Project - Phase 8 Report

Group 14
Tiago Carvalho fc51034
Diogo Lopes fc51058
Miguel Saldanha fc51072
João Roque fc51080
João Afonso fc51111

16/05/2021

1 Motivation

The idea for our project emerged when we were wondering how cool it would be to have an API that could help us decide which shows to watch next based on our personal taste. We figured that such a service could be developed with relative ease if users marked shows as viewed and/or liked.

Because users can watch more than just movies, for example animes, we wanted to use more than one dataset. Both animes and movies can have a lot in common not only with each other but also with books, so we also decided to use a book dataset. Using data from multiple datasets would give us a more realistic experience when it comes to cloud-native applications development, since these applications use data from so many sources.

With three datasets, we aimed to be able to effortlessly search through any of them. We wanted users to be able to mark movies, animes or books as seen or liked, and get suggestions of what to see next. Since the datasets have very similar categories among them, suggesting books mixed with movies and animes would be a possibility that we thought would add value to our application.

Our idea to implement the suggestion mechanism was to base these suggestions on the user's likes and views, which would indicate to us which categories the user prefers, and, therefore, allow us to suggest good movies, animes or books to the user.

We called our API "Seen", since users can see movies, animes and books and then get suggestions based on their profile, on what they have seen.

2 Dataset characterization

We searched for various datasets and the following one were what we ended up with. We then filtered what we felt it was important from the datasets

NOTE: We didn't include image's links because of the limited space per free MongoDB database, that would mean having a third database for movies, instead of two, and wouldn't be that useful for this API since it has no graphic user interface. Nevertheless, the posterior implementation wouldn't be hard at all taking in account the project architecture and implementation.

2.1 Dataset 1 — IMDB

This dataset provides a lot of information about movies and shows that can be seen in IMDB.

We downloaded the dataset (updated one year ago) from the Kaggle website.

From the whole dataset, these are the columns that were important to us:

Columns	Example
id	606e2683b3fff1da8a207ae9
name	The Arrival of a Train
category	[Action,Documentary,Short]
rating	7.4
type	short

Table 1: Movie example in our database

2.2 Dataset 2 — MyAnimeList

The second dataset, regarding the MyAnimeList website, was obtained from Kaggle.

This dataset not only has a lot of anime content, but also user information, but because we want to connect with the other datasets it doesn't make sense to use that data. We used the following columns:

Columns	Example
id	606e252aebddc73ebfb15507
name	Shakugan no Shana: Season II
category	[Action,Drama,Fantasy,Romance,School,Supernatural]
rating	7.72
imageUrl	https://myanimelist.cdn-dena.com/images/anime/10/18669.jpg

Table 2: Anime example in our database

2.3 Dataset 3 — GoodReads

At last, this dataset represents books from the GoodReads website, also downloaded from Kaggle.

The columns that are meaningful for us to be able to use this dataset together with the animes and movies datasets are the following:

Columns	Example
id	606e25ad5e927a606f534284
name	Of Mice and Men
description	The compelling story of two outsiders [...]
category	[Classics,Fiction,Academic,School,Literature,Historical]
rating	7.7
imageUrl	https://images.gr-assets.com/books/1511302904l/890.jpg

Table 3: Book example in our database

3 Use cases

We have 3 types of roles: an Admin, which is a logged-in user with special permissions, a normal User, which is a logged-in user, and a not logged-in user that we call Any.

Services	Role	Functionalities
Normal	Any	User Log in
		User Sign in
		Search for username
		See Book, Show and Movie Library
		Get top 10 Items with more likes
	User	Set Book/Show/Movie as seen/unseen
		Set Book/Show/Movie as liked/disliked
		Ask for suggestions to read and/or watch
		Delete account
	Admin	Add Book/Show/Movie to Library
		Remove Book/Show/Movie from Library
Spark	Any	See best Director in terms of rating*
		See which worker has the most connections

