Evaluation of a backend for computer games using a cloud service

Malin Lundberg
Abstract

Evaluation of a backend for computer games using a cloud service

Malin Lundberg

Cloud services are popular for hosting applications because they offer simplicity and cost efficiency. There are a lot of different providers offering many different services, which can make it hard to find the one most suitable for you. To make informed decisions you need to evaluate different services. This project evaluates a cloud service, called Amazon Lambda, from one of the biggest cloud service providers. Amazon Lambda is a simple service which runs one function in response to an event. In this project it is evaluated on suitability, performance and cost. To evaluate suitability, a certain kind of applications, games, were selected. The game industry is innovative and put high requirements on performance. A few simple Lambda functions were implemented and integrated in a prototype game. Some calculations were made for determining the cost of hosting such a game on Amazon Lambda. A few tests were implemented and run in order to further evaluate the performance.
Contents

1 Introduction 6

2 Background 6
   2.1 JavaScript ......................................................... 6
   2.2 Node.js ............................................................. 7
   2.3 JSON ................................................................. 7
   2.4 Promises .............................................................. 8
   2.5 Gulp ..................................................................... 8
   2.6 NoSQL ................................................................. 9
   2.7 HTTP ................................................................. 9
   2.8 CORS ................................................................. 10
   2.9 Token based authentication ........................................ 10

3 Cloud computing 11
   3.1 Providers of public cloud services ................................. 12
   3.2 Amazon Web Services ............................................. 12
      3.2.1 Different services ............................................. 12
      3.2.2 Availability zones ............................................ 13
      3.2.3 Using Amazon Web Services ................................ 14
      3.2.4 Error handling ................................................ 15
      3.2.5 Cost of using AWS .......................................... 16

4 Problem formulation 18
   4.1 Game prototype .................................................... 19

5 System design 19
   5.1 Functions .......................................................... 19
   5.2 Outline of Lambda call ........................................... 20
   5.3 Database design ................................................... 21

6 Implementation 21
   6.1 Automatic upload and update of Lambda functions ............. 21
   6.2 DynamoDB ........................................................ 23
   6.3 Lambda functions ............................................... 24
   6.4 API Gateway ....................................................... 25
   6.5 Integration in game ............................................... 25

7 Methods of evaluation 26
   7.1 Latency tests ..................................................... 26
   7.2 Price estimations ................................................ 26
8 Results

8.1 Implementation .................................................. 28
  8.1.1 Automated process ..................................... 28
8.2 Latency tests ..................................................... 28
  8.2.1 Average latency .......................................... 28
  8.2.2 Changes over a day .................................... 28
  8.2.3 Impact of wait time between calls ............... 28
8.3 Price estimations ............................................... 29

9 Discussion ................................................................. 32

9.1 Automated process ............................................ 32
9.2 Interaction with AWS ......................................... 32
9.3 Testing of latency ............................................... 33
  9.3.1 Hiding latency ........................................... 34
9.4 Cost of using cloud services ................................. 34
9.5 Security ............................................................. 35

10 Conclusion ............................................................... 36

10.1 Future work ....................................................... 36
  10.1.1 Further development .................................. 37
  10.1.2 Automatic upload ...................................... 37
  10.1.3 Testing .................................................... 37

References ............................................................... 39
List of abbreviations

API Application Programming Interface
AWS Amazon Web Services
CLI Command Line Interface
CORS Cross-Origin Resource Sharing
CSS Cascading Style Sheets
HTML HyperText Markup Language
HTTP Hypertext Transfer Protocol
HTTPS Hypertext Transfer Protocol Secure
IaaS Infrastructure as a Service
JSON JavaScript Object Notation
NIST National Institute of Standards and Technology
NoSQL Not Only SQL
PaaS Platform as a Service
SaaS Software as a Service
SDK Software Development Kit
SSJS Server-Side JavaScript
1 Introduction

Many companies today use cloud services to host their applications. Cloud services have many advantages, such as cost, scalability and ease of use. However, there are a lot of services on the market from many different providers. It can be hard to find which services are suitable for which applications.

It would be impossible to evaluate all cloud service providers for all kinds of applications. Therefore, this report focuses on one service provider and one kind of application. The cloud service provider chosen for this project is Amazon Web Services, which is considered one of the first and biggest on the market [23]. The application type chosen is games, and particularly game backend. The game development industry is interesting because it is a rapidly growing industry with much innovation.

The aim of this project is to evaluate the performance of a cloud service as a backend for games. The cloud service evaluated is Amazon Lambda, which is a service for running code in response to other events. A couple of Lambda functions is implemented, integrated into a game and tested. Although Lambda has potential and it is easy to work with, it is not found to be suitable as a backend for games. The main reasons for this are that Lambda has too much latency and that it gets too expensive. The latency is found to be over 100 ms, which means it is not suitable for many games, such as real time multi-player games. The main reason Lambda is too expensive is because you pay for each call made and most games would do a lot of calls which adds up to huge sums.

2 Background

A Web Service is a programmatic interface that enables communication between applications using standard web techniques [11]. This section describes some tools and concepts that are used in this project, and that are also useful when talking about web development and cloud computing. The programming language used to implement this project is JavaScript and more specifically Node.js is used on the server side. Communication between the frontend and the backend of the game is done through HTTP and the data sent is in the format of JSON. To enable resource sharing between frontend and backend, CORS is activated. To make the calls between the frontend and the backend asynchronously, promises are used. Data on the backend is stored in a NoSQL database. To make the development process more efficient and to implement automatic uploading of Lambda functions, the task runner Gulp is used. To secure the communication, between frontend and backend, and to keep track of different players, token based authentication is used.

2.1 JavaScript

JavaScript is a general-purpose programming language which can be used in any application [7]. It is most commonly known for its use on the client side of web applications [49]. There, JavaScript is used together with HTML, for content, and CSS, for styling, to create interactive web pages. On the client side, JavaScript is the most widely
used language\[62\]. One example of what JavaScript can do on the client side of a web application is to change the text of an HTML element. An example of this can be seen in figure 1.

```
<script>
    document.getElementById("Id_of_element").innerHTML = "this is a new text";
</script>
```

Figure 1: An example of client side JavaScript code for changing the text of an HTML element on a web page.

Another place where JavaScript can be used is on the server side of web applications. This is called Server-side JavaScript, SSJS \[52\]. An advantage of using SSJS is that you use the same language as on the client side, which can be a convenience for developers. SSJS is also seeing a boost from the evolving of better JavaScript engines, which are used to execute JavaScript.

### 2.2 Node.js

One solution for SSJS is Node.js, which is a platform for running JavaScript on the server side \[25\]. The JavaScript engine used by Node.js is Chrome's V8 JavaScript engine \[8\]. The core of Node.js is the use of non-blocking operations, which makes I/O operations scale really well\[4\]. This is because a single thread handles multiple operations at the same time. In turn, this means Node.js works well in real-time web applications that want data to be exchanged freely between client and server. Basically, because you can handle more requests simultaneously, you can send more data between the client and the server. However, Node.js is not suitable for cpu-intensive operations since these are harder to run asynchronously. This project uses I/O operations and low-intensive cpu operations, which makes Node.js suitable.

An important advantage of Node.js is its extensive collection of open source libraries, which are handled through Node.js package system, npm \[8\]. Npm is built into Node.js, which makes it easy to install and version handle dependencies\[4\].

### 2.3 JSON

To exchange data between a server and a client you need to agree on a data format. A popular format is JavaScript Object Notation, JSON \[48\]. Two advantages of JSON is that it is compact and human readable, which can be seen in figure 2. It is also easy to parse, which makes it fast to use \[12\]. JSON is based on JavaScript, but it can be used in any language \[13\].
2.4 Promises

When you retrieve data on the web you often want to do it asynchronously, meaning there is no synchronization and work can be done in parallel. Because you do not know how long a call may take, you want a way to handle it when it is done. One way to do this is by using promises [22]. When you use promises for a call, it returns a promise. At this time the promise is unfulfilled meaning that the call has not returned yet. When the call later returns the promise will change status to either resolved or rejected, depending on the success of the call. By using promises that return instantly we can keep going with other work until the promise is fulfilled and then we can handle the response.

