e.getPosition().toObject().equals(actionPosition)) {
98                     send(BuildResponse.newBuilder()
99                             .setStatus(BuildResponse.Status.CANNOT_BUILD)
100                             .setMessage("BUILDING_EXISTS")
101                             .build());
102                     return;
103                 }
104             }
105         }
106 
107         //Cannot build if prerequisite buildings are not owned:
108         final ArrayList<EBuildingType> prerequisites = BuildingType.FARM.
```

```
    getPrerequisites();
109         for (EBuildingType prerequisiteType : prerequisites) {
110             boolean owned = false;
111             for (MPEntityProto e : partialState.getEntitiesMap().values()) {
112                 if (e.hasBuildingEntity()) {
113                     if (e.getBuildingEntity().getBuildingType() ==
    prerequisiteType) {
114                         owned = true;
115                         break;
116                     }
117                 }
118             }
119             if (!owned) {
120                 send(BuildResponse.newBuilder()
121                         .setStatus(BuildResponse.Status.CANNOT_BUILD)
122                         .setMessage("PREREQUISITE_MISSING")
123                         .build());
124                 return;
125             }
126         }
127 
128         Objectis.create(new RuleProcessingResult(System.currentTimeMillis()-t));
129         t = System.currentTimeMillis();
130 
131         //----------------------------------------END OF RULE CHECKING
    ------------------------------------------------
132 
133         //Deduct resources:
134         player.setFood(player.getFood() - BuildingType.FARM.getFoodCost());
135         player.setWater(player.getWater() - BuildingType.FARM.getWaterCost());
136         player.setSand(player.getSand() - BuildingType.FARM.getSandCost());
137         player.setMetal(player.getMetal() - BuildingType.FARM.getMetalCost());
138 
139         //Create building:
140         BuildingEntity building = new BuildingEntity();
141         building.setDirection(Direction4.NORTH);
142         building.setBuildingType(EBuildingType.FARM_EBuildingType);
143         building.setAreaOfInterest(BuildingType.FARM.getAreaOfInterest());
144         building.setPosition(request.getPosition().toObject());
145         building.setPlayerID(player.getId());
146         building.setWorldID(worldSession.getWorldID());
147 
148         DBManager.buildingEntity.create(building);
149         DBManager.player.update(player);
150 
151         send(BuildResponse.newBuilder()
152                 .setStatus(BuildResponse.Status.OK)
153                 .setMessage("OK")
154                 .build());
155 
156         Objectis.create(new StateModificationResult(System.currentTimeMillis()-t
    ));
157         t = System.currentTimeMillis();
158 
159         //Define and send the state update:
160         final StateUpdateBuilder stateUpdateBuilder = StateUpdateBuilder.create
    ().addUpdatedEntity(building);
161         State.sendUpdate(worldSession, stateUpdateBuilder, worldSession.
```

```
    getWorldID(), actionPosition, 10, false, false);
162
163         Objectis.create(new StateSendResult(System.currentTimeMillis()-t));
164         Objectis.create(new TotalResult(System.currentTimeMillis()-t));
165     }
```

Listing 9.3: Implementation for the BuildFarm action. – BuildFarmWebSocket.java

```
1  public class SellBuildingStub extends BinaryWebSocketClient {
2
3      private MPClient client;
4
5      public SellBuildingStub(MPClient client) throws IOException,
       WebSocketException {
6          super("ws://localhost:8080/api/action/sellBuilding");
7          this.client = client;
8      }
9
10     @Override
11     public void onReceive(byte[] bytes) {
12         try {
13             SellBuildingResponse response = SellBuildingResponse.parseFrom(bytes
       );
14             handleResponse(response);
15         } catch (InvalidProtocolBufferException e) {
16             e.printStackTrace();
17             client.getGameCanvas().showMessage(e.getMessage());
18         }
19     }
20
21     public void handleResponse(SellBuildingResponse response) {
22         if (response.getStatus() == SellBuildingResponse.Status.OK) {
23             System.out.println("Building sold by " + client.getWorldSession().
       getId());
24         } else {
25             client.getGameCanvas().showMessage(response.getMessage());
26             System.err.println(response.getMessage());
27         }
28     }
29
30 }
```

Listing 9.4: Implementation of a WebSocket service stub in Mars Pioneer – SellBuildingStub.java

9.E aMazeChallenge case study code

This appendix contains code from the aMazeChallenge case study described in section 5.3.

```
1 public class GetState implements AthlosService<GetStateRequest, GetStateResponse
    > {
2     @Override
3     public GetStateResponse serve(GetStateRequest request, Object...
    additionalParams) {
4
5         //Check world session ID:
6         if (request.getWorldSessionID().isEmpty()) {
7             return GetStateResponse.newBuilder()
8                     .setStatus(GetStateResponse.Status.INVALID_DATA)
9                     .setMessage("INVALID_WORLD_SESSION")
10                    .build();
11        }
12
13        //Verify world session:
14        final AMCWorldSession worldSession = Auth.verifyWorldSessionID(request.
    getWorldSessionID());
15        if (worldSession == null) {
16            return GetStateResponse.newBuilder()
17                    .setStatus(GetStateResponse.Status.INVALID_DATA)
18                    .setMessage("INVALID_WORLD_SESSION")
19                    .build();
20        }
21
22        //Get the challenge grid:
23        final Challenge challenge = DBManager.challenge.get(worldSession.
    getWorldID());
24        final Grid grid = challenge.getGrid();
25
26        final MemcacheService memcache = MemcacheServiceFactory.
    getMemcacheService();
27
28        //Get the game state:
29        Game game = (Game) memcache.get("game_" + challenge.getId());
30        final List<PickableEntity> pickables = game.getPickables();
31        final Map<String, PlayerEntity> playerEntities = game.getPlayerEntities
    ();
32
33        final AMCPartialStateProto.Builder builder = AMCPartialStateProto.
    newBuilder();
34
35        //Pickable entities:
36        for (PickableEntity pickable : pickables) {
37            builder.putEntities(pickable.getId(), pickable.toGenericProto().
    build());
38        }
39
40        //Player entities:
41        for (Map.Entry<String, PlayerEntity> entry : playerEntities.entrySet())
    {
42            builder.putEntities(entry.getKey(), entry.getValue().toGenericProto
    ().build());
43        }
```

```
44
45         //Players:
46         for (Map.Entry<String, AMCPlayer> entry : game.getAllPlayers().entrySet
    ()) {
47             builder.putPlayers(entry.getKey(), entry.getValue().toProto().build
    ());
48         }
49
50         //World sessions:
51         for (Map.Entry<String, AMCWorldSession> entry : game.
    getPlayerWorldSessions().entrySet()) {
52             builder.putWorldSessions(entry.getKey(), entry.getValue().toProto().
    build());
53         }
54
55         //Retrieve the partial state:
56         builder
57                 .setTimestamp(System.currentTimeMillis())
58                 .setWorldSession(worldSession.toProto())
59                 .setGrid(grid.toProto())
60                 .addAllActivePlayers(game.getActivePlayers())
61                 .addAllQueuedPlayers(game.getQueuedPlayers())
62                 .addAllWaitingPlayers(game.getWaitingPlayers())
63                 .build();
64
65         return GetStateResponse.newBuilder()
66                 .setStatus(GetStateResponse.Status.OK)
67                 .setMessage("OK")
68                 .setPartialState(builder.build())
69                 .build();
70     }
71 }
```

Listing 9.5: The GetState service in aMazeChallenge – GetState.java

```
1 public class PlayerEntityDAO implements WorldBasedDAO<PlayerEntity> {
2     @Override
3     public boolean create(PlayerEntity object) { return Firestorm.create(object)
     != null; }
4
5     @Override
6     public boolean update(PlayerEntity object) { Firestorm.update(object);
    return true; }
7
8     @Override
9     public boolean delete(PlayerEntity object) { Firestorm.delete(object);
    return true; }
10
11     @Override
12     public PlayerEntity get(String id) { return Firestorm.get(PlayerEntity.class
    , id); }
13
14     @Override
15     public PlayerEntity getForWorld(String worldID, String itemID) {
16         final QueryResult<PlayerEntity> fetch = Firestorm.filter(PlayerEntity.