Table 4: Seen's Use cases
*at least with 10 movies and 10.000 reviews in each

4 API

Role	Path				get	post	put	del	description
User	/lib	/{page}			×				Returns a *page* from the database
	/suggest					×			List of suggestions to watch
Admin	/item					×			Creates an item to add to the database
	/item	/{type}	/{id}					×	Deletes item with specific *id* and *type*
Any	/item	/{type}	/{id}		×				Gets item with specific *id* and *type*
User	/item	/{type}	/{id}	/seen			×	×	Marks/Unmark item as seen
	/item	/{type}	/{id}	/like			×	×	Marks/Unmark item as liked
Any	/getTopTen	/{type}			×				Returns top ten most liked Items with *type*
	/user					×			Creates User
	/user	/login			×				Logs in
User	/user							×	Deletes logged-in User
	/user	/logout			×				Logs out
Any	/user	/search	/{username}		×				Searches User by username
	/bestDirector				×				Returns the Director with best rating with at least 10 movies with 10.000+ reviews
	/moreConnected				×				Returns the worker's imdb link that worked with the most (distinct) people

Table 5: Seen's API table

5 Architecture (application and technical)

5.1 Diagram

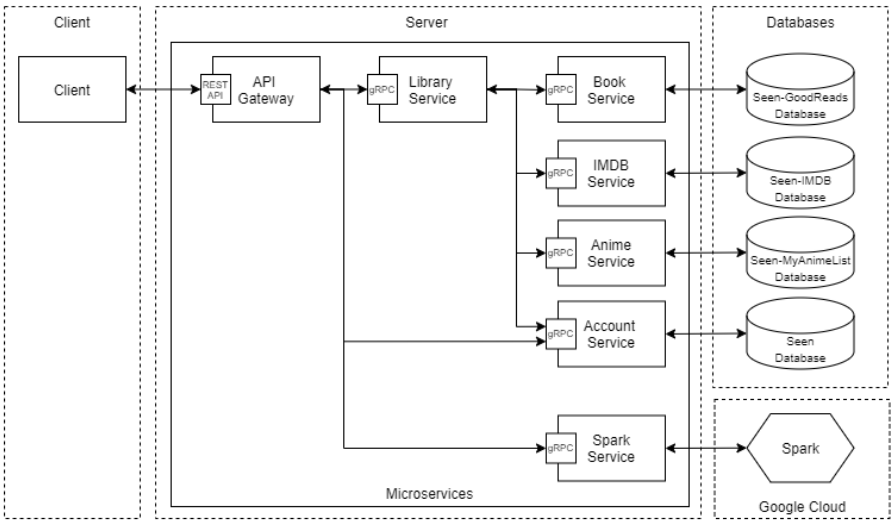

Figure 1: Project's architecture.

5.2 Client

The Client should be able to access our API on his browser:

`https://recommendations.sytes.net`

The Swagger provides a user interface to use and test our calls by adding "/ui" to the end of the url above.

5.3 Server

In total there are 7 different microservices working at the same time. Every single one of them runs on Google Cloud, inside the same cluster but in different dockers.

Our reasoning was to have an entrance microservice, which would redirect the request to the microservice responsible for that type of request, for example when sending a request for a page in our library, the API Gateway receives that request and sends it to the Library Service, which is responsible for asking for books, movies and animes to the Book, IMDB and Anime Services, respectively, and then put them together in just one response, which is then sent to the API Gateway, to be shown to the Client.

This API Gateway service also has the responsibility of transforming the REST requests from the Client to gRPC requests that are used internally, between Services. This service is also responsible for verifying the User authorization.

From the total of 7 microservices, 4 are responsible for the database connection, meaning that they are responsible for translating the request they receive into inserts, updates, removes or queries to the database. They are also responsible for translating responses from the database into responses that can be understood by the other microservices.

5.4 Spark

Later in the course of the project, we added the Spark microservice. This service receives the gRPC requests from the API Gateway service and then processes them, requesting a job in the Google Cloud storage, where we have a Cluster with the sole purpose of running this type of jobs.

5.5 Databases

Every database has a service that has the responsibility to access and manage it. While 3 of the databases are hosted by MongoDB, a NoSQL database, the last one is an SQL database hosted on an external server. This last database was initially hosted on Google Cloud, however we removed it from there, because it was costing us a lot of money.