2.5 Gulp

To make development workflow more efficient, a task runner can be used. Task runners automate tasks, which makes developers work simpler and less error prone. There are several task runners available, such as Grunt, Gulp, Broccoli and Mimosa [10]. Grunt and Gulp are the most well documented of these. Choosing between them is a matter of taste. Grunt is older, and therefore have a bigger community and more plugins [21]. One advantage of Gulp is that it is considered easier to read, partly because it focuses on small tasks that does only one thing.

Gulp is a toolkit, that can be installed using npm, which helps automate time-consuming tasks [20]. When using Gulp, you define different tasks, in the form of functions. These task performs various work for you, like moving and compressing files. One example of a task can be seen in figure 3. One advantage of Gulp is that you can use all the libraries from npm to code your tasks.

Gulp tasks can be started manually from the command line or automatically. One way to start tasks automatically is by using Gulp watch, which tracks if certain specified
gulp.task('example_task', function() {
  gulp.src('path/src/*')
    .pipe(gulp.dest('path/dest'));
});

Figure 3: An example of a Gulp task which moves files from the folder `path/src` to the folder `path/dest`.

files are changed, and if so, runs a specified task, which for example recompiles your project [19]. An example of the syntax can be seen in figure 4. Gulp watch is itself used in a task which needs to be started from the command line.

gulp.task('watch', function() {
  gulp.watch('path/dir/*', ['update'])
});

Figure 4: An example of Gulp watch that runs the task `update` in response to files changing in the directory `path/dir`.

### 2.6 NoSQL

A popular kind of database is NoSQL, Not Only SQL. Unlike a relational database, a NoSQL database is not dependent on relationships between tables [5]. Benefits of using a NoSQL database, compared to a relational database, are that it is easier to design and it scales better. One way of storing data in a NoSQL database is by using key-value store, which stores values which are indexed by keys. Key-value store is used in this project. Some other ways of storing data are document store, wide column store and graph store [15].

In key-value store, values are retrieved by specifying the related key which makes read and write operations very fast. A special form of key-value store is document store, which given a key can also look up values within that field. Document store stores data as documents of, for example, JSON type and is more flexible than regular key-value store. Some popular NoSQL databases using key-value store are DynamoDB, Riak, Redis and Memcached [28].

### 2.7 HTTP

Hypertext Transfer Protocol, HTTP, is used by Web browsers to retrieve information from Web servers [27]. Hypertext Transfer Protocol Secure, HTTPS, is an extension of HTTP which encrypts the data sent in order to keep it secure.
2.8 CORS

By default, HTTP requests from browsers are only allowed to access content from the same domain as the first resource it retrieves origins from [10]. For example, if the browser requests a page from domain A, that page is not allowed to request content from domain B. Since it has become popular to load resources, such as CSS styling, from external domains, it is possible to get these so called cross-origin HTTP requests to work. It can be done by using the CORS, Cross-Origin Resource Sharing, mechanism which allows cross-origin HTTP requests through the use of HTTP headers [17]. When a server gets a request it can see from where it originated through the use of the HTTP Origin header. The server can then check its same-origin policy to see if that origin is allowed. A list of allowed origins will be sent back in the response from the server in the Access-Control-Allow-Origin header.

2.9 Token based authentication

An important feature in many applications is the ability to connect actions to certain users, so you know who does what. The main idea is to restrict access to some content so only special users can access this content. For example, only employees are allowed to access information about a company or only the owner is allowed to access their bank account. In the first case the restriction protects the company and in the other individual assets are protected.

To gain access in a restricted application you have to provide proof that you are allowed to do so. This is usually done by typing in and sending a username and password to the application. In case the application is web based the username and password are sent over HTTP or HTTPS. The application can then check these credentials, for example in a database, and grant access if they are valid. Then the user should be allowed to access the restricted parts, but since HTTP is stateless the application will not know that the user is allowed access unless the credentials are checked every time. A naive solution to this would be to send the username and password every time and check them against the database. Sending the password over and over again would increase the risk of someone stealing it.

To avoid sending passwords over HTTP again and again the idea of sessions was introduced. Basically, when a user logs in a new session is started and information about it is stored on the server [13]. However, this approach has had some problems due to the storing of the session information. Every login would create a new record and every following request will need to access this record to check the information. This means there will be problems with scaling if the amount of users increase drastically. Then we will have a lot of new records and a lot of accesses.

To get around these problems, token based authentication was introduced. Token based authentication stores no information on the server and is thus stateless. The general idea is that when the server has validated a username and password, a unique token is created and returned to the user. Then the user sends the token with every following request to prove that it has access rights. The token can be verified without
accessing any database and by looking only at the request itself. Due to its scalability, token based authentication is used by most of today’s big applications.

In games, such as in this project, you want to keep track of what is done by which player. At the same time you want to make sure that the actions are made by the player and not anyone else. To do this, some form of authentication is needed. The authentication method chosen for this project is token based authentication because it is a common and theoretically simple method.

3 Cloud computing

With cloud computing, users use remote resources, such as storage space and computing power, and pay only for what they use [23]. This, instead of investing in, and maintaining, their own resources. This have several benefits, of which two important ones are cost and flexibility. By using cloud computing, instead of their own hardware, users remove the cost for hardware and related resources such as physical space, electricity and humans for maintenance. In addition, the user can easily increase or decrease the amount of resources they use to fit the demand which makes it flexible as well as cost efficient.

Cloud computing has its origin in an old idea of timesharing and utilizing computing power in a fashion similar to other services, such as electricity [23]. There has been many attempts at timesharing over the years, of which the first one was launched by Professor John McCarthy in the late 1950s. Later, there were companies such as Sun and HP selling compute time on their computers in the early 2000s [10]. However, it was not until after Amazon launched their first cloud compute service in 2006 that cloud computing started reaching a broad market and gaining popularity. After this, the business models and technology for large scale data centers were refined enough to offer cost benefits for both providers and users.

According to the National Institute of Standards and Technology, NIST, anything classified as cloud computing has to fulfill five characteristics [9]. These characteristics are on-demand self-service, rapid elasticity, broad network access, resource pooling and measured service. The first of these characteristics, on-demand self-service, means the service should scale automatically to meet the needs specified by the users. As part of rapid elasticity, adding and removing resources should be rapid to accommodate for spikes in demand. Broad network access means the service is accessible through a common network such as the Internet, government networks or academic networks [17]. Resources also have to be shared between users, resulting in resource pooling, on some level. As part of measured service, cloud services should automatically monitor the resources used by each user and be able to report these to the user and the cloud service owner. This can be used as a basis for the user to pay only for what they actually use.

A cloud service can be deployed as a public service and be available to everyone, as a private service for a single organization, as a community service for a selected few organizations or as any combination of these [9]. NIST also defines three levels of cloud computing services. These are Infrastructure as a service, IaaS, Platform as a service, PaaS, and Software as a service, SaaS. In reality the lines between them are blurred, and
classifying a service as one or the other may not be easy.

**IaaS**
IaaS is the lowest level of cloud computing and provides hardware for users to run software on. The user does not control of the hardware and its architecture, but can decide what software to run on it, such as operation systems and applications. IaaS is realised as user controlled virtual machines running on various physical resources, such as processing, storage and network [14].

**PaaS**
PaaS is the middle level which provide IaaS plus a development environment, consisting of, for example, operation system and some applications. Users of PaaS upload applications to be run in a hard- and software environment of which they have no control, except possibly through configuration settings.

**SaaS**
SaaS is the highest level and provides an application running on top of PaaS. The user have no control over hardware and software, including the application itself, possibly apart from application configuration settings specific to the user. One example of a SaaS is web-based email, such as Google Mail.