    class)
17                 .whereEqualTo("worldID", worldID)
18                 .whereEqualTo("id", itemID)
```

```
19                  .limit(1)
20                  .fetch();
21          if (fetch.hasItems()) {
22              return fetch.getItems().get(0);
23          }
24          return null;
25      }
26
27      @Override
28      public Collection<PlayerEntity> listForWorld(String worldID) {
29          return Firestorm.filter(PlayerEntity.class)
30                  .whereEqualTo("worldID", worldID)
31                  .fetch().getItems();
32      }
33
34      public Collection<PlayerEntity> listForPlayerAndWorld(String playerID,
        String worldID) {
35          return Firestorm.filter(PlayerEntity.class)
36                  .whereEqualTo("worldID", worldID)
37                  .whereEqualTo("playerID", playerID)
38                  .fetch().getItems();
39      }
40      public Collection<PlayerEntity> listForPlayer(String playerID) {
41          return Firestorm.filter(PlayerEntity.class)
42                  .whereEqualTo("playerID", playerID)
43                  .fetch().getItems();
44      }
45  }
```

Listing 9.6: Implementation of the player entity DAO in aMazeChallenge – PlayerEntityDAO.java

9.F Libraries

The following sections describe libraries related to the Athlos framework mentioned in section 4.5.5

9.F.1 Firestorm

Firestorm is an object-oriented data access API for Google's Cloud Firestore, a popular cloud-based data persistence option provided within the Google Cloud Platform and associated with Firebase. Firestore is Google's leading NoSQL database, replacing the Cloud Datastore, and allows real-time access to data at a "*global scale*" (Google 2021). It contains many useful features related to persistence, such as scalability, query support, facilities for use within serverless backends, and real-time updates. Firestorm also features an extensive API that enables developers to control how information is stored in the database but provides limited support for mapping an object model to its document-based database – a concept known as Object-Document Mapping (ODM) that is similar to Object-Relational Mapping (ORM) used in relational databases. The use of ODM in backend applications provides several advantages for online applications including query optimization, protection from injection attacks, and the ability to keep a consistent data model. ODM tools and their ORM counterparts allow developers to think about and manipulate data as objects rather than documents, which can simplify and expedite the development process significantly. Surprisingly, such tools are in limited supply and those that exist are targeted toward less capable data stores or mobile applications rather than backends.

While developers can use the default APIs provided by persistence options like Google's Firestore, the use of ODM tools like Firestorm can have significant benefits for MMOG backends, especially in terms of scalability. As seen in figure 9.6, Firestorm automatically converts objects into *documents* and stores them into their corresponding Firestore *collections* based on their class, rather than allowing developers to freely create collections of mixed item types. The grouping of same-type objects into such collections provides better organization, can help maintain consistency and helps in information retrieval. Relationships between documents can be created by either utilizing document IDs, which are set by developers or automatically generated by Firestore, or by using document references – a type of document "address" within the Firestore.

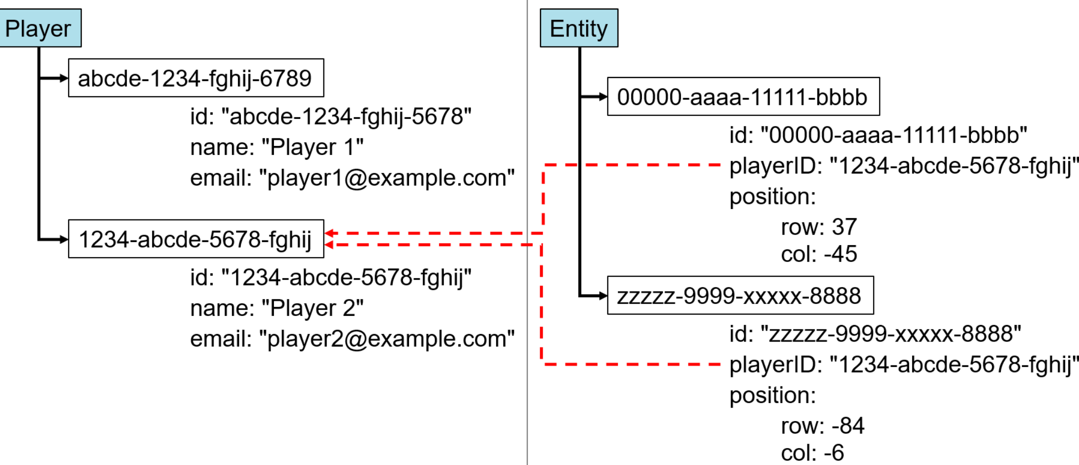


Figure 9.6: A simplified version of the ODM structure created by Firestorm in conjunction with the Athlos model.

Firestorm is currently implemented in Java as a proof-of-concept. It attempts to complement the traditional Firestore API and provide an additional ODM layer on top of its existing functionality. First, the library's components are initialized at the start of the backend's deployment, providing a globally-accessible instance of the library to the backend. This instance is accessible by the server or backend instances throughout the rest of the backend's deployment without the need to further manage any connections. To create an ODM mapping Firestorm uses annotations, pre-processor directives, or dynamic programming techniques to define 'mappable' classes. For instance, the `@FirestormObject` annotation is used in Java to define a class that is to be converted into a document. Firestorm also requires that such classes or any of their parent classes contain a String-based ID field for identification, an empty constructor, and public accessor methods for all their fields. It is also possible to exclude specific attributes from the class mapping by defining them as such. Before utilizing a mapped class within Firestore, developers must register it during the initialization of Firestorm. The registration process checks the class for proper form and identifies its mappable attributes, including those in its parent classes. Once registered, a class can be used in various operations that are defined by the Firestorm API such as creating, deleting, retrieving, listing objects, and more. Firestorm also leverages Firestore's extensive query system to enable object-based queries that filter information and return results directly in the form of objects. These are complemented by object-based transactions and batch writes that can ensure atomic database operations. An important feature that relates to MMOG backends specifically is the ability to attach or detach real-time listeners to documents in the database by using their corresponding objects.

Real-time listeners can offer the ability to listen and then react to changes made on documents in Firestore in real time. This enables the use of a pub/sub communication model, which is an efficient and scalable method to handle state updates. Firestorm also provides a unique mechanism for paginating results easily and by interacting with objects rather than raw information or documents, which is a useful feature in various contexts like viewing long lists of items – such as a list of sessions or available worlds. Finally, Firestorm provides a way to perform asynchronous operations which can help in reducing latency and when running large transactions in time-constrained serverless environments.

While Firestorm is only implemented as a Java library at the moment, its API is abstract and can be implemented in other languages as well. Firestorm itself is designed as an independent tool that is not directly related to Athlos, even though the main idea behind this concept originates from the need to store game-related information efficiently, with minimal development overheads, and with scalability in mind. In its short lifespan, Firestorm has been used experimentally in various projects, including the case studies presented in section 5 and other non-related business applications with relative success. Firestorm is evaluated in terms of scalability and performance in section 6.

9.F.2 Objectis

Many of the problems mentioned with respect to cloud datastores like Firestore are also present in caching systems. These systems are important components of MMOG backends as they can provide quick and resource-efficient access to the game state while maintaining strong consistency. Distributed caching systems provided by public clouds, such as Google's MemoryStore, Amazon's ElastiCache, and Microsoft's Cache for Redis, can enable MMOG backends to access information stored in these caches at high performance and scale. A very popular caching system that is employed by all of the aforementioned cloud services is Redis. Redis is an open-source, in-memory, extensible data store that enables the storage and retrieval of items based on key-value pairings. It includes various data structures such as lists, sets, etc., and is cloud-ready by offering the ability to distribute its database across multiple nodes. Apart from being used as a real-time, strong-consistency datastore, solutions like Redis can be used to stream data from source to destination, allowing backends to utilize the pub/sub communication model.

The structure and design of caching systems like Redis allows information to be accessed very quickly but provides very limited support for more complex data operations and queries. It is therefore left up to developers to realize the necessary mechanisms – such as filtering, transactions, and more – to support their backend's operations. Like persistent data stores, caches like Redis have limited support for ODM, which complicates the development process. Instead of working with objects, developers are forced to identify information manually by providing specific keys in the form of strings or raw bytes. Similarly, when an object needs to be stored in the cache, it has to be either manually serialized into its byte representation or converted into text-based formats like JSON. Furthermore, collections of items within these caches do not support the storage and retrieval of objects directly. Instead, developers are forced to manually create and configure collections of keys, which they can then retrieve to access lists of serialized information. Such processes complicate the development effort significantly and entangle developers in the problems of manual serialization and data design.

To solve these problems, Objectis aims to provide software-based guidelines and tools with which information can be stored and retrieved as objects from the Redis cache. Objectis works by utilizing the same ODM concepts introduced in Firestorm, and by defining a very similar data access API. Like in Firestorm, classes have to be annotated using special directives and then registered to allow Objectis to record their data model. They can then be used in various data operations. Developers do not need to define any entry keys or convert information to its serialized form (or inversely, to its object form), as these processes are handled internally by Objectis. This is achieved by automatically generating a key for each item based on its data type and ID, and by automatically serializing object values into raw bytes. To bridge the gap in managing collections of items, Objectis internally manages IDs that reference specific items within special sets. These sets are used in various operations to directly retrieve lists of objects rather than having to retrieve a set of IDs and then retrieve entries based on these IDs. The information structure created by Objectis looks very similar to the one shown in figure 9.6 for Firestorm, with the exception of data being stored in its serialized form under a single entry rather than containing nested key-value pairs. Furthermore, the filtering API defined in Firestorm is also utilized by Objectis, even though the internal functionality is very different. This allows these two tools to be interoperable, making it easier for developers to learn a single API to work with both datastores and caching systems. While filtering is a relatively efficient process on datastores like Firestore because of their support for indexing, it can be a compu-

tationally expensive process in caches like Redis. Objectis takes this into account by providing multithreaded processing capabilities for environments that support it. With multithreading enabled, Objectis can concurrently process large numbers of items simultaneously to reduce the time taken to retrieve and serialize them. While this approach works well in many contexts, it is not scalable, and cannot be utilized in environments that do not support multithreading – such as some serverless platforms. To help solve the problem of rapidly growing sets of items, Objectis also introduces *special collections* which allows developers to create custom collections of items. These special collections can be populated with a subset of the items found in the set of a particular class, therefore making it easier to manage objects by limiting their scope based on the context. For instance, in an MMOG backend, developers can use special collections to divide entities in a world based on their ability to change their state. Dynamic, stationary, and static entity types can be split into three special collections, allowing the retrieval of a subset of the items that would normally have to be processed.