For the Books', Movies' and Animes' databases, we used a NoSQL database since we might have had to change the format of our documents, meaning that if we had an SQL database we would need to drop the entire database and repopulate it again every time we decided to change the schema. MongoDB provides a very easy and intuitive python implementation to work with, which was also a factor to consider when selecting the type of databases that we would be using. Furthermore, when we started developing the project, we researched multiple databases so as to find the option that would be the most beneficial in the long term. We found that MongoDB was very suitable for cloud-based services due to its natural compatibility with the horizontal scaling and agility offered by cloud computing. Taking all this into account, it was clear to us that MongoDB would be a good choice.

However, for the Users' database we used an SQL one, because we already knew what we wanted from the User, and we knew we would use structured data for it.

6 Implementation

We actually made 2 different deployments, one that runs locally and other that runs in the cloud.

We created a local version of our application for several reasons: we were more familiar with the development of local-running applications, running the application on the cloud costs money, and we can debug the app better when it is running locally which is important during development. When developing a feature, first we would put everything running locally, by running a script, and once the feature was running without issues on our machines, we would adapt it to run on the cloud.

6.1 Server (Microservices)

6.1.1 Library

The business logic resides inside this microservice, providing a list of existing Items, allowing their management. Furthermore, allows the use of special requests with search algorithms.

This Service is also responsible for processing the new likes and views from the logged-in User, requesting an addition/change.

Other functions like Top 10 and the suggestion algorithm is responsibility of this service, making the necessary queries and results adaptation.

Suggestion Algorithm - Since we are a recommendation API, we also implemented an algorithm to suggest what the User should see next.

This function receives as input:

- The id of the User that wants to receive suggestions

- The types of content for which the User wants to receive suggestions
- The number of suggestions for each content of each type

This algorithm takes into account all the likes and views of the specific User to decide the 3 categories that the User might enjoy the most. It should be noted that the categories that are being used for this algorithm are not dependent in the type of content that was specified because, for example, a User might have only read books so far and now wants to find movies that have categories similar to the books he enjoys.

For this purpose the algorithm uses a scoring mechanism that attributes each category 1 point for every view and 3 points for every like. Afterwords, the 3 categories with the highest score are chosen. If the User has not viewed or liked any content so far, then the function will return a completely random sample for each type.

After getting the 3 categories, the function will then recommend a random sample (every sample has the number of elements that were determined by the User) for each category and for each specified type.

6.1.2 Docker Compose

For the microservices deployment we use a docker-compose file, which defines each microservice container details. In this file, we define a microservice's name, a build stating that we are using a Dockerfile and where the context of the microservice is (its folder, with its required content). We also define the image, hostname and container for the microservice, for which we use the same name as the microservice's. Finally, we assign a port to the microservice, which is crucial for it to behave properly.

We do the steps stated before for every single microservice inside the docker-compose, but that's not all we have to do. Since the API Gateway service sends requests to the Spark Connector, Account and Library services, we add them as environment variables, meaning that when we run the python file with the service implementation we can fetch the generated IP and then use it to connect to the services with gRPC channels. For the Library we add all the services responsible for the databases, which are the Book, Imdb, Anime and Account services.

6.1.3 Dockers

To create a docker image for each microservice to be containerized we used dockerfiles. These files are really simple, they basically have everything they need in the folder they are in, and just make a new folder called service, and copy the current folder inside the newly created one. After copying, they run the installation of the requirements, using pip and the "requirements.txt" file. Then, the same port used on the docker-compose file before, is exposed to be accessible

from outside this controller ambient/sandbox. Finally, the entry point is defined as the python file containing all the implementation of the microservice.

6.1.4 Protobuf Files

The protobuf files we created are the basis of our services communication between each other like we stated before.

This files can be seen like interfaces that the services need to comply, so the recipient of the message processes it.

We created an “utils” protobuf file, this file stores the interface of two messages, Success and Empty, and since they are repeatedly used we have them here to be ready to be imported. This file also stores an enumerated, with every Item type supported on this API: BOOK, SHOW and ANIME.