### 3.1 Providers of public cloud services

There are many companies offering various public cloud computing services and some of the biggest of these are Amazon, Rackspace, GoGrid, Microsoft, Google and Salesforce [23]. At the beginning of 2016, Synergy Research Group made a summary of the cloud service market [26]. According to them, Amazon was the largest provider with a market share of 31% worldwide. Amazon was followed by Microsoft with 9% and IBM with 7%. Amazon offers Amazon Web Services, AWS, which will be discussed further in the next section.

### 3.2 Amazon Web Services

Amazon Web Services, AWS, is a provider of cloud computing services [29]. It was announced in 2002 [1], but did not start offering cloud services until 2006 when the cloud storage service Amazon S3, Simple Storage Service, was released [24, 2]. AWS is considered to be one of the first and biggest providers of cloud computing [23]. The origin of AWS was a desire from Amazon to remake their own infrastructure in order to make it scale better [3]. This in turn led to the idea of also renting out the infrastructure to others.

#### 3.2.1 Different services

Since its launch, AWS has expanded a lot and now provides a wide range of services. These services span many areas such as compute, storage and content delivery, databases
and networking. The main services used in this project are Amazon DynamoDB, Amazon Lambda and Amazon API Gateway. Lambda is used to run code, API Gateway is used when calling Lambda and DynamoDB is used to store data.

**Amazon DynamoDB**
Amazon DynamoDB is a cloud-based NoSQL database that can be used to store documents or do storage using the key-value store model. It was released on Amazon Web Services in 2012. DynamoDB automatically duplicates to different locations, which makes it less likely that data will be lost due to failure.

**Amazon Lambda**
On Amazon Lambda, code can be uploaded to run in response to events, for example a file being uploaded to another of service on AWS or a HTTP request. The underlying resources are automatically managed by AWS. The code, after being uploaded to Amazon Lambda, is called a Lambda function.

Apart from code, a Lambda function also consists of associated dependencies and configurations. Dependencies are, for example, usage of packages in Node.js and in which case you need to upload these together with your code. Some settings are name, which sets the name of your Lambda function, and role which determines the permission of your Lambda function. A role could for example be that your Lambda function is allowed to access a table in DynamoDB. For Lambda to know which function in which file to run a handler needs to be set. The handler is defined as `filename.functionname`. Memory size and timeout can also be set and can affect the cost of running your Lambda function.

Lambda functions can be implemented using JavaScript, Java or Python. When uploading your function to Lambda you get to choose a suitable runtime from the choices Java 8, Node.js 0.10, Node.js 4.3 and Python 2.7. You can upload your code to Lambda either by inline editing, as a zip file or as files on Amazon S3, which is a cloud storage service on AWS. Examples of how to do the uploading can be seen in the next section. If your code requires more than one file, for example if you in Node.js use other packages, you can not use inline editing. If you want to keep track of old version of your Lambdas, Amazon S3 is the best way to go since you can easily store older versions.

**Amazon API Gateway**
The idea of Amazon API Gateway is to let the user create APIs for calling other AWS services. For example, you can create an API that calls your Lambda functions. The API can add functionality like returning HTTP status codes and controlling access.

**3.2.2 Availability zones**
AWS have a concept of availability zones which makes the customers data safer from failure. The idea is that AWS hardware is spread out across the world in regions, with one or two regions in most continents. Within these regions there are multiple availability zones that are physically isolated from each other while maintaining low
latency connections between each other. By spreading services across multiple availability zones data is safe even in the event of failure, for example a fire, in one availability zone.

However this does not protect against all kinds of failure, as was shown with an incident in September 2015 [30]. At this time errors manifested in all of the US-EAST region. The problems started in DynamoDB, which had an increase in errors for read and write operations. According to AWS, the problem originated in DynamoDB’s internal metadata service [14]. The metadata service is responsible for partitioning tables over multiple servers. Due to a recent introduction of secondary indexes, these partitions had increased in size, making the response time for handling them longer. The occurrence of a small interruption in the network, caused many simultaneous requests. This made response times exceed timeout, which caused a bottleneck for the requests, which in turn caused more timeouts. AWS could eventually solve the problem by increasing the timeout and capacity for requests to the metadata service.

The errors in DynamoDB also affected other services in the US-EAST region using DynamoDB for internal storage. Users of these services, including DynamoDB, saw an increase in error and response time. Some users could not access their services at all while many more saw their performance fall to below acceptable levels.

In total, US-EAST region had problems for five hours and most of the customers affected had been totally unprepared for this. One way for customers to not get affected by these kind of problems is to duplicate their data across multiple regions [35]. Of course this is also more costly. In either case, incidents like this shows that even one of the biggest cloud service providers can have problems and that there are still risks in using cloud services.

3.2.3 Using Amazon Web Services

There are a multitude of ways to interact with AWS. Some of these are the AWS Management Console, AWS CLI (Command Line Interface) and AWS SDK (Software Development Kit) for JavaScript in Node.js.

AWS Management Console

With AWS Management Console you interact with AWS in your browser through a homepage [37]. This provides a graphical interface where it is easy to get an overview of your resources. Creating a new Lambda function is done by filling in a form, part of which can be seen in figure 5.

AWS CLI

AWS CLI (Command Line Interface) is a tool for interacting with AWS using the terminal or automated scripts [34]. For a user experienced in working with the terminal, this is a faster way to set up and manage your resources on AWS. In figure 6 is shown the command, with flags, for uploading a new Lambda function. The biggest difference, compared to using the AWS Management Console, is the need to know the account id associated with the execution role.
$ aws lambda create-function
  --function-name randomScore
  --runtime nodejs
  --role arn:aws:iam::012345678910:role/lambda_basic_execution
  --handler game.randomScore
  --zip-file fileb:////archive.zip

Figure 5: Part of the form for uploading Lambda functions in AWS Management Console

AWS SDK
AWS provides several SDKs for different languages, one of them is for JavaScript in Node.js. This library provides objects for AWS services and thus these can be interacted with \[38\]. AWS SDK for JavaScript in Node.js can be installed using npm. An example of code for uploading a new Lambda function using the AWS SDK for JavaScript can be seen in figure \[7\]. The SDK function takes the same arguments as AWS CLI, but here the arguments are passed in as a JSON objects instead of with flags.

3.2.4 Error handling
It is not trivial to make error handling between Lambda and API Gateway work \[18\]. When returning from a Lambda function there are three calls to choose from: success, fail and done. These calls takes in parameters, for success or failure, and returns them. Success takes a successful return, fail takes a failed return and done is a wrapper of the other two. These are called on the context object that is a parameter for all Lambda
```javascript
var params = {
    Code: {
        ZipFile: fs.readFileSync('path/archive.zip')
    },
    FunctionName: 'randomScore',
    Handler: 'game.randomScore',
    Role: 'arn:aws:iam::012345678910: role/lambda_basic_execution',
    Runtime: 'nodejs',
    Description: 'returns a random score'
};
lambda.createFunction(params, function(error, data) {
    if (error) {
        // an error occurred
        return console.error(error);
    }
    else {
        // successful response
    }
});
```

Figure 7: Code for uploading a new Lambda function using the AWS SDK for JavaScript functions. For example you might have a Lambda function that, when called, returns a random number. If the number generation is successful you want to return it using context.succeed(randomNumber) and if something goes wrong you want to use something like context.fail("Error, something went wrong"). You can also use done for each of these cases as context.done(null, randomNumber) and context.done("Error, something went wrong", null) respectively.

When Lambda returns an error it is returned as a string. So if you return an object as the error in your Lambda function, what you get back is a string representation of that object. When setting up API Gateway to call your Lambda function you have to tell it what to do in case of an error. As default API Gateway just passes along everything it gets from Lambda to the caller with an 200 (OK) HTTP status code. If you want to return anything other than an OK HTTP status code you need to set an integration response. In your integration response you can check for errors matching a regular expression and in that case return a suitable HTTP status code. This requires you to synchronize your integration response with your error messages so the regular expression will match. If you change one of them you have to change the other to match.