Like all other tools, Objectis is implemented as a Java library as a proof-of-concept and utilized in the case studies presented in section 5. While its implementation is specific to this programming language, the APIs and concepts utilized are transferable to other languages as well. Coupled with Redis' support for client libraries in many programming languages like Python, .NET, and JS, Objectis can be expanded to support additional software development stacks.

9.F.3 World generation

The process of *world generation* is a rather complicated task that has been left unexplored in this thesis so far. When creating worlds for their games, developers can choose to manually create custom designs using a process known as *level design*. Level design deals with the creation of customized maps or stages that are used in the game. Developers must manually specify the world's boundaries, the state of the terrain in the world, any entities that exist on it, and additional world-specific attributes – known as *environmental design*. Secondly, in cases where it is appropriate, developers must also provide a sequence of events which can unfold within the world under specific cases and/or time periods. For instance, in arcade games like Super Mario Bros created by Nintendo, levels are designed in advance and always feature the same environment and event sequences. The process of level design leads to a fixed

initial environment in which interactive situations can arise to challenge players based on the gameplay. While the process of level design can be radically different depending on the type of game, the main principles of this concept remain the same across games.

Other types of games may have to dynamically generate levels to challenge their players. For instance, in a Sudoku puzzle game, levels would have to be generated randomly to enrich the gameplay experience and keep players engaged. The levels of such games are therefore not compatible with the process of level design which produces a fixed initial state in the world. For such games, developers may opt to utilize special algorithms that incorporate randomness to ensure that each level is unique. The algorithms used to generate levels may be different in each game, but most may employ some form of backtracking algorithm to randomly generate solvable puzzles like Sudoku boards or mazes.

Other games must feature expandable worlds that can extend up to very large limits, or to a theoretically infinite scale. In these games, it is impractical to design the levels, or to generate the entire world's state in one go. Instead, sections of the world can be generated as required by the game – for instance when a player walks into a part of the world for the first time. It is therefore important to have the proper mechanics in place and divide the world into parts so that they can be generated independently. Depending on the game, worlds may expand as players explore them, leading to lower initial loading times and resource usage. The process of generating infinite worlds based on seed values and pre-defined instructions is known as *procedural generation*. Procedural generation helps power many popular games like Temple Run and Minecraft. In Temple Run, relatively simple procedural generation is used to allow players to "run" forever in the game without ever having to face the same in-game situations to execute repetitive actions. In more complex games like Minecraft, procedural generation is used to create the terrain of the world and entities that exist within it on-the-go. Using procedural generation it is possible to create an exact replica of a world by using the same seed value, which generates terrain components like height maps, temperature maps, humidity maps, and more. In this thesis, procedurally generated worlds are prioritized over other types of world creation because they fall in line with the research objectives. These objectives involve the management of world states that can expand to very large scales, are fully persistent, and can be accessed in constant time regardless of size (H3 and H4).

The cost of implementing procedural generation algorithms is relatively high, as they are dif-

ficult to understand and fine-tune. To achieve procedural generation, developers often use *procedural coherent noise algorithms* which can generate sets of coherent values in multiple dimensions. An example of such an algorithm is OpenSimplex, which can generate up to 4 dimensions of gradient noise. The generated noise can subsequently be used to form maps of terrain, or other attributes like temperature that can affect the characteristics of the world. To eliminate the development overheads of working with such algorithms, an experimental library called 'Open Simplex Noise Generator' is used in conjunction with Athlos to enable MMOGs to generate infinitely scalable terrain. The Noise generator works by hiding the underlying implementation of the OpenSimplex algorithm and allowing developers to generate and retrieve values in up to four dimensions and interpolate their ranges through an intuitive set of functions. This library enables the generation of use of context-specific values for each game, without the need to understand the underlying mechanics or implement interpolation calculations to ensure that the generated values stay within their expected ranges. As with other libraries, the noise generator is implemented in Java for proof of concept and is used with in the Mars Pioneer case study discussed in section 5.2. Meanwhile, the algorithms in this library can be easily adapted for other programming languages in the future.

9.G Tool evaluation

The following sections evaluate various tools which are part of the Athlos framework.

9.G.1 Firestorm

The first of these experiments (E10) aims to assess the performance of Firestorm, initially introduced in section 9.F.1. The aim of this experiment is to measure the overheads of this library in terms of the time taken to complete a datastore operation, provide insights into its usefulness in terms of software development effort, and prove its effectiveness in helping realize MMOG backends. A concern regarding this library is whether the overheads of realizing the ODM technique are detrimental to data access performance, as Firestorm adds a significant layer above the traditional API utilized by Google's Firestore.

To measure the performance overheads of Firestorm a generic `Player` class is defined based on the data attributes of its corresponding Athlos model. These attributes and the data associated with them in each object remain constant during the duration of each experimental run. Several frequently used functions provided by the Firestorm library are evaluated, including `create`, `get`, `getMany`, `list`, `update`, and `delete`. For each of these functions, five experimental trials are made with three object operations in each trial, allowing the calculation of average values for the time taken to execute these operations. To determine the library's overheads and to provide experimental control, the trials also include calls made using Firestore's traditional API. During the timespan of these trials, several factors are kept in control, such as the data center location, the data model, the network and device load, and the device utilized. Before each trial is initiated, a clean-up operation is executed to remove any existing documents from the database. In addition, warm-up calls are made to the Firestore database in order to ensure that a connection has already been established. Furthermore, to determine the effectiveness of Firestorm in reducing development effort, the lines of code needed to employ each of the aforementioned functions are measured in both best-case and worst-case scenarios.

The results obtained, shown in table 9.8, indicate that the Firestorm library takes longer to perform some of these operations compared to the traditional API, whereas it performs better in other cases. This is a positively surprising result, as the Firestorm library itself

Average time taken to perform an operation (ms)			
	Firestore API	**Firestorm**	**Difference**
Create	96.20	96.27	+0.07%
Get	89.80	78.67	-12.40%
GetMany	78.13	79.13	+1.28%
List	89.20	86.20	-3.36%
Update	86.47	83.53	-3.39%
Delete	94.80	82.80	-12.66%

Table 9.8: Times taken to perform various operations using the Firestorm library and the Firestore API.

uses the traditional API to communicate with the Firestore. In some cases, Firestorm takes longer to perform these operations. For instance, when creating an object or retrieving many objects identified by their IDs it performs worse than the traditional method. However, in all other cases, Firestorm manages – surprisingly – to reduce the time taken to perform such operations despite the overheads of managing object/document definitions. The reasons for this mixed behavior are not yet fully understood, and therefore a more complex evaluation targeting the performance of this specific library may be carried out in the future. It is argued that the differences between the results obtained for either of the two approaches are statistically insignificant. While this points towards Firestorm producing no discernible overheads in terms of performance, more comprehensive research is required to provide solid foundations to this claim.