Then the protobuf files are very straight forward, we define a protobuf service with the name of the actual microservice, and inside that protobuf service the functions that this Service supports.

Those functions are also very simple, they consist in declaring a function that the Service has inside, with a message input and output, and the Service just has to comply with those restrictions, meaning that he receives a message of some type, and needs to “unpack” it and then “pack” it again to return as a response.

Because we import the "utils" file, we then can use the Type (BOOK, SHOW or ANIME) that we declared inside other messages declarations. And use the “repeated” as a crucial part of our messages, since we usually want to send lists back and forth with our Item’s content.

To build the protobuf files into importable python files with all the necessary functions for communication we use a bash script, which compiles them and copies each one to the Services that use them.

6.2 Databases

Because we are using the free subscription of MongoBD, we can only have up to 512 Mb in each Cluster we create, this means we cannot fit inside one cluster all the data we have to store.

To resolve this problem we created 4 different MongoDB clusters, one for each Item type, and an extra one for the movie database, since the movie data set takes roughly 1Gb we had to split it in 2 halves.

To upload our datasets, we used a script. The script connects to the database, then reads 100.000 lines from the dataset, adapts each one with the relevant data to upload and then sends those 100.000 items to the database.

Those adaptations are simple, using the csv package we can read both csv and tsv files row by row already split by columns. Then we transform the categories string into a list of strings, separating each category, facilitating queries. At last, we also strip the string from the white space at the beginning and at the

end.

About key duplicates we could have that problem using different databases, the generated key in one database could be the same as another one in another database, even if not likely it could happen. To resolve that problem we always use the type of the Item, with that type we choose which Service to send the request to, identifying the id and avoiding the "deletion" by omission of Items, because the first id found would be the one returned, omitting other results.

For the movie database we also prepend (insert at the beginning) a database identification ("1" for the first half of items and "2" for the second one).

For the user database we used another strategy and started an SQL database using SQLAlchemy to connect, using a secure channel with TLS.

To maintain the same structure inside and outside the database we also created a python file named "models", which has the correspondent classes to the tables inside the database, this way we can use those classes to fetch results already formatted.

6.3 Spark

We have 2 use cases: one that uses a csv (generated with a job beforehand) to create a graph using Spark GraphX to calculate the degree of each vertex, checking which has the highest degree to check which person has worked with the biggest quantity of people; and another one that sends a direct query to calculate which movie director has the highest average rating in all their works.

6.4 Technical

6.4.1 Databases

Since the MongoDB already provides an API with secure channels we didn't need to make any changes to the connection, the transmitted data is secured, and we tested it by running Wireshark to sniff the packets and observing they are encrypted.

To use a secure connection between the Service and the users' database, we initialize a connection with certificates (SSL server certificate authority, SSL server certificate and an SSL client key) using the TLS cryptographic protocol designed to provide secure communications.

6.4.2 Spark

Having our jobs already on the cloud, the Spark Service (Spark Connector) has the responsibility of asking the Dataproc cluster for the result of the job that was requested, that will run it and send back the result.

6.4.3 Authentication with Auth0

At first, we had our own authentication method, which we implemented using basic authentication and JWT tokens.

The user could register an account, using their email, which would then be confirmed with an email message. Then, they could choose their username and password, finishing the registration process.

The passwords were hashed using SHA512_Crypt. Later, we changed the authentication to an external service, auth0. To configure it, we defined an API on auth0 and configured an app.

We used implicit flow, with client credential flow enabled just for testing. Our auth0 also uses roles, admin and normal user, with a set of permissions for each. The roles are given with a rule we set, making every registration automatically a normal user, with admins being configured manually.

The difference between admins and normal users is that admins can add and delete items from the database. On Swagger, the user can use the authorize button to request the token to the auth0 service, using our defined API as audience and supplying the scopes, i.e., the permissions they require.

After this, the user is redirected to the auth0 page to conclude with their authentication. With this done, Swagger gets the token needed and adds it to all its request headers.

