# Cloud providers research

The purpose of this document is to take a look at the architecture for sem.6’s individual project and choose which parts can be replaced by a cloud solution that simplifies the operation while not compromising on any non-functional requirements.

# Analysing C2

The best way to identity what parts of the application can be replaced by a cloud service we need to take a look at the software’s C2 diagram.

A screenshot of a diagram

Description automatically generated  
  
The first opportunity that can be identified is an external bucket for storing media. This is the only sensible choice without having to manage a bespoke server myself just for storing media. An even worse option would be storing media in the database, which is terrible for performance as databases are not designed to store binaries in such a way.

The second opportunity that can be identified is moving the databases to a cloud provider. This way, they do not have to be managed and deployed by the development team and can be automatically scaled by the cloud provider. Managing your own database in a Kubernetes cluster is quite complicated as the team has to configure their own scaling strategy and make sure that data is saved in a reliable way.

The third opportunity is the cloud provider for deploying themselves, which will take the containers of the microservices, frontend and RabbitMQ and deploy them to a Kubernetes cluser. Setting up a separate machine and forwarding its ports so that they can be accessed to the internet is quite cumbersome. Moreover, this semester I do not have access to another computer besides my own.

The application gateway can also be handled by the deployment cloud provider, since there is no point in re-inventing the wheel – there are a number of robust gateway implementations that work out of the box.

# Choosing cloud providers

## AWS S3 Bucket

For the first opportunity where media needs to be stored on a cloud service, a good option is AWS’ S3 buckets. They are a really popular and robust choice when it comes to storing any type of files, as their interface, CLI and libraries are easy to use and understand. Moreover, AWS’ documentation is quite good and there a lot of threads on forums in case of commonly met errors.

AWS also provides IAM control over users who modify the bucket, which allows for the setup of a low-privileged account that can only modify the bucket used for this specific project.

Another good reason to go with AWS S3 would be the fact that their implementation is so popular that, even though not a standard, many other cloud providers support with compatibility layers. There are also ways to easily emulate a bucket locally with docker images, for example Minio.

A screenshot of a computer

Description automatically generated  
  
The last reason for going with AWS is the fact that they have a 1 year free trial, after which they charge for the amount of data that is in the bucket. In the case of this project, the production bucket would be mostly empty, so no additional charges will be acquired.

A close up of a text

Description automatically generated

## MongoDB Atlas

The second opportunity to implement a cloud provider can be achieved through using MongoDB Atlas. It is a cloud service that provides MongoDB clusters managed by the creators of MongoDB themselves.

The main reason for going with a cloud provider to manage the application’s databases is that there is no need for the developers to manage them themselves in Kubernetes. Moreover, scaling the databases manually is a tricky setup that can take an entire semester by itself to get right. In this case, MongoDB offers automatic horizontal and vertical scaling for their clusters.

A white background with black text

Description automatically generated  
Another reason to choose MongoDB Atlas is that it is very easy to configure on which server the database runs, therefore simplifying the GDPR requirements of the database being in a specific place in Europe.

A screenshot of a computer

Description automatically generated  
  
If the databases were managed by the development team, it would be hard to configure which database replica runs where, since Kubernetes takes advantage of nodes, which are in most cases either a standalone machine or a VM. These nodes can be located anywhere within a region, but a more specific location would be hard to pinpoint.

## Google cloud

For the third opportunity there are 3 popular options:

* AWS
* Azure
* Google cloud

The reasons to choose Google cloud are several.

Firstly, google cloud has supreme documentation and clear tutorials on how to get things done. Their instructions are easy-to-understand and follow while also being applicable to most scenarios. Competitors like Microsoft with Azure are notorious for their convoluted documentation and examples. Moreover, I have personally not had a great experience with Azure and this is still the case for the group project in this semester.

Secondly, Google cloud is being chosen over AWS due to the fact that deployment of this project will happen with Kubernetes. Google, being the developer of Kubernetes, offers a simple way to create a cluster and execute commands to deploy there.