To evaluate Firestorm in terms of software development effort an experiment is designed to measure the minimum source lines of code required to perform various operations. In this experiment, Firestorm is evaluated by using its best-case and worst-case code constructs. On one hand, 'best-case' code constructs allow operations to be defined quickly, using one-liner expressions. However, developers may also opt to use more complex expressions when necessary to handle the results of such operations and assign application logic to be executed when they are completed. In such 'worst' cases, a listener pattern is employed to handle these results, which increases the minimum lines of code required. Figure 9.7 compares the minimum SLOC required to perform operations when using the Firestore API, Firestorm's best-case one-liners, as well as its worst-case listener expressions. As the data shows, the traditional Firestore API requires more lines of code in three out of the six operations even when the worse-case constructs of Firestorm are employed. When using simpler, one-liner expressions (best-case), Firestorm is more effort-efficient than the traditional API by a very wide margin. The results, which

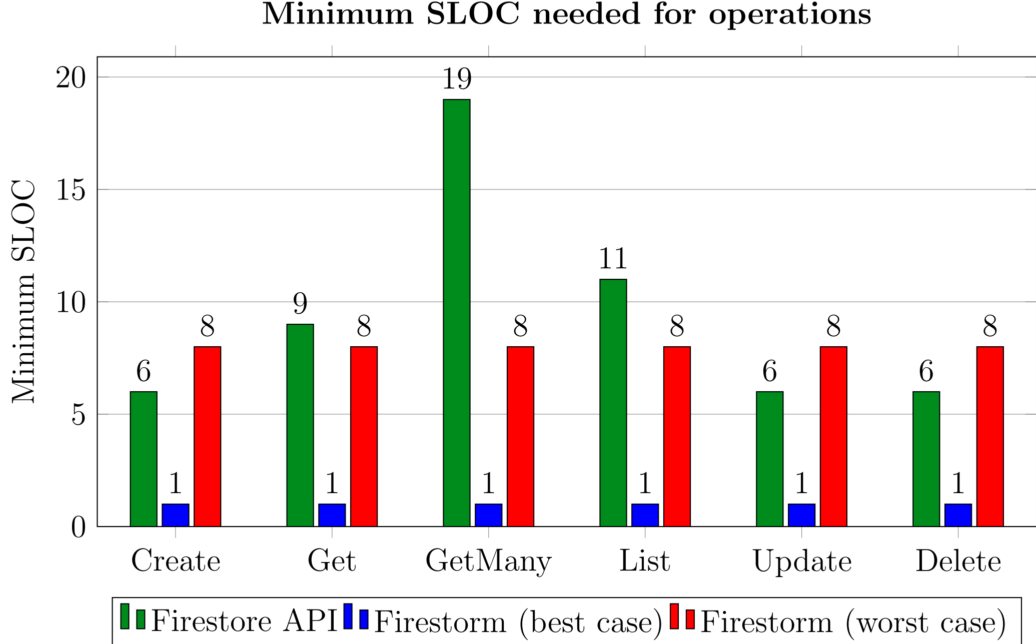


Figure 9.7: A comparison between the Firestore API and the best and worst cases of code used by Firestorm in terms of minimum SLOC required to perform various operations.

are limited in scope, provide evidence that Firestorm can enhance the development process by offering significant reductions in development effort through its encapsulated code, and by automatically managing Object-Document mappings in a performance-aware way.

9.G.2 Objectis

The Objectis library which is introduced in section 9.F.2 and helps provide ODM support for the Redis cache is also evaluated. As it features an API that is similar to that of Firestorm, a code evaluation is avoided and more focus is given to its performance. The performance of two types of batch operations, `write` and `read` is tested in terms of the time taken to perform them. In the first mode, which is the library's default, Objectis uses a single thread of execution to carry out operations. In the second mode, the library automatically creates and manages several threads based on the processor's capabilities. These threads can then be used to perform various tasks in parallel and thus potentially reduce the time taken to perform an operation. In some cases where the use of threads is not advantageous – such as in small numbers of objects – the library can automatically choose to use a single thread of execution. This second mode is thus called Hybrid Multi-Threaded mode (HMT), and can be optionally employed on environments that can support multiple threads.

Time taken to perform batch create operations (ms)			
Number of objects	**Jedis API**	**Objectis**	**Objectis HMT**
10	1.00	1.67	1.33
100	5.33	10.33	4.67
1,000	41.67	83.33	20.00
10,000	340.67	676.67	187.33
100,000	3268.67	6662.33	1861.33

Table 9.9: Results for the time taken to perform creation operations involving different numbers objects using the Jedis API, Objectis, and Objectis' Hybrid Multi-Threaded mode.

Time taken to perform batch read operations (ms)			
Number of objects	**Jedis API**	**Objectis**	**Objectis HMT**
10	1.00	0.67	1.00
100	10.00	3.67	2.67
1,000	55.33	9.00	7.00
10,000	405.00	43.33	27.00
100,000	3592.33	365.33	153.67

Table 9.10: Results for the time taken to perform read operations involving different numbers objects using the Jedis API, Objectis, and Objectis' Hybrid Multi-Threaded mode.

The first part of experiment E11 involves the evaluation of the two operations using numbers of objects ranging from 10 up to 100,000, increasing by a factor of 10 in each run. During these runs, the two modes of the library are tested in terms of the operations listed. To enable a comparison and provide a control element to the experiment, the traditional Java-based Jedis API is also tested. A total of three trials are made for each number of objects, with averages taken out of 5 repetitions in each run. Redis is hosted locally on a computer with 32GB of RAM of which 16GB are allocated for the cache, uses an 11th-generation Intel Core i7 processor, and runs within the Windows Subsystem for Linux. These specifications are kept throughout all the trials in this experiment. More factors are kept in control in an attempt to record valid data. For instance, the data model being used is identical to that being described in experiment E10, while all entities and their attributes are kept identical for the three tested approaches in all runs. A clean-up operation and warm-up calls are made before each run.

The results from this experiment are shown in table 9.9 for batch writes, and 9.10 for batch reads, and illustrated correspondingly in figures 9.8 and 9.9. In terms of batch write operations the Jedis API appears to be more advantageous for smaller numbers of objects as it requires less time to finish these operations compared to Objectis. The single-threaded mode of Objectis has the worst performance out of the three approaches, as it requires the most time over all

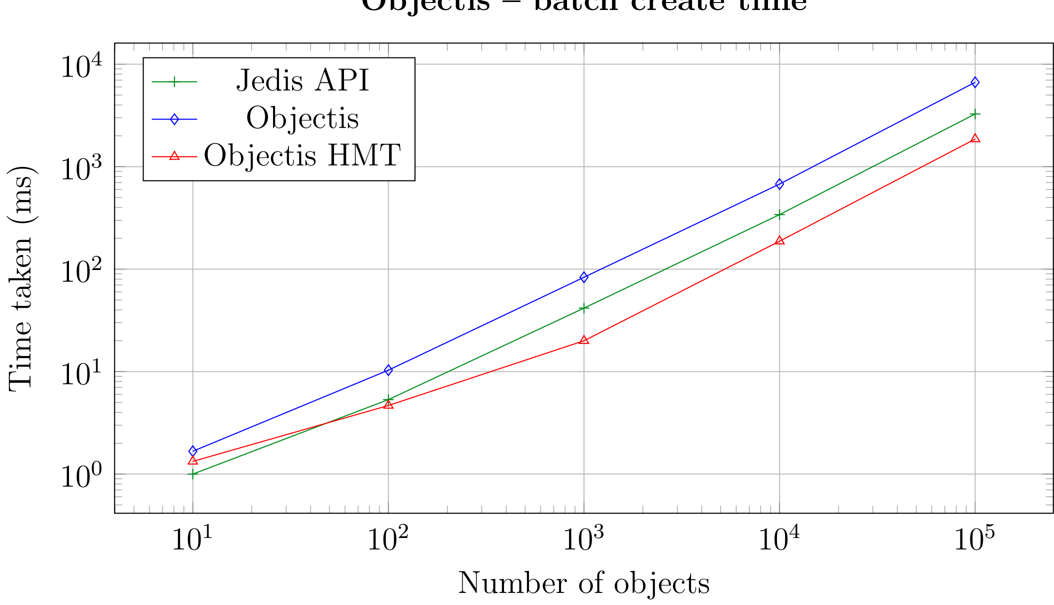


Figure 9.8: A comparison of the time taken to create different numbers of objects when using the Jedis API, or the Objectis library in default or multi-threaded mode.

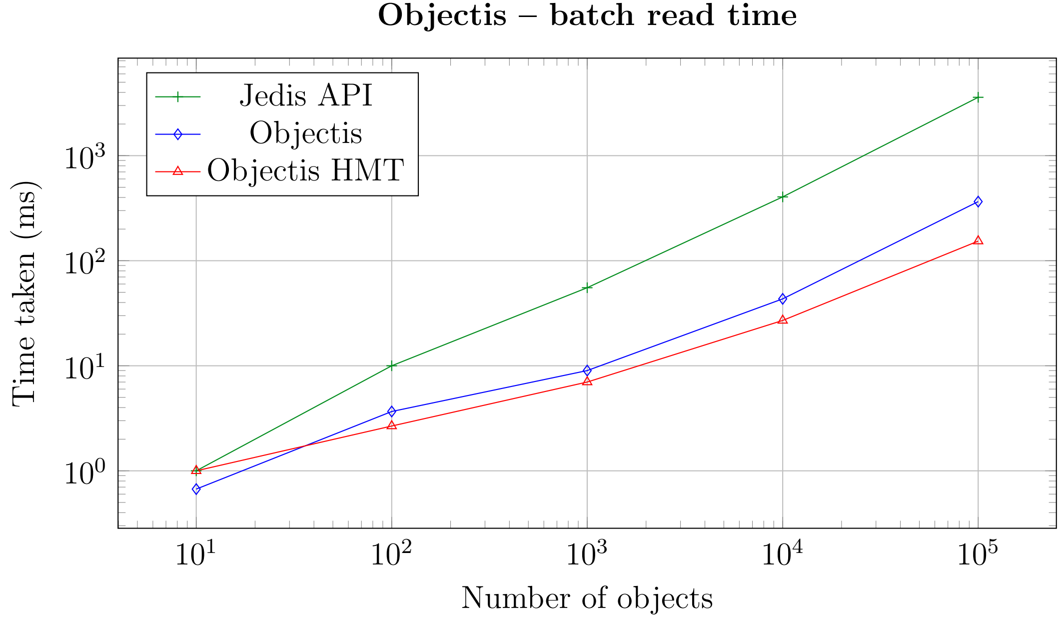


Figure 9.9: A comparison of the time taken to read different numbers of objects when using the Jedis API, or the Objectis library in default or multi-threaded mode.