### 3.2.5 Cost of using AWS

The core idea of AWS pricing is that you only pay for what you use when it comes to, for example, storing space, computational power and data transfer [41]. The exact
parameters for pricing vary between the different services. There are no requirements of using a minimum amount or time; you can stop using the services at any time.

**Cost of using Amazon Lambda**

For running Lambda functions, you are charged based on the number of requests and the duration of them, in combination with the allocated amount of memory \([42]\). The first 1 million requests each month are free and each subsequent request costs \(0.0000002\). A request counts as each time an event or invocation starts an execution, including test invocations. For the duration you are charged for every GB-second used, which is the same as allocated memory multiplied by the execution time. The execution time is rounded up to the closest 100ms and you get 400000 GB-seconds for free each month. To calculate the cost for a month, you need the number of requests, allocated memory and average runtime rounded up to the closet 10th of a second. The price can then be calculated using equation 1. As an example, if you in a month call your Lambda function 10 million times, using 128 MB of memory and it runs for 400 ms on average, then your total price will be \(3.467\).

\[
\text{monthly cost} = \left( N - f_r \right) \times p_r + \left( \left( N \times R \times M \right) - f_c \right) \times p_c 
\]  

where
\[
\begin{align*}
N & = \text{number of requests} \\
f_r & = \text{number of free requests} = 1000000 \\
p_r & = \text{price/request} = 0.0000002 \\
R & = \text{average runtime, rounded up to the closest 10th of a second} \\
M & = \text{allocated memory, in GB} \\
f_c & = \text{free compute time} = 400000 \text{ GB-seconds} \\
p_c & = \text{compute price/GB-second}
\end{align*}
\]

**Cost of using Amazon API Gateway**

Prices for API Gateway are also dependent on which region you deploy your API in \([31]\). Apart from this you pay for the number of calls your API receives and outgoing data transfers. The price for API calls range from \(3.5\) to \(4.25\) for one million requests depending on region, where the price in Oregon is \(3.5\). The price for outgoing data transfers depends, apart from region, also on the amount of data, where you pay a smaller price per GB the more data you transfer. Prices for Oregon can be seen in table 1, where the first 10 TB costs \(0.09/\text{GB}\) and the next 40 TB costs \(0.085/\text{GB}\) and so on. As an example, if you deploy your API in Oregon and it is called 10000000 times, each returning 2kB of data, your total price will be \(37\). If you want better performance, with faster execution, you can pay for dedicated caching. The price for this ranges from \(0.02/\text{h}\) for 0.5 GB to \(3.8/\text{h}\) for 237GB in Oregon. During your first year of using API Gateway, the first one million calls to your API each month is free.
Table 1: Prices for outgoing data transfers using API Gateway and DynamoDB in Oregon

<table>
<thead>
<tr>
<th>GB Range</th>
<th>Price/GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-10</td>
<td>0.090</td>
</tr>
<tr>
<td>10-50</td>
<td>0.085</td>
</tr>
<tr>
<td>50-150</td>
<td>0.070</td>
</tr>
<tr>
<td>150-500</td>
<td>0.050</td>
</tr>
</tbody>
</table>

**Cost of using Amazon DynamoDB**

The price of using DynamoDB also depends on which region you choose to deploy in [33]. The price also depends on the storage capacity you use, the amount of data you transfer and your desired throughput. The first 25 GB of storage is free and the rest costs $0.25/GB per month. In addition to the data you upload, there is an additional overhead of 100b/item for indexing purposes. The price for incoming data transfer to DynamoDB is free and the price for outgoing data transfer is the same as for API Gateway, which can be found in table [1]. Throughput is measured in units and is separate for reads and writes. One unit can handle one operation/second if the items are smaller than 4kB for reads and 1kB for writes[40]. Larger items require additional units. When creating your table, you set the desired amount of read and write units. If the capacity is exceeded, additional requests will result in errors. Each month you can use 25 read and 25 write units for free across all your tables. Additional units are $0.0065/h for 10 write units or 50 read units. As an example, if you store 50GB of data, retrieves 10GB of data and your throughput is set to 30 read units and 30 write units, your monthly total will be $10. For additional fees you can also reserve capacity, stream data from DynamoDB and replicate your data across multiple regions.

**4 Problem formulation**

This project includes the implementation of several functions that will be uploaded to Amazon Lambda and will be called through Amazon API Gateway when certain events occurs in a game. It will also include storing of some data in Amazon DynamoDB.

The system will be tested for functionality and scalability. The goal is to identify advantages and disadvantages in comparison to other setups such as traditional and virtual servers.

The questions addressed in this project are

- Is it suitable to use Amazon Lambda in a backend for games?
- How can an architecture for this look?
- What are the technical limitations?
- How can the technical limitations be handled?
- What performance can be expected?
• What parts of the system are especially suited for Amazon Lambda?

4.1 Game prototype

Explore game is a simple game prototype developed by Magnus Lundstedt. In it you are a small black circle that can move around using a XBOX 360 controller for PC. The game consists of zones where each zone is 1920x1080 pixels. By moving outside this you enter another zone. The first zone contains blue squares, while the rest of the zones are empty. By moving over the squares, which are randomly spread out in the zone, you collect points that are visible in the top left corner. Next to the score you can see which zone you are currently in. A screenshot of the game with its original appearance can be seen in figure 8. The game is written in JavaScript and runs in the browser. It is compiled for the browser by Gulp. HTML and CSS is used in the displaying of the game, but all the underlying functionality is managed by JavaScript.

![Figure 8: Original appearance of game prototype created by Magnus Lundstedt](image)

5 System design

The system consists of three parts: Lambda functions, DynamoDB tables and an API created using API Gateway. Figure 9 shows how the different parts of the system communicate with each other.

5.1 Functions

The following functions were implemented and uploaded to Amazon Lambda.
Login Takes in username and password and checks these against the database. If valid, generates and returns a token unique for the user to be stored in the client and used in the following requests. Called when the user tries to login before the game starts.

Collision with object Takes in the id of an object and retrieves the score of that object from the database. The score is then added to the players score in the database and is also returned to the client. To be called when the player moves over an object.

Get objects in zone Takes in a zone (x,y) and retrieves all objects in that zone from the database and returns them to the client. To be used when the game is started and when the player enters a new zone.

Save current zone Takes in a zone (x,y) and sets this as the current zone for the player in the database. Is to be called when a player moves into a new zone.

Get players in zone Takes in a zone (x,y) and retrieves all players in that zone from the database and returns these to the client. To be used when the game is started and when the player enters a new zone.

5.2 Outline of Lambda call

There are a couple of steps that all of the Lambda functions need to perform when they are called. The general outline of these are as follows:

1. Assert that the right data is passed into the Lambda function

2. a) Check authentication (only Login)
   b) Check authorization (all except Login)

3. Run the function

4. Save result to database if needed

5. Return result
5.3 Database design

There are three tables needed in DynamoDB. These are for storing user authentication information, player information and object information. The fields of these tables can be seen in figure 10.

![Figure 10: The DynamoDB tables](image)

6 Implementation

All deployments to AWS was done in the region US West which is located in Oregon. In order to get familiar with AWS, a simple function was created that, given a name, returns "Hello name". This function was deployed to Amazon Lambda and an API was set up in Amazon API Gateway through which the Lambda function could be called. The function was deployed, and the setup was done, online using AWS Management Console. In order to see how Amazon DynamoDB works, a table was setup and three more Lambda functions were implemented and uploaded. The first Lambda functions retrieved all the data from the table. The other two respectively added and updated a record in the database.

6.1 Automatic upload and update of Lambda functions

Using AWS through the Management Console gives a good understanding of how the different services are used. However, when uploading several functions, and making settings for these, it becomes quite a repetitive task. The steps are the same for each function and doing these through a graphical interface is slow. It is also easy to miss a step in the setup. Because of this, a more efficient way of doing this was explored.