The user can also do this manually, but they would have to add the token manually after logging in. This can also be done using a client app using our API, similarly to what Swagger does: the user logs in on auth0 and the client app gets the token, which it then adds to its own headers. The token is verified using the public key from auth0.

6.4.4 Secure channels with istio

After the cluster is created (using create_cluster.sh), we then install Istio (using install_istio.sh). When installing Istio, we supply a valid certificate which is used in its ingress, allowing for secure communications through HTTPS.

Istio also encrypts all the connections between the ingress and microservices.

6.4.5 CI/CD pipeline

To implement the CI/CD pipeline, we created a trigger that when a commit is done on the deploy branch on git, the cloud build starts by running the yaml file.

First, it builds all the microservices, pushes them to the container registry, starts the Docker and runs the microservices.

After they are running, we start a test network using Docker, which is then tested (integration and unit tests), to assure that everything is alright and can be deployed on Kubernetes. If all the tests pass, the system is then deployed.

6.4.6 Deployment

The deployment is done using its yaml file, deployment.yaml, where we have defined all the services of each microservice, the auto-scaling properties and an Istio ingress, which uses the certificate we supplied.

7 Evaluation and validation

To perform an evaluation on the system by stress testing, we used three different technologies: Prometheus, Grafana and Locust.

Prometheus is a monitoring system that retrieves data directly from the cluster as it is executing, while Grafana displays and amalgamates the data in a way that can be analyzed by the human eye (plots, etc.).

To perform the stress tests on itself, we used Locust, a stress testing tool.

All these tools were configured automatically with scripts to further automate the deployment of the system (see locust.sh and add_plugins.sh).

To test our Spark implementation we used the interface inside the Google Cloud named Dataproc.

We also tested individual components of the system so as to find possible bugs in the implemention.

7.1 Evaluation

We ran Locust, the tool to perform the stress testing, in another cluster to avoid affecting the results.

It was tested with up to 800 users, which at that point it was getting close to 100% CPU usage, according to Grafana. At the time, we were using weak machines (12 CPUs) as Google Cloud's free trial did not allow any higher than that. We later got access to better machines and so we tested with up to 6800 users, which was still limited by quotas since we couldn't spawn more users to do so.

Every aspect of the system was being tested: getting entries, putting new entries, get pages from the database, get a recommendation (and even the login and logout which was later changed and so there is no need for load testing as it is handled by an exterior service, auth0). We did stress testing with 100 users, 500 users, 800 users, and later, when we had access to better machines, up to 6800 users.

For the Spark job we went to Google Cloud > Dataproc > Jobs, and observe the time taken running, and got an average of around 2 minutes and 10 seconds for the first one and 7 minutes and 40 seconds for the second.

We developed unit tests to verify if the behavior of the following microservices: "anime", "book", "imdb" and "account". We wanted to confirm that these microservices were acting in accordance with the system specifications and that the respective databases were returning the desired values.

We tested each functions and covered situations in which the objects in use did/didn't exist. It was also verified functions' behavior when receiving invalid input (for example, searching for an object with an id that doesn't exist).

Most of the tested operations present the expected behavior, but there were some cases in which the functions didn't properly deal with exceptions (these bugs were fixed).

After testing the modules at a lower level, we proceeded to also develop tests for the "library" and "api gateway" microservices. As for the "library" microservice, we developed tests in Python so as to verify if the interactions with the lower-level components were working properly. We decided to take a different approach to test the "api gateway" microservice: we used curl to test the REST API.

7.2 Validation

With 800 users the system reached around 75% CPU usage and 2000 to 2500%

system load, which we assume to be a somewhat comfortable limit for the resources we have, it can go higher, but it could probably start running into performance problems after a certain point (images in appendix 11).

The latency remained good nonetheless, around 2 to 4ms. Disk usage was always very low as most of the data was not stored in the cloud but rather on an exterior database (MongoDB and MySQL). There was a high usage of memory though, probably to store all the requests (around 14gb).