numbers of objects. Meanwhile, the HMT mode performs moderately at small numbers of objects but outperforms the other approaches at larger scales. In terms of batch reads, the single-threaded mode of Objectis performs better than the two other approaches. However, it is increasingly outperformed by the HMT mode for larger numbers of objects. Meanwhile, the Jedis API has significantly worse performance compared to using either mode of Objectis.

This experiment is relatively simple and small in scope as it evaluates a specific set of operations under specific circumstances. While there are many other approaches to evaluate the performance of caching libraries, batch operations are used as they provide the best insights into the performance of each approach at large scales. Using smaller numbers of objects like those tested in Firestorm would not produce any valuable results as the latency of caches for handling small numbers of objects is negligible. In addition, MMOG backends are expected to perform such operations to persist changes in the state of a game in the cache, after updating the state during their runtime. For smaller, incremental changes to the state, the latency would nonetheless be negligible, thus offering little chance to study the effects of Objectis on performance. While being limited in scope, this experiment shows that concurrent programming techniques can be leveraged to improve the usefulness of various tools provided in MMOG backends, thereby increasing their performance and scalability.

9.G.3 ByteSurge

The first attempt at handling serialization within the proposed framework is to provide a serialization API with which objects can be easily encoded into byte form, communicated across machines, and then converted back into objects at the destination. This API is initially provided using an experimental tool called ByteSurge, mentioned in section 4.4.5 which organizes data using containers and schemas. ByteSurge is implemented in Java for proof-of-concept and allows developers to first define the schema of a message, and then populate it with containers of data corresponding to the items defined in the schema. This tool supports many useful features like the ability to nest data, use collections, data compression, and concurrent operations to improve performance. The definition of schemas also creates a concrete message structure that can be known at both source and destination devices, thus reducing the overall overheads of communication. Another major advantage provided by ByteSurge is the ability to easily pack information and convert it into its byte representation, or to unpack it into objects,

without explicitly working with streams. Even though it provides a more graceful way to handle serialization than streams, ByteSurge has its own limitations, and could perhaps be replaced with other methods of serialization. For instance, ByteSurge's usefulness is limited with regard to inheritance, as it does not provide the facilities to transmit fields that are defined within parent classes.

An exploratory evaluation of ByteSurge and other related approaches is conducted to determine and compare their performance. In this experiment, ByteSurge is used to serialize and de-serialize different numbers of data objects, under various configurations. In the first configuration, plain (uncompressed) streams are used, while in the second configuration compression is utilized using GZip before a data stream is produced. For comparison, a third configuration is used to convert the same objects into their JSON representation. An alternative fourth configuration is also used for comparison by utilizing Google's Protocol Buffers (Feng & Li 2013) to serialize the objects. For each of these configurations, the amount of time taken to serialize and de-serialize information is recorded in milliseconds, as well as the size of the produced stream in bytes. The number of objects converted is gradually increased by tenfold (i.e. 1, 10, 100, up to a million objects), with the purpose of determining the order of growth of each approach. During this experiment, several factors are kept in control, such as the computational power and load of the device used, and the programming language and environment used. The data object is a simple class representing basic information about a person, containing a String-based name, an integer-based age, and a floating-point height attribute. The values for these attributes are automatically generated and contain values within certain ranges and sizes. For each of the configurations and approaches, three runs are made with averages being calculated.

The results obtained from this experiment are mixed, but nevertheless useful for guiding the development of a methodology to handle serialization. To guide the interpretation of these results, the average exponential growth (AEG) of each approach is calculated using the formula shown in figure 9.1, where the sum of the difference of the logarithms of all data points is found and then divided by the number of data points (n). The AEG is a metric devised to measure the average growth between data points, allowing for a meaningful comparison between these approaches.

$$AEG = \sum [\frac{log(y_2 - y_1)}{log(x_2 - x_1)}] \div n \tag{9.1}$$

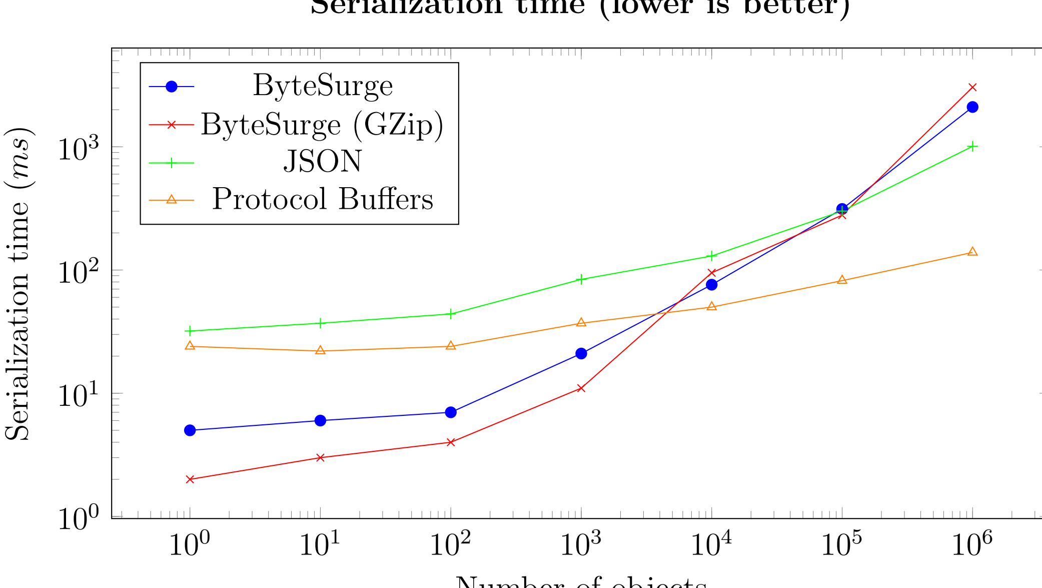


Figure 9.10: A comparison between ByteSurge (uncompressed and compressed) vs JSON serialization times.

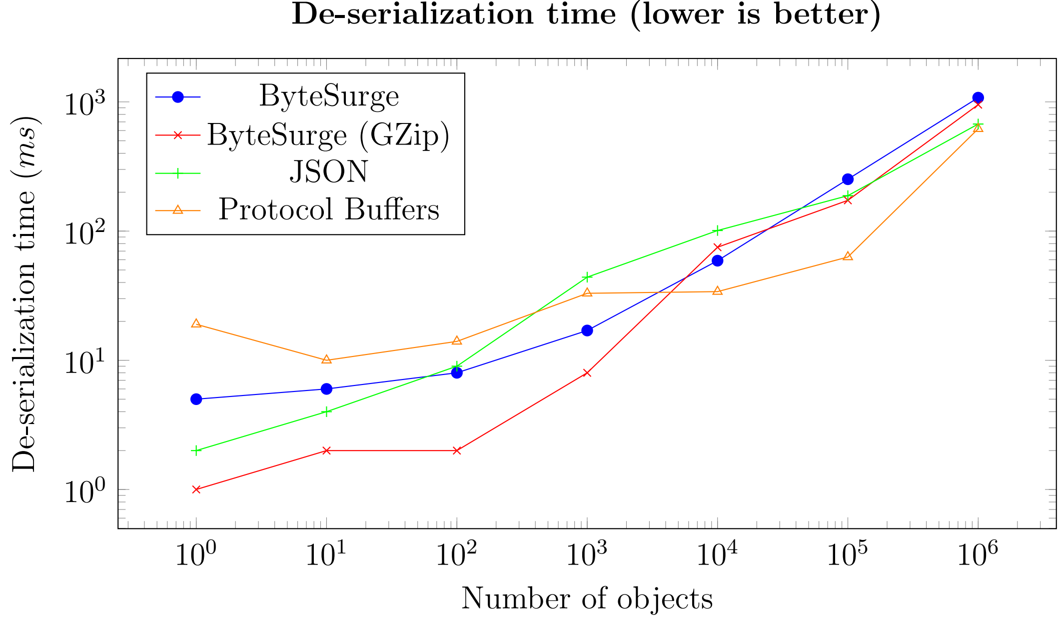


Figure 9.11: A comparison between ByteSurge (uncompressed and compressed) vs JSON deserialization times.

Data size (lower is better)

Size in bytes

Number of objects

ByteSurge
ByteSurge (GZip)
JSON
Protocol Buffers

Figure 9.12: A comparison between ByteSurge (uncompressed and compressed) vs JSON size.