The goal here is to make the uploading and updating of Lambda functions automatic. AWS CLI has potential for this and its functionality was tested by setting up and configuring a simple Lambda function. For the most part the functionality of AWS CLI
matches well with that of AWS Management Console. The biggest difference is that AWS Management Console automatically give API Gateway permission to run the Lambda function when they are linked. In AWS CLI you explicitly had to give this permission in addition to linking them. Another important difference, compared to when working in AWS Management Console, is the id:s you have to keep track of when working with AWS CLI. Instead of using the name of for example a function you use the id of it when making settings. You also have to restate the id for every setting you do. In AWS Management Console you choose the function and then make all the settings for it. In AWS CLI you first state the setting you want to make and then you give the id. You get the id either by calling one of the functions that retrieves id:s or by noting the id that you get back when creating a function.

AWS CLI could have been used in a script for an automated process, but was discarded in favor of AWS SDK for JavaScript. The main reason for this is that AWS SDK for JavaScript can be used together with Gulp and especially Gulps watch function. The project already uses Gulp which make it easy to build on. AWS SDK for JavaScript have generally the same functions, taking the same arguments, as AWS CLI. The difference is that you use a JavaScript library and call JavaScript functions instead of using commands in the terminal. Because they have the same functions, you can use AWS CLI to quickly try out functions you want to use in AWS SDK for JavaScript.

The core idea of the automation process is that Gulp will watch for any changes of the files associated with the Lambda functions. If any change is made, Gulp will run three different tasks in sequence to upload the functions. The first task will identify each Lambda function and prepare a folder for it. The second task will zip each folder and the third will upload the function, or update it in case it already exists. By using a convention for naming the folders, no information needs to be passed between the tasks. The naming of the folders follows the pattern \textit{filename.functionname}, where functionname is the name of the function and filename is the name of the file where the function is found.

The first task checks a specified location and retrieves all JavaScript files and finds all of the exported functions in these. These are the Lambda functions to be uploaded or updated. For each of these a new folder is created, in a different location, to which the file for that function and all dependencies are copied. Dependencies not needed for a particular Lambda function are also copied to the folder of that function. The second task then makes a zip archive of each of the folders, keeping the name, and stores them to a new location.

The last task uploads, or updates, Lambda functions. This task iterates over each zip archive. By retrieving a list of the current Lambda functions and checking if the name of the zip archive is among them, the task knows if the functions already exists or not. In case it already exists, the code for it is replaced with the zip archive. In case it does not exist, a function is called to create it and its settings in API Gateway. Uploading a new function requires some individual configurations. Examples of these configurations includes the role Lambda is to assume to run the function, the method type, the name of the resource and a description of the function. The solution for this is a file where the configurations for each function is saved in a JSON object. These configurations can then be retrieved when a new function is to be created.
Most of the implementation of the tasks was pretty straightforward and much of the functionality could be accomplished with existing packages. The biggest problem was to get the tasks to return in a correct way. When tasks are run sequentially, they each need to make a return in order for the following task to know when to start. Incorrect returns from a task causes the following scheduled tasks to start before the completion of the previous. For example, if the task making zip archives returns incorrectly, the upload task might start before the zip archives are created and will encounter an error because there are no zip archives to upload.

The problem with returning from the tasks comes from the need to iterate over multiple files or folders within the task. In order to make a proper return, there is a need to wait for each iteration to be fully completed. This was accomplished by each iteration returning a promise. A promise says that an action is in progress and is expected to complete in the future. Thus chaining can be used after a promise, but the next function in the chain will not start executing until the promise is fulfilled. After iterating, the promises from all the iterations were returned so they could be waited for, before the following task started.

For the iteration, the best way found was to use a mapping function, which was the easiest to get working with promises. This is mostly due to the need for dividing up the code when using a mapping function. The code to be used inside the iteration had to be its own function which could then return a promise. This way the return of the mapping function will be an array of promises.

With the Gulp tasks, the Lambdas can be updated automatically in real time while the developer environment is active. For uploading new Lambdas it is decided that this is not necessary or desirable to be done while the developer environment was active. Usually you want to be completely finished with a function before you upload it to Lambda. In addition, the need for setting configuration for each new Lambda further suggested that new Lambdas should not be uploaded automatically. For this reason a flag is used that can be set to indicate that a new function should be uploaded. The flag is used when one wants to upload a new Lambda. When the flag is not present, which is the standard case, new Lambda functions will not be uploaded.

6.2 DynamoDB

Three tables were setup in Amazon DynamoDB; one for storing user authentication information, one for storing player information and one for storing objects.

User authentication information could have been stored in the table for player information, but was given its own table for safety and scalability reasons. By storing usernames and password in a separate table, this table only have to be accessed when a player logs in. It will not be accessed at any other time and accidentally leak passwords when other information is retrieved. If the usernames and passwords had been stored in the table with player information, one would have to be a lot more cautious with the accessing of that table. Therefore it felt more secure to to keep these in a separate table.

The other important point was scalability. By keeping usernames and passwords in a separate table, a user could more easily have multiple players in the player information
table. If these two tables were the same you would have to duplicate the password for each player the user have, which could be hard to maintain. For example if the password is changed, it would be easy to miss updating it in one of the players. This would also raise the question of which player object to check against when checking credentials for logging in.

6.3 Lambda functions

Six main Lambda functions were implemented: Login, get player information, collision, save current zone, get objects and get players. In addition, two extra functions were created: add object and add player. The extra functions were implemented to simplify the creation of new objects and players to be used when testing the other functions. The Lambda functions were implemented locally and uploaded to Amazon Lambda using the automated process described in section 6.1.

The functionality of a user logging in to the game was split into two Lambda functions. These are Login, which is responsible for authentication, and get player information, which retrieves all information about the player. This was done to minimize the work each function does, which makes them more flexible and easier to use independently in other systems.

If the authentication in the Login function is successful, a token is generated and returned. This token is then sent to, and verified by, all other Lambda functions before they do any other work. If a token is invalid, a fail is returned containing an error message starting with "Error". The tokens are generated and verified using the package hashAuthToken which is installed using npm.

All of the functions expect to get in some data and if it does not get in this data, it returns a fail containing a message starting with "Assert". All of the functions, except Login, expects to get in a token. Login instead expects to get in a username and a password. In addition to a token, the collision function wants to get in an object id. The save current zone, get objects and get players functions additionally wants to get in a zone.

All of the functions also access DynamoDB in some way. This is done by using the dynamo-db-doc package installed using npm. If an error occurs when accessing DynamoDB, a fail containing a message starting with "Exception" is returned. The Login function checks for a match of username and password. If one of them does not match, a fail is returned containing an error message starting with "Error". The get player information function retrieves the whole entry for a specified user. The collision function first retrieves the entry for an object with a specified id. If the specified id does not exist in the table, a fail is returned with an error message starting with "Assert". If the object is found, its score is extracted and then the score for a specified user is updated to the old score plus the score of the object. The save current zone function updates the current zone for a specified player. The get players and get objects functions retrieves the entries for all players and objects, respectively, within a specified zone.
6.4 API Gateway

Setting up an API using API Gateway was done manually, through AWS Management Console, and through the automated process described in section 6.1. The automated process created a resource containing a HTTP POST method for each function and connected this method to the corresponding Lambda function. Adding permissions for the API to run the Lambda functions was also done by the automated process. Enabling CORS for all methods was done manually, as was deploying the API. Generating and downloading the SDK was also done manually.

For each method there were three HTTP responses added, in addition to the default 200 OK. These are 400 Bad Request, 401 Unauthorized and 500 Internal Server Error. One of these were added by the automated process, but the others were thought of later and added manually. The added status codes were set to return JSON objects following the Error model. For each of them there was also an integration response added, mapping which responses from the Lambda function would invoke them to be returned. For fail messages from the Lambda function starting with "Assert", the HTTP status code 400 will be returned. For the fail messages starting with "Error", 401 will be returned and for the fail messages starting with "Exception", 500 will be returned.

6.5 Integration in game

The API created in API gateway was deployed and downloaded as a SDK. It was then integrated in the game so that the Lambda functions could be called from the game. The first Lambda function to be called is the Login function, which is called when the game is started. The username and password it gets passed in are stated in the file and are not given by the user. After the Login function returns a token, the Lambda function for getting player information is called.