When we later tested with 6800 users, we got to around 600% CPU usage, which is equivalent to 6 CPUs running at their full potential. System load had spikes of up to 400% and the memory usage was around 35%. Since most of the load was concentrated on the api-gateway, we increased it's scaling capabilities immensely, with it getting to 100 pods, give or take. As mentioned before, our system could be tested with a bigger load, but unfortunately we were limited by the quotas.

8 Cost analysis

Services	**Cost**
Anthos	0,92€
Cloud Build	0€
Cloud SQL	108,48€
Compute Storage	7,55€
Compute Engine	155,00€
Kubernetes Engine	0€
Source Repository	0€
Stackdrive Logging	0€
All	271,66€

Table 6: Total cost table

In total, until this moment, we spent around 272€ on this project, all covered by coupons and discounts.

Most of our expenditure was on Cloud SQL (40%) and Compute Engine (57%). For the Cloud SQL we weren't expecting being this expensive, but since we were using a CPU with 8 cores our expense is 8 times bigger than it could've been.

After discovering the amount of money used on the Cloud SQL, we shut it down and repopulated an independent SQL database, hosted by a Raspberry Pi (with all the security measures previous implemented).

The Compute Engine can be easily explain since it's the expenses of running our microservices, because they were between 6 and 7, since the beginning of the development, and used 2 or 4 vCPUs, that were switched back and forth because the free trial billing wouldn't let the use of more than 12 vCPUs, and since we also needed them elsewhere we couldn't use more than 2 here.

9 Discussion

With the project up and running it can endure a lot of traffic, both in user capabilities and data transfers.

It has shown how dynamic it can be being able to support either 800 or 2000 or even more users at the same time.

With in this mind we can see how well our architecture supports our API, even though its costs have been higher than expected.

About the costs it's expected of us to minimize them with a few alterations, like we stated before, at section8, running the Cloud SQL with fewer cores, as the extra one weren't needed, or like we did, ditching it all together and store the data on a private server. Of course that our dataset is not the biggest, but at this scale it seems appropriate.

But thinking more broadly, if we were trying to make an actual API to use on an application of some sort, these expenses would be really hard to justify, because it would be really hard to get that money back by monetizing that application, and even if we decreased our cost, like we would intend, our application wouldn't be profitable at all, or at least that's what we would expect with our little experience on this project.

Overall about security we did the basis to protect the users' data, and also the databases' data, by encrypting communication channels with TLS (provided by istio) and storing login data with Auth0.

Other thing we could do to improve the Security of our API would be encrypting User's and Item's data stored inside the databases, preventing attacks on those Services.

10 Conclusions

At the end, we can see that we made a functional API using, mainly, Google Cloud, integrated with MongoDB and an external SQL database.

Not only that but also a useful API that anyone could use, either a User or even an application as an intermediary.

But the costs that we experienced are really high for what we are doing, if we didn't have our coupons and discounts it wouldn't be justified the amount of money spent.

So we can say that the intelligent thing would be, firstly, implement the API on a Linux machine, for example, and after knowing how to monetize it start the integration with Google Cloud, or other cloud Service, to endure more users and data without compromises.

Concluding, this API is fully implemented with all capabilities committed on the Use Cases, on section 3.

10.1 Contributions

10.1.1 51034 - Tiago Carvalho

- Implemented API functions
- Error correction and debugging of the implementation
- Configured Grafana, Prometheus and Locust, along with the scripts for its automation (along with Istio as well)
- Helped implement the software tests
- Helped reconfigure the system for auth0 integration
- Did part of the implementation of Spark's microservice
- Report

10.1.2 51058 - Diogo Lopes

- Designed the architecture of the system, focusing on the protobufs creation.
- Implemented some microservices (before the integration with mongo).
- Developed useful scripts for running the system locally during development.
- Reviewed microservices implementation.
- Created and implemented the use case for getting the top ten items with more likes.
- Report.