When serializing data (i.e. converting from objects), the results – presented in figure 9.10 – show that the time taken to serialize objects grows exponentially in all approaches. Surprisingly, the compressed version of ByteSurge performs better than other approaches at 10,000 objects or below despite the computational overheads of compression. However, it has a much higher AEG factor than the other approaches which leads to a higher serialization time at larger scales. The uncompressed version of ByteSurge performs worse at low numbers of objects but performs slightly better than the compressed version at scale due to a lower AEG. JSON and PB are comparable in terms of growth, with JSON taking the most time to serialize low numbers of objects while having a slightly lower AEG than the ByteSurge approaches. PB takes slightly less time to serialize smaller numbers of objects than JSON but provides significantly better performance at large scales than all other approaches.

The results obtained from de-serialization show that PB also performs worse for low numbers of objects and best at larger scales. The JSON approach, which is claimed to provide good performance for de-serialization lives up to expectations as it performs very well in both small and large numbers of objects. Surprisingly, the compressed version of ByteSurge surpasses the uncompressed version in terms of performance again. It has the lowest de-serialization time at small numbers of objects overall and a slightly lower de-serialization time than the uncompressed version at a large scale.

Another aspect of this comparison is the total number of bytes generated by each serialization approach. In terms of data size, the compressed version of ByteSurge performs significantly better than other approaches, while JSON has the largest growth rate in data size. Meanwhile, PB and the uncompressed version of ByteSurge appear to have nearly identical growth in terms of the data sizes generated.

The experiment described explores four different approaches in terms of data serialization by comparing their performance in terms of time taken to serialize and de-serialize information, as well as the total size of the generated data. Surprisingly, the uncompressed version of the developed ByteSurge method appears to perform in-par with its compressed version while also offering significantly lower data sizes. It is suspected that the compressed version leverages concurrent execution more effectively than the uncompressed version despite the computational overheads. While performing relatively well based on its serialization and de-serialization times, the JSON approach leads to considerably larger data sizes. This, coupled with the fact that there are better approaches in terms of performance leads to its rejection for use in MMOG backends. At the same time, Protocol Buffers appears to have the best performance at a large scale, while it has a significantly lower performance at a small scale, and moderate potential in reducing data size compared to ByteSurge.

This experiment is limited in its scope. In a more thorough approach, various types of objects could be used to determine the performance of each approach. It would also be useful to compare these approaches in a different development environment, or perhaps while an application is deployed on a public cloud and utilizes real data. Nevertheless, this exploration makes it possible to navigate the possibilities and select an approach in terms of serialization. The first criterion to take into account is that MMOG backends will rarely have to compress huge numbers of objects at one go. Based on knowledge and experience gathered from the related works and the feasibility study, MMOG backends instead have to operate on relatively small numbers of objects concurrently, rather than batch-process them in large quantities. By these standards, the best approach out of the four explored would be the compressed version of ByteSurge, which provides better performance below 10,000 objects for both serialization and de-serialization, while also offering better size reduction. However, other factors must also be taken into account. While the ByteSurge approach acts as an abstraction over streams, it still has to overcome several limitations of this technology, such as its lack of support for inheritance.

Secondly, other approaches like JSON and PB offer considerably more variety in terms of tools and support, as they are already widely used in the software community. Another important aspect is the ability for developers to utilize a technology-independent serialization scheme. Although ByteSurge could be developed into various versions that can be utilized in multiple programming languages, this may take a long time to realize. While this approach performs better based on the results obtained, its usefulness in serverless cloud environments may also be impeded by the lack of support for concurrency in their runtimes. While ByteSurge can also work in sequential mode by utilizing a single thread of execution, its performance may be severely degraded and may not reflect the data acquired in these experiments. Lastly, the utilization of a serialization approach is also dependent on how data will be transmitted. While serialized data from other approaches like JSON and Protocol Buffers can be communicated over the network using various existing tools, ByteSurge does not have its corresponding toolkit.

Bibliography

Ably (2019), 'Ably realtime', https://www.ably.io/. Last accessed: 2019-12-10.

Agrawal, D., Das, S. & El Abbadi, A. (2011), Big data and cloud computing: current state and future opportunities, *in* 'Proceedings of the 14th International Conference on Extending Database Technology', ACM, pp. 530–533.

Alijani, G. S., Fulk, H. K., Omar, A. & Tulsi, R. (2014), 'Cloud computing effects on small business', *The Entrepreneurial Executive* **19**, 35.

Amazon (2022), 'Gamesparks'. Last accessed: 2022-06-30.
URL: *https://aws.amazon.com/gamesparks/*

Amazon Web Services (2019*a*), 'Amazon ec2 instance types', https://aws.amazon.com/ec2/instance-types/. Last accessed: 2019-12-10.

Amazon Web Services (2019*b*), 'Dynamodb - overview', https://aws.amazon.com/dy-namodb/pricing/provisioned. Last accessed: 2019-12-10.

Amazon Web Services (2019*c*), 'Dynamodbmapper - amazon dynamodb', https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Dynamo-DBMapper.html. Last accessed: 2019-12-10.

Apel, S. & Schau, V. (2016), Generic and distributed runtime environment for model-driven game development, *in* '4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)', IEEE, pp. 623–630.

Apple (2019), 'Arcade: Games that redefine games.', https://www.apple.com/lae/apple-arcade. Accessed: 2019-04-23.

Assiotis, M. & Tzanov, V. (2005), 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', number 4 *in* '2005'.

Attaran, M. & Woods, J. (2019), 'Cloud computing technology: improving small business performance using the internet', *Journal of Small Business & Entrepreneurship* **31**(6), 495–519.

Azman, H. & Farhana Dollsaid, N. (2018), 'Applying massively multiplayer online games (mmogs) in efl teaching', *Arab World English Journal (AWEJ) Volume* **9**.

Baker, J., Bond, C., Corbett, J. C., Furman, J., Khorlin, A., Larson, J., Leon, J.-M., Li, Y., Lloyd, A. & Yushprakh, V. (2011), Megastore: Providing scalable, highly available storage for interactive services, *in* 'Proceedings of the Conference on Innovative Data system Research (CIDR)', pp. 223–234.

Ballabio, M. & Loiacono, D. (2019), Heuristics for placing the spawn points in multiplayer first person shooters, *in* 'IEEE Conference on Games (CoG)', IEEE, pp. 1–8.

Barri, I., Roig, C. & Giné, F. (2016), 'Distributing game instances in a hybrid client-server/p2p system to support mmorpg playability', *Multimedia Tools and Applications* **75**(4), 2005–2029.

Bartle, R. A. (2009), From muds to mmorpgs: The history of virtual worlds, *in* 'International handbook of internet research', Springer, pp. 23–39.

Basiri, M. & Rasoolzadegan, A. (2016), 'Delay-aware resource provisioning for cost-efficient cloud gaming', *IEEE Transactions on Circuits and Systems for Video Technology* **28**(4), 972–983.

Baughman, N. E. & Levine, B. N. (2001), Cheat-proof playout for centralized and distributed online games, *in* 'Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)', Vol. 1, IEEE, pp. 104–113.

Becker, K. (2001), 'Teaching with games: the minesweeper and asteroids experience', *Journal of Computing Sciences in Colleges* **17**(2), 23–33.

Blackman, T. & Waldo, J. (2009), 'Scalable data storage in project darkstar'.

Bloch, J. (2008), *Effective java (the java series)*, Prentice Hall PTR.

Boillat, T. & Legner, C. (2014), Why do companies migrate towards cloud enterprise systems? a post-implementation perspective, *in* '2014 IEEE 16th conference on business informatics', Vol. 1, IEEE, pp. 102–109.

Boroń, M., Brzeziński, J. & Kobusińska, A. (2020), 'P2p matchmaking solution for online games', *Peer-to-peer networking and applications* **13**(1), 137–150.

Brewer, E. (2017), 'Spanner, truetime and the cap theorem'.

Burger, V., Pajo, J. F., Sanchez, O. R., Seufert, M., Schwartz, C., Wamser, F., Davoli, F. & Tran-Gia, P. (2016), 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', pp. 1–9.

Buttazzo, G., Lipari, G., Abeni, L. & Caccamo, M. (2005), *Soft Real-Time Systems*, Vol. 283, Springer.

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. (2018), 'A manifesto for future generation cloud computing: Research directions for the next decade', *ACM computing surveys (CSUR)* **51**(5), 1–38.

Carlucci, G., De Cicco, L. & Mascolo, S. (2015), Http over udp: an experimental investigation of quic, *in* 'Proceedings of the 30th Annual ACM Symposium on Applied Computing', pp. 609–614.

Carter, C. J., El Rhalibi, A. & Merabti, M. (2013), A novel scalable hybrid architecture for mmog, *in* 'IEEE International Conference on Multimedia and Expo Workshops (ICMEW)', IEEE, pp. 1–6.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A. & Gruber, R. E. (2008), 'Bigtable: A distributed storage system for structured data', *ACM Transactions on Computer Systems (TOCS)* **26**(2), 4.