The game had existing functionality for detecting when a player enters a new zone and the coordinates for this zone. A call to the Lambda function updating the current zone to this zone was added. The game also had existing functionality for detecting collisions with objects, where the corresponding Lambda function could be called.

To add objects to the game there was an already existing function which takes in the type, id and position of the object. This function was modified to also take in the color of the object, in order to make it possible to have differently colored objects. Another function was created for adding other players to the game. This function was identical to the function of adding objects, except that it painted out the objects as circles instead of squares.

Retrieving players and objects in a zone had to be done at two times, when the game starts and when the player enters a new zone. For this reason, a function was created which was then called at these times. The function called the Lambda functions to get players and to get objects, for a given zone. The function then called the function to add player, or object, for each of the retrieved items. To show objects in other zones than the staring zone, there was need to convert the position. In the database the object has a zone and a position. In the game the object has only a position, which means the
position in the game had to be calculated using both the zone and the position from the database.

7 Methods of evaluation

When testing the implemented Lambda functions there was a noticeable latency. This was considered one of the most important aspects of the performance and therefore tests were created to measure the latency. Another important aspect of using a cloud service is the cost. For this reason, a model for estimating the cost of playing a game, which uses cloud services, was set up.

7.1 Latency tests

For testing the latency, a Lambda function was implemented, which takes in a time and then immediately returns it. By passing in the current time and then comparing it with the time of the return, the latency of calling the Lambda function can be calculated. To get the time, JavaScript’s Date.now function was used, which returns the number of milliseconds since January 1, 1970 00:00:00 UTC [51]. By subtracting the time of the call from the time of the return, we get the latency in milliseconds.

Three different tests were performed using this Lambda function to check average latency, difference of latency over the course of a day and the effect of different wait times between calls have on latency. All of the tests calls the Lambda functions recursively to ensure that the calls are made sequentially and not interfering with each other. To check average latency, the function was called 10000 times in a row. To check the difference of latency over the course of a day, the function was called with a wait time of 10 seconds between the return of the previous call and the start of the next. This was done over the course of 24 hours. To check the effect on latency by the amount of time between calls, the function was called with exponentially growing time between the return of the previous call and the start of the next. The time between started at 1 millisecond and was doubled for each call. A total of 23 calls were made this way.

7.2 Price estimations

To calculate the monthly cost for hosting the game, we need to calculate the cost for each service used. we need to make estimations of the number of calls and the amount of data. The cost for using Lambda is the cost for calling the Lambda functions, which can be seen in equation 2, and the cost for running the Lambda functions, which can be seen in equation 3.

The cost for using API gateway is the cost for calling, which can be seen in equation 4, and the cost for outgoing data transfer, which can be seen in equation 5. The cost for using DynamoDB is the cost of outgoing data transfer, which can be seen in equation 6, the cost for writing to and reading from DynamoDB, which can be seen in equation 7 and 8 respectively, and the cost for storage, which can be seen in equation 9.
\[ \text{Lambda calling cost} = N \times p_r \]  
\[ \text{Lambda runtime cost} = N \times R \times M \times p_c \]  
\[ \text{API calling cost} = N \times p_r \]  
\[ \text{API data transfer cost} = N \times S \times p_t \]  
\[ \text{DynamoDB data transfer cost} = N \times S \times p_t \]  
\[ \text{DynamoDB write throughput cost} = p_{wu} \times U_w \times H \]  
\[ \text{DynamoDB read throughput cost} = p_{ru} \times U_r \times H \]  
\[ \text{DynamoDB storage cost} = p_s \times D \]  

where

- \( D \) = Amount of stored data (in GB)
- \( H \) = Number of hours
- \( M \) = Lambda allocated memory, in GB
- \( N \) = Number of requests
- \( p_a \) = API Gateway price/request = $0.0000035
- \( p_c \) = Lambda compute price/GB-second = $0.00001667
- \( p_r \) = Lambda price/request = $0.0000002
- \( p_s \) = DynamoDB storage price/GB = $0.25
- \( p_t \) = Outgoing data price/GB = $0.09
- \( p_{ru} \) = Price/h/read unit = $0.00013
- \( p_{wu} \) = Price/h/write unit = $0.00065
- \( R \) = Lambda average runtime, rounded up to the closest tenth of a second
- \( S \) = Size of object item (in GB)
- \( U_r \) = Number of read units
- \( U_w \) = Number of write units

The total monthly cost for hosting the game is the sum of using Amazon Lambda, Amazon API Gateway and Amazon DynamoDB. This can be calculated using equation (10):

\[ \text{Total monthly cost} = (2) + (3) + (4) + (5) + (6) + (7) + (8) + (9) \]  

8 Results

This section describes the results of the implementation, the latency tests for Lambda and cost estimations for running this project.
8.1 Implementation

The implementation did not result in a fully functional game, which was not the point either. Instead the game has enough functionality to test Lambda functions for some relevant features. The Lambda functions are called when they are supposed to during the gameplay. They were also easy to integrate into the game using the downloaded SDK, in the cases where the game already supported the corresponding feature.

8.1.1 Automated process

There is still a slight problem with uploading multiple new Lambdas at the same time, which resulted in errors on AWS part due to too many request. This can be worked around by making sure only one new Lambda function is uploaded at a time.

8.2 Latency tests

The testing of latency for Lambda shows that there can be a much variation. In the following sections are shown the results for the three different tests. In an undocumented test it was also seen that there was a difference between the latency on different networks. The tests for average latency and wait time between calls were performed during day time, by estimate some time between 10:00 and 15:00 UTC +2h.

8.2.1 Average latency

The average latency when calling the Lambda function 10000 times in a row was 488 milliseconds. The biggest latency was 1712 milliseconds and the smallest latency was 183 milliseconds. The spread of the latencies of all the calls can be seen in the histogram in figure [11]. As can be seen in the histogram, most of the calls had a latency of either around 200 or around 700 milliseconds. 45% of the calls had a latency of less than 250 milliseconds and 44% of them had a latency of 650-750 milliseconds. The rest were either in the range 250-650 milliseconds, with 3%, or above 700 milliseconds with 8%.

8.2.2 Changes over a day

The average latency per hour, over a 24 hour period, can be seen in figure [12]. The average number of calls per hour was 315 and the time zone from which the calls were made is UTC +2h or CEST, Central European Summer Time (Daylight Saving Time). As can be seen in the figure, the average latency is mostly between 1300 and 1400 ms. There is a little increase in latency between 14:00 and 14:59 where the average latency goes up to 1500 ms. The biggest increase is however between 15:00 and 16:59 where the average latency is around 1900 ms.

8.2.3 Impact of wait time between calls

The resulting latencies of calling the Lambda function with different amount of wait time since the last call can be seen in figure [13]. A majority of the calls with a wait of
under 8 second, had a latency of around 650 milliseconds or lower. For the calls with a wait time between 8 seconds and 1 minute, the latency was just above 1300 milliseconds. For the calls with a wait time between 2 and 35 minutes, the latency was around 1630 milliseconds. For the call with a wait time of one hour, the latency was 2600 milliseconds.

8.3 Price estimations

To estimate the amount of storage, D, we need the average size of player and object items and the amount of these. The size of player items and object items were calculated using the total size of each table divided by the number of items in each table. The average size of a player item is 102B and the average size of an object item, $S$, is 106B. The total number of object items depends on the number of zones we have and the number of objects in each zone. To simplify calculations we will use a fixed number of zones, 1000000, and a fixed number of objects in each zone, 10. This gives us the total cost shown in equation 11, where $P$ is the number of players.
Figure 12: Average latency per hour over one day.

\[ D = \frac{(1060000000 + 102 \times P)}{1073741824} \]  

The number of hours in a month, \( H \), is calculated using the assumption that there are 30.4368499 days in a month. With 24 hours each day we get a total of 730.4843976 hours. The allocated memory for Lambda, \( M \), is set to 128 MB for these calculations. The average runtime, \( R \), for the Lambda functions are 200 ms.