Following Google’s documentation, it only takes about 20 minutes to deploy a containerized application to a cluster.

<https://cloud.google.com/kubernetes-engine/docs/deploy-app-cluster>

# Cost management & calculations

## AWS S3 Bucket

As Amazon notes, the user/organization pays for the number of items that are actually in the bucket and the amount of time these items have been there.

In order to get a more accurate estimate on how expensive maintenance on the S3 bucket will be, we can use Amazon’s AWS pricing calculator.

We are going to configure that the only service we use is S3 and it is located in London. Putting in estimates of 1TB of media stored monthly, along with 2 mil. monthly requests, where 1 piece of media stored is roughly 500KB. We can also estimate that the amount of returned data by the S3 bucket would be about 2TB a month, since a lot more people see the read photos rather than write.

A screenshot of a computer

Description automatically generated  
  
With all of these estimates in place, Amazon calculates that there is going to be a one-time fee of 11.38 USD, with a monthly estimate of 36.81 USD.

A screenshot of a computer

Description automatically generated  
  
One way to alleviate these costs is to apply heavy image compression, which is a common practice. Moreover, an upper limit for an image’s size can be enforced, which would be 500KB. Not having an upper image limit is unwise, since it is a vulnerability and an attacker could purposefully upload large images to generate costs.

The calculations seen above are already accounting for each image being 500KB. However, if we decide not to include compression, but still have an upper image limit of 2MB, the pricing becomes 4-fold, since images are 4 times the size. This causes the average amount of monthly storage to be 4TB, rather than 1TB.

A screenshot of a computer

Description automatically generated

## MongoDB Atlas

In order to calculate how much MongoDB Atlas would cost, we can use their calculator

when creating a new cluster.

To calculate the amount of data that will be generated, we can take an active user base of 100,000 users as an example and a total user base of 1 000 000. Each active user would create 1 listing per day, bid 3 times and post 5 comments. Following the application ERDs, that would amount to monthly dataset of:

* 100 000 listings
* 300 000 bids
* 500 000 comments

**User base size calculations**

* \_id: ObjectID 24 bytes
* email: string 100 bytes MAX
* username: string 50 bytes MAX
* password: string, 60 bytes when hashed
* firstName: string 50 bytes MAX
* lastName: string 50 bytes MAX
* address: string 50 bytes MAX
* phone: string 14 bytes MAX
* createdAt: date 30 bytes
* updatedAt: date 30 bytes

In total, each user in the application amounts to 458 bytes. Multiplying that by the 1 000 000 users, we get 458 000 000 bytes, or 458MB.

**Listing size calculations**

Each listing has its own ListingEntity and ListingFragment in the bid database. The ERDs can seen in their own evidence piece.

Listing entity:

* \_id: ObjectID 24 bytes
* propertyType: string 100 bytes MAX
* listingDescription: string 500 bytes MAX
* buildYear: number 2 bytes
* size: number 2 bytes
* startingPrice: number 4 bytes
* buyoutPrice: number 4 bytes
* location: string 50 bytes MAX
* startsOn: date 30 bytes
* endsOn: date 30 bytes
* creatorId: ObjectId 24 bytes
* images: string[] 2000 bytes max, 500 bytes average
* comments: ObjectID[] no limit, 600 bytes average
* createdAt: date 30 bytes
* updatedAt: date 30 bytes

ListingFragment

* \_id: ObjectID 24 bytes
* startingPrice: number 4 bytes
* buyoutPrice: number 4 bytes
* startsOn: date 30 bytes
* endsOn: date 30 bytes
* creatorId: ObjectId 24 bytes
* createdAt: date 30 bytes
* updatedAt: date 30 bytes

In total, each listing in the application amounts to 2106 bytes. Multiplying that by the 100 000 average listings created per month, we get 210 600 000 bytes, or 210MB.