Chu, H. S. (2008), 'Building a simple yet powerful mmo game architecture', *Verkkoarkkitehtuuri. Part* .

Chuang, W.-C., Sang, B., Yoo, S., Gu, R., Kulkarni, M. & Killian, C. (2013), Eventwave: Programming model and runtime support for tightly-coupled elastic cloud applications, *in* 'Proceedings of the 4th annual Symposium on Cloud Computing', ACM, p. 21.

Coleman, R., Roebke, S. & Grayson, L. (2005), 'Gedi: a game engine for teaching videogame design and programming', *Journal of Computing Sciences in Colleges* **21**(2), 72–82.

Deng, Y., Shen, S., Huang, Z., Iosup, A. & Lau, R. (2014), Dynamic resource management in cloud-based distributed virtual environments, *in* 'Proceedings of the 22nd ACM international conference on Multimedia', pp. 1209–1212.

Dhib, E., Boussetta, K., Zangar, N. & Tabbane, N. (2016), Modeling cloud gaming experience for massively multiplayer online games, *in* 'Consumer Communications & Networking Conference (CCNC)', IEEE, pp. 381–386.

Dhib, E., Boussetta, K., Zangar, N. & Tabbane, N. (2017), Cost-aware virtual machines placement problem under constraints over a distributed cloud infrastructure, *in* 'Sixth International Conference on Communications and Networking (ComNet)', IEEE, pp. 1–5.

Dhib, E., Zangar, N., Tabbane, N. & Boussetta, K. (2016), Resources allocation trade-off between cost and delay over a distributed cloud infrastructure, *in* '7th International Conference on Sciences of Electronics, Technologies of Information and Telecommunications (SETIT)', IEEE, pp. 486–490.

Diao, Z. (2017), Cloud-based Support for Massively Multiplayer Online Role-Playing Games, PhD thesis, Universitätsbibliothek.

Diao, Z., Zhao, P., Schallehn, E. & Mohammad, S. (2015), Achieving consistent storage for scalable mmorpg environments, *in* 'Proceedings of the 19th International Database Engineering & Applications Symposium', ACM, pp. 33–40.

Doglio, F. (2015), *Pro REST API Development with Node. js*, Apress.

Donkervliet, J., Cuijpers, J. & Iosup, A. (2021), Dyconits: Scaling minecraft-like services through dynamically managed inconsistency, *in* 'IEEE 41st International Conference on Distributed Computing Systems (ICDCS)', IEEE, pp. 126–137.

Donkervliet, J., Trivedi, A. & Iosup, A. (2020), Towards supporting millions of users in modifiable virtual environments by redesigning {Minecraft-Like} games as serverless systems, *in* '12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)'.

Eclipse (2019), 'Jetty - servlet engine and http server', `https://www.eclipse.org/jetty/`. Last accessed: 2019-12-10.

Eickhoff, J., Donkervliet, J. & Iosup, A. (2021), 'Meterstick: Benchmarking performance variability in cloud and self-hosted minecraft-like games extended technical report', *arXiv preprint arXiv:2112.06963* .

El Rhalibi, A. & Al-Jumeily, D. (2017), Dynamic area of interest management for massively multiplayer online games using opnet, *in* '10th International Conference on Developments in eSystems Engineering (DeSE)', IEEE, pp. 50–55.

Engine, P. (2022), 'Photon engine'. Last accessed: 2022-06-30.
URL: *https://www.photonengine.com/PUN*

Farlow, S. & Trahan, J. L. (2018), Periodic load balancing heuristics in massively multiplayer online games, *in* 'Proceedings of the 13th International Conference on the Foundations of Digital Games', ACM, p. 29.

Feng, J. & Li, J. (2013), Google protocol buffers research and application in online game, *in* 'IEEE conference anthology', IEEE, pp. 1–4.

Foundation, A. (2019), 'Apache tomcat', `http://tomcat.apache.org/`. Last accessed: 2019-12-10.

Fridh, M. & Sy, F. (2020), 'Settlement generation in minecraft'.

Gascon-Samson, J., Kienzle, J. & Kemme, B. (2015), 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', IEEE Press, p. 2.

GauthierDickey, C., Zappala, D. & Lo, V. (2004), Distributed architectures for massively multiplayer online games, *in* 'ACM NetGames Workshop', Citeseer.

Ghobaei-Arani, M., Khorsand, R. & Ramezanpour, M. (2019), 'An autonomous resource provisioning framework for massively multiplayer online games in cloud environment', *Journal of Network and Computer Applications* .

Google (2018), 'Overview of cloud game infrastructure', `https://cloud.google.com/solutions/gaming/cloud-game-infrastructure`. Accessed: 2019-03-05.

Google (2019), 'Stadia: Take game development further than you thought possible.', `https://stadia.dev/about`. Accessed: 2019-04-23.

Google (2021), 'Firebase for games — supercharge your games with firebase'.
URL: *https://firebase.google.com/games*

Google Cloud (2019), 'Datastore - nosql schemaless database', `https://cloud.google.com/datastore/`. Last accessed: 2019-12-10.

Hailpern, B. & Tarr, P. (2006), 'Model-driven development: The good, the bad, and the ugly', *IBM Systems Journal* **45**(3), 451–461.

Hosseini, M. (2017), 'A survey of bandwidth and latency enhancement approaches for mobile cloud game multicasting', *arXiv preprint arXiv:1707.00238* .

Huang, S., Chen, W., Zhang, L., Li, Z., Zhu, F., Ye, D., Chen, T. & Zhu, J. (2021), 'Tikick: Towards playing multi-agent football full games from single-agent demonstrations', *arXiv preprint arXiv:2110.04507* .

Iosup, A., Shen, S., Guo, Y., Hugtenburg, S., Donkervliet, J. & Prodan, R. (2014), Massivizing online games using cloud computing: A vision, *in* 'IEEE International Conference on Multimedia and Expo Workshops (ICMEW)', IEEE, pp. 1–4.

Jamin, S., Cronin, E. & Filstrup, B. (2003), Cheat-proofing dead reckoned multiplayer games, *in* 'Proc. of 2nd international conference on application and development of computer games, Hong Kong', Vol. 67, Citeseer.

Janzen, B. F. & Teather, R. J. (2014), Is 60 fps better than 30? the impact of frame rate and latency on moving target selection, *in* 'CHI'14 Extended Abstracts on Human Factors in Computing Systems', pp. 1477–1482.

Jardine, J. & Zappala, D. (2008), A hybrid architecture for massively multiplayer online games, *in* 'Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games', ACM, pp. 60–65.

Kasenides, N. & Paspallis, N. (2019), 'A systematic mapping study of mmog backend architectures', *Information* **10**(9), 264.

Kasenides, N. & Paspallis, N. (2020), Multiplayer game backends: A comparison of commodity cloud-based approaches, *in* 'European Conference on Service-Oriented and Cloud Computing', Springer, pp. 41–55.

Kasenides, N. & Paspallis, N. (2021), amazechallenge: An interactive multiplayer game for learning to code, *in* '29th International Conference on Information Systems Development', Association for Information Systems.

Kasenides, N. & Paspallis, N. (2022), 'Athlos: A framework for developing scalable mmog backends on commodity clouds', *Software* **1**(1), 107–145.

Kavalionak, H., Carlini, E., Ricci, L., Montresor, A. & Coppola, M. (2015), 'Integrating peer-to-peer and cloud computing for massively multiuser online games', *Peer-to-Peer Networking and Applications* **8**(2), 301–319.

Keele, S. et al. (2007), Guidelines for performing systematic literature reviews in software engineering, Technical report, Technical report, Ver. 2.3 EBSE Technical Report. EBSE.

Kienzle, J., Verbrugge, C., Kemme, B., Denault, A. & Hawker, M. (2009), Mammoth: a massively multiplayer game research framework, *in* 'Proceedings of the 4th International Conference on Foundations of Digital Games', pp. 308–315.

Kumar, R. & Kaur, G. (2011), 'Comparing complexity in accordance with object oriented metrics', *International Journal of Computer Applications* **15**(8), 42–45.

LeadingEdgeTech.co.uk (2019), 'How is cloud computing different from traditional it infrastructure?', `https://www.leadingedgetech.co.uk/it-services/it-consultancy-services/cloud-computing/how-is-cloud-computing-different-from-traditional-it-infrastructure/`. Accessed: 2019-03-17.

Lin, Y. & Shen, H. (2015*a*), Cloud fog: Towards high quality of experience in cloud gaming, *in* '44th International Conference on Parallel Processing', IEEE, pp. 500–509.

Lin, Y. & Shen, H. (2015*b*), Leveraging fog to extend cloud gaming for thin-client mmog with high quality of experience, *in* 'IEEE 35th International Conference on Distributed Computing Systems (ICDCS)', IEEE, pp. 734–735.