To estimate the total number of requests, \( N \), we need to multiply the number of players, the number of hours they play each day, the number of calls made each hour and the number of days in a month. To get the number of calls made by one player in one hour it was tested how long it took to collect 19 objects, which was 16 seconds. This gives us 4275 calls per hour. The total number of request can then be calculated using equation 12. \( G \) is the number of hours a player play each day on average.

\[ N = G \times 130117.5333 \times P \]  

A read, or write, unit in DynamoDB can handle 1 operation per second. That means that in the course of a month, a unit can handle a total of 60 * 60 *24 * 30.4368499 = 2629743.831 operations. Each call performs one read and one write operation. This means that the number of read and write units needed are the same and can be seen estimated using equation 13.

\[ U_r = U_n = \frac{N}{2629743.831} \]
This leaves us with two unknowns to consider; the number of players, N, and the number of hours they play each day, G. The costs for different number of players and different number of play hours per day can be seen in Table 2. As can be seen in the table, the cost is roughly linear when the number of players or the number of hours increase. The division of the total price between Lambda, API Gateway and DynamoDB barely varies with these changes. The cost for Lambda is around 14% of the total cost, the cost for API Gateway is around 81% and the cost for DynamoDB is around 5%.

<table>
<thead>
<tr>
<th>Number of players</th>
<th>Number of hours played per day</th>
<th>0.5</th>
<th>1</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0.53</td>
<td>0.81</td>
<td>1.38</td>
<td>2.51</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>3.08</td>
<td>5.91</td>
<td>11.57</td>
<td>22.89</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>28.56</td>
<td>56.86</td>
<td>113.48</td>
<td>226.71</td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td>283.33</td>
<td>556.41</td>
<td>1132.58</td>
<td>2264.91</td>
</tr>
<tr>
<td>10000</td>
<td></td>
<td>2831.08</td>
<td>5661.90</td>
<td>11323.56</td>
<td>22646.87</td>
</tr>
<tr>
<td>100000</td>
<td></td>
<td>28308.53</td>
<td>56616.81</td>
<td>113233.37</td>
<td>226466.50</td>
</tr>
<tr>
<td>1000000</td>
<td></td>
<td>283083.08</td>
<td>566165.90</td>
<td>1132331.53</td>
<td>2264662.78</td>
</tr>
<tr>
<td>10000000</td>
<td></td>
<td>2830828.63</td>
<td>5661656.77</td>
<td>11323313.05</td>
<td>22646625.62</td>
</tr>
</tbody>
</table>

Table 2: Total monthly cost for hosting the game ($)

Figure 13: Latency of calls with different time since the last call returned.
Amazon Lambda is easy to get started with and easy to integrate into your application using the downloadable SDK. It is however not suitable for all applications. Lambda has a noticeable latency which makes it unsuitable for applications that requires real time updates, like real time multiplayer games. In some games it is not important that there is a bit of latency, or this latency can be hidden. For example, in a turn based game, a bit of latency between the turns are not going to be noticeable to the players. In the case of the game in this project, latency for fetching objects in the zones can be hidden by fetching objects in adjacent zones before the player goes there.

Another aspect that makes Lambda unsuitable for many games is the price. Paying for every single request to you Lambda functions can get quite expensive for many games. It is also very hard to estimate the number of requests that will be made, which makes it hard to set a budget. A spike in the number of users or the amount of time they play in a month can drastically increase the bill. Therefore it might be better to pay for other services, like a server, where the price plan is more straightforward and not as sensitive to spikes.

The bottom line is that Amazon Lambda is unsuitable for most games. It is suitable for games where latency is irrelevant and where you will only have a small number of request to you Lambda functions. Lambda can still be used for smaller parts of a game and in other applications. The following sections look further into the latency and cost of Lambda, as well as the process of working with Lambda and other services.

9.1 Automated process

In this project, making the uploading of Lambda functions partly automatic, probably took more time than it saved in the end. This is mainly because the Lambda functions were few and not target for much change. However, for a larger project, with more functions subjected to more change, developing the automated process would probably have ended up saving time. Also, parts of the now implemented automatic uploading process can be used in other projects to save time there. Automation of the development process saves the most time if it is implemented and used properly from the start. Apart from saving time, an automated process will also lower the amount of possible user errors made in a manual process.

There are several improvements that can be made to the automated uploading process. For example, that it should only upload functions that was actually changed. Instead of, as now, upload all functions when one is changed. Also, it would be desirable for only the dependencies needed for the function to be uploaded, instead of all of the dependencies.

9.2 Interaction with AWS

There are different way of interacting with AWS, useful both when first setting up your services and later when changing them. Which way you choose to use can depend on many factors, such as how often you need to work with AWS and your technical level.
The most accessible way to interact with AWS, if you have no technical experience, is to use the AWS Management Console. It has a graphical interface which guides you through the processes.

AWS Management Console is good if you rarely need to interact with AWS or if you want to look at some new feature or service. If you on the other hand work with AWS often, AWS Management Console will slow you down since you need to go through all the steps manually. This is when other ways of interacting, such as AWS CLI and AWS SDK, are better. Both of them can be set up to automatically run many instructions, with little or no manual steps.

The difference between AWS CLI and AWS SDK is that AWS CLI is run in the terminal while AWS SDK is run in a programming environment. Other than that they have the same functionality. So which one you use is mostly a matter of personal preference and which one best fit into your development process. The bottom line is whether you want to write shell scripts or functions.

9.3 Testing of latency

The average latency, which can be seen in the histogram in figure 11, have two time spans that sticks out. Most of the calls seem to have a latency of either around 200 or 700 ms. It is not completely clear why this is. A reasonable explanation is that these represents two caching levels of Lambda, where half the calls need to access a lower cache when calling the Lambda function. Another explanation is that the calls are directed to different availability zones which happens to have different latencies. The calls can also have taken different routes over the internet, resulting in different latencies.

In figure 12 we can see the day average. There is some variations between different hours, but most noteworthy are the spikes between 14:00 and 17:00. It is unclear why the latency is so much higher at this time. Since the spikes are in the afternoon it might mean higher overall traffic on the network. For example, because people start coming home from work and go onto the internet while others are still on the internet at work. This does not feel very likely because there should then also be spikes at other times of the day. Another thing is that the Lambda function is deployed in Oregon which is in UTC -7h. This means a time zone 9 hours behind, placing the spikes between 05:00 and 08:00. It is unlikely that Oregon has a spike of network users at this hour.

The spikes between 14:00 and 17:00 in the day average graph might be a coincident. More test over several days would be needed to get a good average. The total number of calls also varied between the hours since the calls were made 10 seconds after the previous call was returned. Instead the calls should have been made with 10 seconds between the start of each call.

The calls for the average latency and the different times between calls were, by estimate, made between 10:00 and 15:00 UTC +2. This would mean they were made before the spikes in latency. If the spikes were not a coincidence, it is reasonable to think that the latencies for the other two test would have been higher if they had been performed at the time of the spikes. It would be good to perform the two tests at differing times of the day to get a better average.
In the test with different times between calls, we can see that there are roughly three
different latencies for different time spans. This is probably because there are different
levels of caches that the functions are moved to after certain amount of times. It would
be interesting with more tests to explore the caching policy of Amazon Lambda. For
example, if cache eviction is dependent on the recent number of calls/time unit.

Since different regions are physically closer to or further away from you, there is likely
an impact on latency depending on in which region you deploy your Lambda functions.
You likely want a region that is fairly close to your users. If your users are all over the
world you may even consider deploying in multiple regions. Further testing would need to
be made to determine exactly how much latency is affected by the region of deployment.

9.3.1 Hiding latency

Some latency always exists when you do things over the internet. The latency can be so
small that it is not noticeable or it can be bigger and severely affect the user experience.
Games, and other applications, that are run in real time are extra sensitive to latency
because they call for users immediate response to events. Especially when it comes to
multiplayer games, you want users to have low latency so they can immediately react to
each others actions. It is also desirable that they should have the same latency. This is a
good reason why you want as low latency as possible.

