SAPIENZA
UNIVERSITÀ DI ROMA

FACOLTÀ D'INGEGNERIA DELL'INFORMAZIONE INFORMATICA E STATISTICA

Bacterias Search Engine

CLOUD COMPUTING FINAL PROJECT

Professors:
Emiliano Casalicchio

Students:
Matteo Migliarini, 1886186
Giuseppe Di Poce, 2072371
Enrico Grimaldi, 1884443

Academic Year 20XX/20XX

Contents

1 Problem Description

The goal of this project is to develop an application that provides a search engine for images of bacteria. The search engine should be able to retrieve the most similar images to a given query image, uploaded by the user. The project will be deployed on Amazon Web Services and the ”algorithmic part” to compute similarity among images will be done using *Weaviate*. *Weaviate* is an open source **vectorial database** that is robust, scalable, cloud-native, and fast. It is a very powerful tool in terms of speed and flexibility: typically performs a 10-NN neighbor search out of millions of objects in single-digit milliseconds and in terms of flexibility you can use Weaviate to conveniently vectorize your data at import time. These vectorization options are enabled by Weaviate modules. Modules enable use of popular services and model hubs such as OpenAI, Cohere or HuggingFace and much more, including use of local and custom models.
It can be interpret as an open-source knowledge graph system that is designed to organize and connect diverse data sources using a decentralized architecture.
While Weaviate itself is not specifically focused on images, it can be utilized in conjunction with other technologies to develop an image search engine. We will use a tool like Weaviate **pre-trained neural network** to extract features from the images and create a fast index of reference items.
When a query image is provided, the search engine will compare the features of the query image with the features of the reference items in the index and return the most similar images.
The key point of the Weaviate usage is the embedding of images in a vectorial databese. When the user add an image (or any other data object) to Weaviate, the vector database computes a vector embedding for that image using a given model. This vector is then placed into an index, allowing the database to quickly perform searches by finding the most similar vectors in the database to a given query vector.

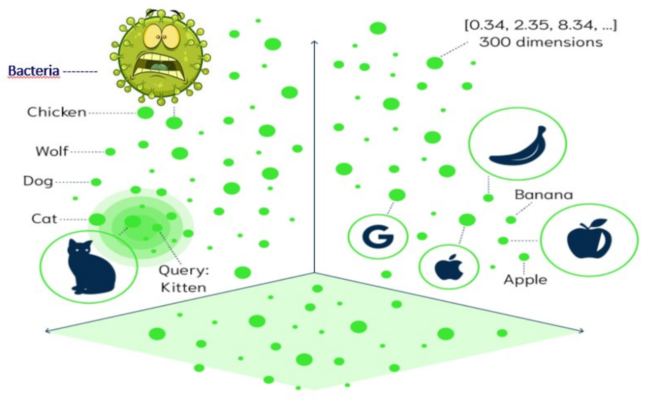


Figure 1:
Weaviate retrieve documents by meaning. With ML models it converts data objects (our images) to embeddings. Embeddings are vector representations, which are stored in a high-dimensional vector space. The following image shows a example of embedded concepts.

2 Solution Design

In order to address the problem we want to implement a scalable web application, capable of rapid elasticity.

2.1 Microservices

In order to allow the scaling of our application we would like to implement it through a set of microservices such that the single functions can be scaled individually. But identifying the set of microservices in which an application can be split in isn't a trivial task, however we identified three main components of our application:

- A web user interface from which our user can interact with the application;
- An API call for querying the database;
- The vector database itself.

What we're doing can be also visualized as a standard client-server application.

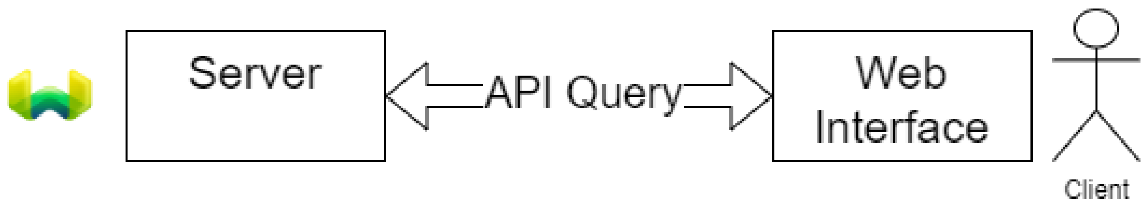


Figure 2: Client-Server architecture

2.2 The web interface

The client could be either thin or fat. In case of a thin client: the query, the web interface, HTML, CSS would be generated in the server and served to the client ready to visualize (*Dynamic Webpage*), in fact the web interface in that case would be something only for visualizing the results of the query. In order to implement this we would need some webpage rendering framework like Flask, Laravel, PHP, etc... which would have to be managed and scaled together with the requests.