Lu, F., Parkin, S. & Morgan, G. (2006), Load balancing for massively multiplayer online games, *in* 'Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games', ACM, p. 1.

Lu, G. & Zeng, W. H. (2014), Cloud computing survey, *in* 'Applied Mechanics and Materials', Vol. 530, Trans Tech Publ, pp. 650–661.

Lundgren, J. (2021), 'Kubernetes for game development: Evaluation of the container-orchestration software'.

Matsumoto, K. & Okabe, Y. (2017), A collusion-resilient hybrid p2p framework for massively multiplayer online games, *in* 'IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)', Vol. 2, IEEE, pp. 342–347.

Meiländer, D. & Gorlatch, S. (2018), 'Modeling the scalability of real-time online interactive applications on clouds', *Future Generation Computer Systems* **86**, 1019–1031.

Microsoft Azure (2019), 'Introduction to azure cosmosdb', `https://docs.microsoft.com/en-us/azure/cosmos-db/introduction`. Last accessed: 2019-12-10.

Mildner, P., Triebel, T., Kopf, S. & Effelsberg, W. (2017), 'Scaling online games with netconnectors: a peer-to-peer overlay for fast-paced massively multiplayer online games', *Computers in Entertainment (CIE)* **15**(3), 3.

Minecraft (2022), 'Lessons for minecraft education', `https://education.minecraft.net/en-us/resources/explore-lessons`. Accessed: 2022-06-24.

Mishra, D., El Zarki, M., Erbad, A., Hsu, C.-H. & Venkatasubramanian, N. (2014), 'Clouds+ games: A multifaceted approach', *IEEE Internet Computing* **18**(3), 20–27.

Mordor Intelligence (2022), 'Gaming market - growth, trends, covid-19 impact, and forecasts (2022-2027)'.

Morgan, L. & Conboy, K. (2013), 'Key factors impacting cloud computing adoption', *Computer* **46**(10), 97–99.

Nae, V., Iosup, A. & Prodan, R. (2010), 'Dynamic resource provisioning in massively multiplayer online games', *IEEE Transactions on Parallel and Distributed Systems* **22**(3), 380–395.

Nae, V., Prodan, R. & Fahringer, T. (2010), Cost-efficient hosting and load balancing of massively multiplayer online games, *in* '11th IEEE/ACM International Conference on Grid Computing', IEEE, pp. 9–16.

Nae, V., Prodan, R., Fahringer, T. & Iosup, A. (2009), 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', IEEE Press, p. 9.

Nae, V., Prodan, R. & Iosup, A. (2011), 'Massively multiplayer online game hosting on cloud resources', *Cloud Computing: Principles and Paradigms* pp. 491–509.

Najaran, M. T. & Krasic, C. (2010), Scaling online games with adaptive interest management in the cloud, *in* 'Network and Systems Support for Games (NetGames), 2010 9th Annual Workshop on', IEEE, pp. 1–6.

Negrão, A. P., Veiga, L. & Ferreira, P. (2016), Task based load balancing for cloud aware massively multiplayer online games, *in* 'Network Computing and Applications (NCA), 2016 IEEE 15th International Symposium on', IEEE, pp. 48–51.

Objectify (2019), 'Objectify', `https://github.com/objectify/objectify`. Last accessed: 2019-12-10.

Paspallis, N., Kasenides, N. & Piki, A. (2022), A software architecture for developing distributed games that teach coding and algorithmic thinking, *in* 'IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)', IEEE.

Plumb, J., Kasera, S. & Stutsman, R. (2018*a*), Hybrid network clusters using common gameplay for massively multiplayer online games, pp. 1–10.

Plumb, J. N., Kasera, S. K. & Stutsman, R. (2018*b*), Hybrid network clusters using common gameplay for massively multiplayer online games, *in* 'Proceedings of the 13th International Conference on the Foundations of Digital Games', ACM, p. 2.

Plumb, J. N. & Stutsman, R. (2018), Exploiting google's edge network for massively multiplayer online games, *in* 'IEEE 2nd International Conference on Fog and Edge Computing (ICFEC)', IEEE, pp. 1–8.

Rezvani, M. H. & Khabiri, D. (2018), Gamers' behaviour and communication analysis in massively multiplayer online games: A survey, *in* '2nd national and 1st international digital games research conference: Trends, technologies, and applications (DGRC)', IEEE, pp. 61–69.

Rosenberg, J. (1997), Some misconceptions about lines of code, *in* 'Proceedings fourth international software metrics symposium', IEEE, pp. 137–142.

Satyanarayanan, M., Bahl, V., Caceres, R. & Davies, N. (2009), 'The case for vm-based cloudlets in mobile computing', *IEEE pervasive Computing* .

Schleier-Smith, J., Sreekanti, V., Khandelwal, A., Carreira, J., Yadwadkar, N. J., Popa, R. A., Gonzalez, J. E., Stoica, I. & Patterson, D. A. (2021), 'What serverless computing is and should become: The next phase of cloud computing', *Commun. ACM* **64**(5), 76–84.
URL: *https://doi.org/10.1145/3406011*

Schmidt, D. (2006), 'Guest editor's introduction: Model-driven engineering', *Computer* **39**(2), 25–31.

Schultheiss, D. (2007), Long-term motivations to play mmogs: A longitudinal study on motivations, experience and behavior., *in* 'DiGRA Conference', Citeseer.

Services, A. W. (2022), 'Gamelift'. Last accessed: 2022-06-30.
URL: *https://aws.amazon.com/gamelift/*

Shabani, I., Kovaçi, A. & Dika, A. (2014), Possibilities offered by google app engine for developing distributed applications using datastore, *in* 'Sixth International Conference on Computational Intelligence, Communication Systems and Networks', IEEE, pp. 113–118.

Shaikh, A., Sahu, S., Rosu, M.-C., Shea, M. & Saha, D. (2006), 'On demand platform for online games', *IBM Systems Journal* **45**(1), 7–19.

Shaikh, A., Sahu, S., Rosu, M., Shea, M. & Saha, D. (2004), Implementation of a service platform for online games, *in* 'Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games', ACM, pp. 106–110.

Shea, R., Liu, J., Ngai, E. C.-H. & Cui, Y. (2013), 'Cloud gaming: architecture and performance', *IEEE network* **27**(4), 16–21.

Shen, S., Hu, S.-Y., Iosup, A. & Epema, D. (2015), 'Area of simulation: Mechanism and architecture for multi-avatar virtual environments', *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* **12**(1), 1–24.

Shen, S., Iosup, A. & Epema, D. (2013), Massivizing multi-player online games on clouds, *in* '13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing', IEEE, pp. 152–155.

TechNavio (2022), 'Global massive multiplayer online (mmo) games market 2022-2026'.

Thakur, D., Shergill, K., Kaur, G., Kaur, S., Abrol, D., Singh, H. & Gill, A. (2021), 'Gaming addiction to massively multiplayer online games (mmogs) and quality of life.', *Indian Journal of Forensic Medicine & Toxicology* **15**(1).

Thompson, G. (2004), 'The amazing history of maze', *The DigiBarn's Maze War 30 Year Retrospective" The First First Person Shooter* .

Tsipis, A., Komianos, V. & Oikonomou, K. (2019), A cloud gaming architecture leveraging fog for dynamic load balancing in cluster-based mmos, *in* '4th South-East Europe Design Automation, Computer Engineering, Computer Networks and Social Media Conference (SEEDA-CECNSM)', IEEE, pp. 1–6.

Vähä, M. (2017), Applying microservice architecture pattern to a design of an MMORPG backend, PhD thesis, Tese de Doutorado. University of Oulu, 2017. Acessado: 30 Jul. 2021.[Online . . . .

Vogels, W. (2009), 'Eventually consistent', *Communications of the ACM* **52**(1), 40–44.

Wang, X., Zhao, H. & Zhu, J. (1993), 'Grpc: A communication cooperation mechanism in distributed systems', *ACM SIGOPS Operating Systems Review* **27**(3), 75–86.

Weng, C.-F. & Wang, K. (2012), Dynamic resource allocation for mmogs in cloud computing environments, *in* '8th International Wireless Communications and Mobile Computing Conference (IWCMC)', IEEE, pp. 142–146.

Yusen, L., Deng, Y., Cai, W. & Tang, X. (2016), Fairness-aware update schedules for improving consistency in multi-server distributed virtual environments, *in* 'Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques', ICST (Institute for Computer Sciences and Social-Informatics), pp. 1–8.

Zahariev, A. (2009), 'Google app engine', *Helsinki University of Technology* pp. 1–5.

Zhang, K., Kemme, B. & Denault, A. (2008), Persistence in massively multiplayer online games, *in* 'Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games', pp. 53–58.

Zhang, W., Chen, J., Zhang, Y. & Raychaudhuri, D. (2017), Towards efficient edge cloud augmentation for virtual reality mmogs, *in* 'Proceedings of the Second ACM/IEEE Symposium on Edge Computing', ACM, p. 8.