When it comes to fetching data from the server this can sometimes be done in advance.
For example, in this project, the objects in adjacent zones can be fetched before the player
gets there. This is an example of how you can hide latency. It can be done with much of
the static data. The dynamic data, for example other players movement, is much more
complicated. This data can not be prefetched and retrieving this data with high latency
will make it outdated and impossible to respond to in a reasonable way.

If you require dynamic data based on other players movement, it might be better to
create a direct connection between them. Instead of the data about the players going
through Lambda, or another web service, it would go straight between the players. This
way the latency would be minimal. Lambda can be used to initially keep track of the
players and when they get close enough to each other, tell them to set up a direct
connection between themselves.

9.4 Cost of using cloud services

An advantage of using cloud services is that you only pay for the resources you actually
use. However, it can be hard to estimate how much it is going to cost. To get an accurate
estimation, of using Lambda, DynamoDB and API Gateway, you need to know things
such as how much storage you need and how many calls will be made. This can be very
tricky if it is depending on your end users numbers and behavior. It can also be hard to
make comparisons to the price of other cloud services if they use a different price model.

The cost of hosting your game, or application, is only interesting in comparison to how
much money it will generate. For the most parts, you are probably not interested in
hosting an application that cost more than it generates in income. Since with AWS you
pay a monthly fee, you ideally want a monthly income that match or surpass your costs. One way to do this is to have a monthly player fee of slightly more than it costs to host the game. Another way to break even is to have an initial fee to buy the game. If we can make a good estimation of for how long a player will keep playing the game, we can set a fee that covers the costs of that time. There are of course a lot of other models to finance a game, like microtransactions or in game advertisement. Regardless of how you choose to finance your game, it is good to have a rough estimate of how much the game costs to host per player.

As can be seen in table 2, the monthly price scales roughly linearly with the number of players and the number of hours they play. There is an initial fee for storing all the objects, which is independent of the number of players. After that the price is based on the number of calls which scales with the number of players and their playtime. Because there is an initial storing fee the price per player becomes lower and lower the more players we have, because the initial fee gets spread out over more players. The fee per player is quite low, with a few dollars at the most.

The cost calculations are however based on many assumptions and estimations. There are many variables that were used as fixed values in the calculations, for example the number of objects and the memory allocated for the Lambda functions. A larger number of objects would mean a larger cost. The number of objects would also be of interest to change with the number of players, adding more objects as the player base increases. The calculations also did not take into account the tiered pricing and the free storage and calls each month.

Most of the cost per player per month is because of API Gateway. The least cost is because of DynamoDB, since there is not that much data that is being stored. With more complex games there would be need for saving more data which would increase the cost. API Gateway and Lambda uses the same amount of calls, but because calls to API Gateway are a lot more expensive they generate a larger cost. Subsequently, one way to save money would be to skip using API Gateway.

9.5 Security

An important part of most applications is security. In many applications it is desirable that the user can not do things that are harmful to the application or to other users. There can also be a need to know who the user is and to restrict access. In these cases you usually use login to authenticate the user and then you authorize based on their identity.

In the test application there is a need to keep track of each user, in order to have separate games for each user. Thus a login function was implemented, so only the user with the right credentials can resume his or her game. However, security was not a concern in this project, since its goal was to evaluate Lambda. Therefore, the login mechanism is not very secure and it is not following best practices. For example, the passwords are sent and stored in plain text. To be secure they should be, for example, salted and hashed, before being sent and stored. To follow best practice, the access token should have been sent in the header, instead of the body. This is to separate the data
about the request from the data in the request.

The login implemented in the game uses a hard coded username and password to login. To support multiple users there would be need to implement some kind of screen for the users to write in their unique credentials. When allowing users to give input to your application you need to validate that the data cannot do harmful things if the user writes in runnable code.

10 Conclusion

It is most possible to use Amazon Lambda as a backend for computer games. However, it is only suitable in some cases and there are several questions to consider. Amazon Lambda have a latency of over 100 ms when called, which means it is not suitable for situations like real time multiplayer games. It, however, works well when you do not care about, or can hide, latency.

With Lambda you pay for each request that is made to your functions. This means there might be other services which are cheaper when you reach a lot of requests. For example, it might be more cost effective to have a server constantly running than to pay for a lot of calls. Lambda is therefore most suitable when you have a low number of requests per user and time unit.

Whether you choose to use Lambda, or any other of Amazons services, it is good to automate the uploading and updating process. This will save a lot of time if it is done at the start of the project and there are many functions or changes of functions that will be made. Amazon provides good APIs for interacting with their services, both with AWS CLI and AWS SDK. Which of these two ways to choose is up to personal preferences and what other tools are used. The automated upload and update functions implemented in this project are not complete, but they are a proof of concept that it can be done and provides a good base for further development.

10.1 Future work

To better evaluate the performance of Amazon Lambda, further testing can be performed. In this project each test was only performed once, and under a short period of time. By repeating the test multiple times over a longer time period the data to analyze would be better. There are also improvements that can be made to the tests, for example with the test over one day, it would be better to start the calls every ten seconds, independent of previous call. In addition, there are further testing that would be of interest. Especially to look closer on caching policy and to look at the performance depending on region.

This work can be used as a base for using Amazon Lambda in other applications, both games and non-games. The automatic upload was made for general Lambda functions which means it can be used with any application. The only limit of the automatic upload is that it requires the file structure to be organized a certain way, although that would be possible to change. Using Lambda is not different when implementing games, versus non-games which means all processes described in this report can be used in any application.
10.1.1 Further development

Since the implemented functions were made to evaluate Lambda, and not run in production, they lack many features. These includes error handling, security and performance, as well as functionality. For example, the player and objects in a zone should only be fetched once, to save calls and avoid duplicated data.

The Login function have additional cases that would be of interest. These include the user logging in on multiple devices or in different browsers. It would not make sense to play the same game from two locations, so this would mean canceling the first session or not allowing a new one to be started.

To avoid cheating there would need to be checks on the player’s actions. For example, when the player changes zone, there should be a check that the new zone is adjacent to the previous one. Also, when the player collides with an object, there should be a check that the object is in the same zone as the player.

In order to get a more fluent player experience, the game should start in the players saved current zone. The game should also create new objects in an empty zone that the player enters. To avoid the wait when objects are loaded, the game should fetch objects from the zones adjacent to the players current zone.

To make an interesting multiplayer game you might want to remove objects from all other players when it is taken by one player. In this case you might also want to generate new objects.

10.1.2 Automatic upload

There is further work to be done and further aspects to investigate when it comes to automatic uploads, and updates, of Lambda functions. For example, it is not ideal to upload all of the Lambda functions when only one of them was changed. Furthermore, to keep the tasks small and flexible it would be a good idea to split the work into two different tasks, one for uploading new Lambda functions and one for updating existing Lambda functions. It would also be valuable to make comparisons with other task runners and with other ways of interacting with AWS, such as AWS CLI.

10.1.3 Testing

Since each latency performance test was only run once, it would be valuable to run them more times to see if the results are consistent. For the test that run over 24 hours, it would be interesting to see if the result is the same every day and if there is a difference between different weekdays. In a bigger perspective one can also look at differences over months and years. There is also the question if the result would be different if the time between the calls was something other than 10 seconds. To have the same number of calls for each time period one could call the function in intervals. Instead of waiting 10 second since the last call, you set up a call to made every 10 seconds regardless of how long each call takes.

It would be valuable to look more at how the time between calls effects the latency. There are further factors that could influence the latency, for example the cache eviction
policy. Here it might be interesting to look at how the latency is affected by the number of calls before, and the proximity to these. By collecting more data and analyzing it further, one will probably find further aspects that are of interest.
References

[1] Amazon.com. Amazon.com Launches Web Services; Developers Can Now Incorporate Amazon.com Content and Features into Their Own Web Sites; Extends "Welcome Mat" for Developers. [Online: 2016-05-10].