**Bid size calculations**

BidEntity:

* \_id: ObjectId, 24 bytes
* amount: number 4 bytes
* listingId: ObjectId, 24 bytes
* creatorId: ObjectId, 24 bytes
* createdAt: date 30 bytes
* updatedAt: date 30 bytes

In total, each bid in the application amounts to 136 bytes. Multiplying that by the 300 000 average bids created per month, we get 40 800 000 bytes, or 40MB.

**Comment size calculations:**

* \_id: ObjectId, 24 bytes
* text: string 150 bytes MAX
* listingId: ObjectId, 24 bytes
* creatorId: ObjectId, 24 bytes
* createdAt: date 30 bytes
* updatedAt: date 30 bytes

In total, each comment in the application amounts to 282 bytes. Multiplying that by the 500 000 average bids created per month, we get 141 000 000 bytes, or 141MB.

**Total amount of data accumulated per month**

With 100 000 active users, each month a total of

* 210MB from listings
* 40MB of bids
* 141MB of comments
* **391MB TOTAL**

Will be accumulated each month. Multiplying that by 12, we get 4692MB or 4.6GB in data yearly.

We are going to choose a dedicated cluster, since the application is of enterprise scale. For the base cluster we can select a M60 cluster with 200GB of storage to ensure storage overhead, 64GB of RAM and 16 CPU cores. Moreover, we can enable automatic vertical scaling, which in need, can scale the cluster up to a M200 cluster with 256GB of RAM and 64 CPU cores.

We are also going to enable sharding, since this is the horizontal scaling that we are looking for. Setting the shards to 3 gives us a cost of 13.67USD per hour, which is quite pricey.

A screenshot of a computer

Description automatically generated

Converting this to a monthly cost by multiplying it with the formula (p/hr \* 24 \* 30) we get **9 842USD**, which quite a high price.

Unfortunately, there isn’t much that can be done to alleviate this price. The first options is to create optimized queries and requests that put as little stress on the database as possible. In the case of this project, that is already thought of and implemented as a part of the design process.

The second option is to fully manage a separate database on own hardware, which is not possible this semester.

The third and final option is to deploy the database to Kubernetes and still have to manage it myself. Scaling would need to be configured manually and there would still be costs accumulated towards Google cloud, rather than MongoDB.

## Google cloud

The price of the Kubernetes cluster can be calculated by referencing Google’s Kubernetes engine pricing.

Google charges a flat fee of 0.1USD/hour for a cluster beyond the free tier. Moreover, the prices of CPU, RAM and disk storage can be seen below.

A screenshot of a computer

Description automatically generated

With Kubernetes, the plan is to configure a HPA that keeps the number of pods per microservice between 1 and 10 depending on the load. Each pod would use 0.15 CPU cores and 256MB of RAM.

Since there are going to be 5 deployments (frontend, media, bid, listing, and auth microservice) that makes the total amount of maximum pods 50. The maximum amount of CPU they would use is 8 Cores and 13GB of RAM. Moreover, RabbitMQ can be configured to consume up to 3 CPU cores and 4GB of RAM, with 512GB of storage for saving messages to disk.

As for the minimum, that’s 5 pods using 0.15 CPU cores and 256MB RAM each, with RabbitMQ using 0.5 cores and 1GB RAM. RabbitMQ would also use almost no disk storage.

Executing the following calculations:

**Flat fee** 0.10USD/hour = 72USD/month

Minimum costs

**CPU cores**

1.5 \* 35.88USD = 53,82 USD

**RAM**

2.28 \* 3.956USD = 9.01 USD

**Storage**

10GB \* 0.044USD = 0.44 USD

**Total minimum/month**

135,27 USD/month

Maximum costs

**CPU cores**

11 \* 35.88USD = 394,68 USD

**RAM**

17 \* 3.956USD = 67.25 USD

**Storage**

512GB \* 0.044USD = 22.52 USD

**Total maximum/month**

556,45 USD/month

For the purposes of demonstration this semester, the minimum monthly costs will apply, as there will be no load testing on the production environment, as to not generate any costs.