10.1.3 51072 - Miguel Saldanha

- Dataset choose
- Helping Specifiying Use Cases, Rest API, Architectural Design
- Producing and Reviewing microservices and protobufs
- Reviewing and Correcting containerize of the microservices
- Reviewing and Correcting Automatization of building and deployment
- Deploy and Configuration of Account Database
- Deploy on Kubernetes

- Configuring safe connections between microservices on Kubernetes
- Configuring scalability off microservices on Kubernetes
- OAuth2 Configuration, Deploy and Roles and Rules Management
- CI/CD pipeline
- Reviewing and Correcting Tests
- Creating Bash Files for automate deploy of microservices and Istio on Cloud
- Help Creating Stress tests calls.
- Implementing Spark, Review and Correcting Spark Microservice
- Some contributions on Report and Review

10.1.4 51080 - João Roque

- Updated System Architecture.
- MongoDB database creation, population and implementation of their Services
- Helped with Yaml API
- Microservices body and debugging
- Docker files and docker testing
- General debugging with local deployment
- Report.

10.1.5 51111 - João Afonso

- Design Systems Architecture.
- Create protobufs.
- Microservices: "library", "anime", "book" and "imdb".
- Helped with Docker & Kubernetes.
- Software testing & bug fixes.
- Helped with auth0.

10.2 Future alterations

- Clean up dead code, and relocate files and folders to improve readability.
- Stash Spark results and use it when "fresh", and when it needs to be updated then update, giving faster results.
- Expand our dataset to more types of Items.
- Dynamic creation of database Services (like Book, Anime and IMDB Service), all using the same communication protocol (protobuf), meaning we would only need to create different "Adapters" and then import them to the Library Service, making easier future integration with new Services.
- At last, a mobile application or website could be implemented having this API as its backbone.
- Adding images to every Item on our database, a link that could be loaded in any application or website. It would mean a database re-population, and changing some return statements, nothing too laborious.

11 Appendix

Results from testing our cloud deployment:

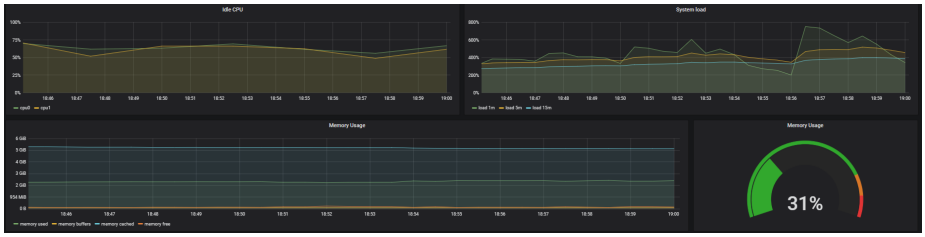

Figure 2: Stress testing with 800 users. CPU usage and system load plots

Figure 3: Stress testing with 800 users. CPU usage and system load plots

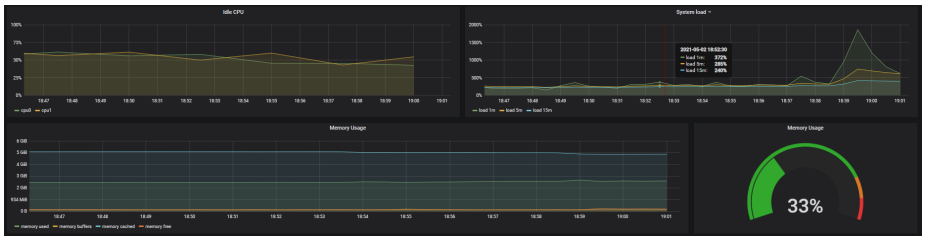

Figure 4: Stress testing with 800 users. CPU usage and system load plots

Figure 5: Stress testing with 800 users. CPU usage and system load plots

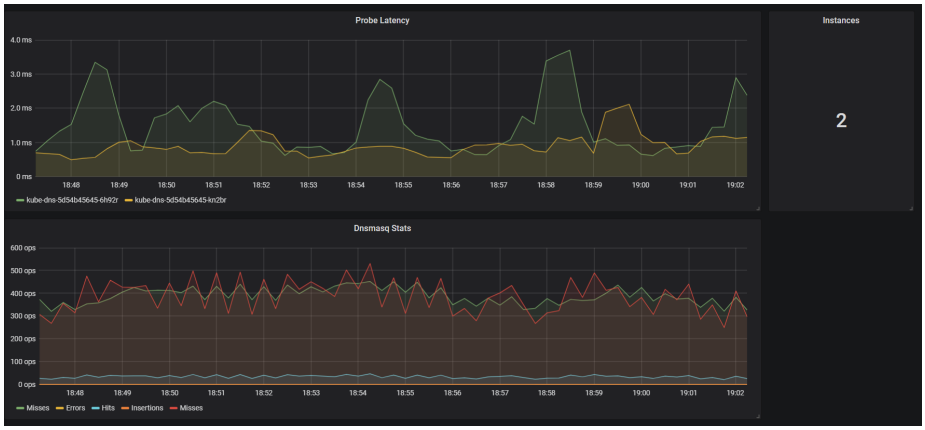

Figure 6: Stress testing with 800 users. DNS latency

Type	Name	# Requests
GET	/item/BOOK/607615b3aeb60e0f26f7c1df	2190
PUT	/item/BOOK/607615b3aeb60e0f26f7c1df/like	2229
PUT	/item/BOOK/607615b3aeb60e0f26f7c1df/seen	2230
GET	/lib/1	2358
POST	/suggest	2285
GET	/user/login	2237
GET	/user/logout	2234
PUT	/user/search/saldanha	2253
	Total	**18016**

Figure 7: Stress testing with 800 users, requests and count of each

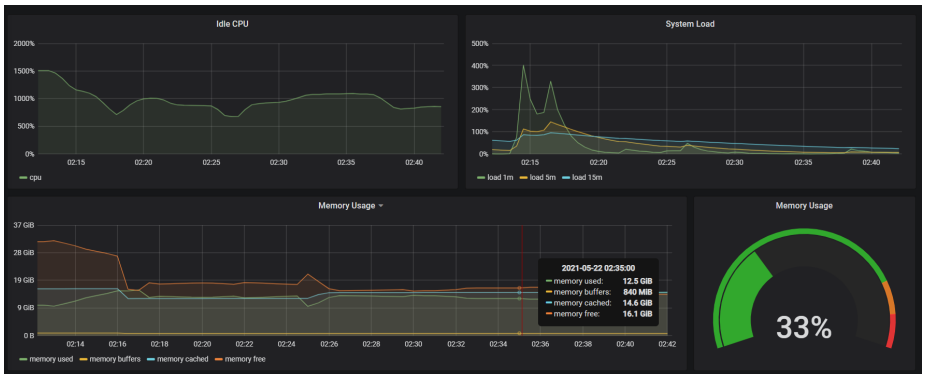

Figure 8: Stress testing with 6800 users. Idle CPU and system load plots

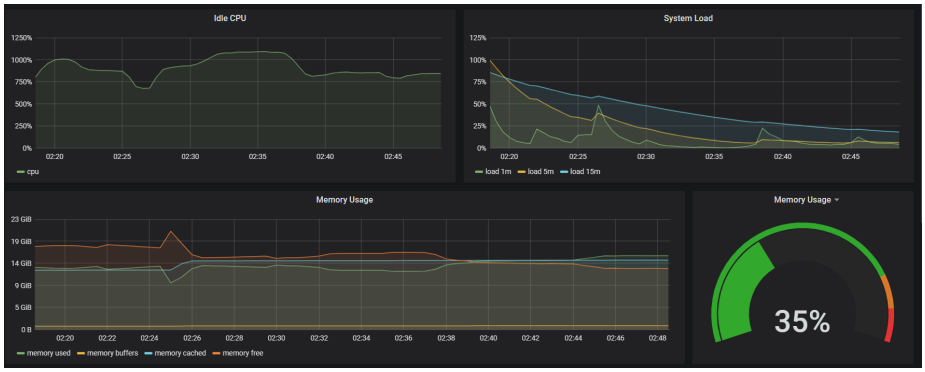

Figure 9: Stress testing with 6800 users after the system stabilized. Idle CPU and system load plots