Instead we decided to go with a fat client. The web interface isn't served by any managed server, but rather is just a plain HTML/CSS/JS placed on an edge node. Since this is just a *Static Webpage* it can be served blazingly fast by AWS, and we don't need to setup any scaling at all, since it's all managed by the cloud provider. Then once the webpage is served some JS functions inside it can perform API calls to the server in order to query and get results. Then JS manages the update of the UI, directly on the client machine. The downside is that we require the client to perform some computation to update the webpage, but since these computations shouldn't be very intensive, this shouldn't be a problem even for lower performing clients.

2.3 The API call

This is just a simple call which connects the client to the server. Following the microservice design pattern we implemented this with a standard REST API, particularly with a POST since we need to payload also the query image in the request and we need to payload a list of images in the response.

2.4 Vector database

We chose to use Weaviate as our vector database. Weaviate is an Open Source database, but the company developing it also offers a service (Weaviate Cloud Service) which already integrates the Weaviate software as a PaaS, allowing for scalability.

But of course we cannot use it, so our goal will be to reimplement it in another Cloud Provider (AWS). A vector database uses vector representations of its items in order to find similar items to the query through cosine similarity. So the vector database is actually made up of two components:

- A pretrained Neural Network that transforms items into vectors;
- The database software itself which handles queries, holds the schema, and the vector-item mappings.

If we find a way of actually split the Weaviate software in this components we could have a great advantage since we could run the Neural Network on specialized hardware with GPU enabled, while running instead the other part on memory-specialized hardware. This would allow us to make a good use of our resources, while also splitting this microservice into 2 microservices, allowing for more functional scaling.

2.5 Why to scale?

Scaling a web application offers several significant benefits, first and foremost, scaling enhances the performance of the application. As the user base and traffic to your image search engine increase, a single EC2 instance might not be sufficient to handle the growing workload. By scaling horizontally and adding more EC2 instances, you distribute the load across multiple servers. This distribution of workload reduces response times, ensures faster query processing, and ultimately improves the overall performance of your application. Notice that, as far as possible from the resources available to us from our academy accounts, we have tried to increase the resources of our ec2 instances by increasing the CPU and RAM of each instance.

Additionally, scaling on EC2 enhances the availability of webb application. With a single instance, if that instance fails, the application would become inaccessible. However, by utilizing multiple instances and incorporating an elastic load balancer, we can achieve high availability. The load balancer automatically distributes incoming traffic across the healthy instances, redirecting requests away from any failed instances. This redundancy ensures that even if one instance goes down, the application remains accessible, minimizing downtime and maintaining continuity of service.

Scalability also offers flexibility in resource allocation. As the demand for the image search engine fluctuates, scaling allows you to adjust resources accordingly. During periods of high demand, such as peak hours or events, we can easily add more instances to handle the increased workload. Conversely, during periods of lower demand, we have the flexibility to scale down, reducing the number of instances and optimizing resource utilization. This dynamic allocation of resources helps you save costs by only paying for the resources you actually need at any given time.

Another benefit of scaling is that it enables to plan for future growth. As the image search engine gains popularity and attracts more users, the demand for we application will increase. Scaling on EC2 allows to easily accommodate this growth by adding more instances as needed. This scalability planning ensures that search engine application can handle the growing user base and increased traffic without sacrificing performance or user experience.

3 Solution implementation

Basically, for technical choices in implementation, we can divide our decision-making strategy into three parts:

- choice of technologies, languages and frameworks for developing the web app locally;
- use of containers and "dockerization" of the simple application for better efficiency and scalability in both development and deployment;
- choice of cloud services offered by Amazon to meet scalability and availability performance without implying high maintenance costs of the service itself.

3.1 Web technologies

First of all, we chose a rather classic setup for the development of the application:

- for the front-end CSS
- for the back-end JavaScript with Vue.js framework
- HTML obviously as the basic structure of the page.

JavaScript seemed to us the best language to be able to implement the back-end of the application very quickly. In fact it is an extremely versatile language, flexible and obviously suitable for handling event-driven programming and communication with services in AWS via AWS lambda functions. The ease with which to handle events (such as uploading the image to the site) via JavaScript allows for agile communication with Amazon's cloud services, particularly with the API for lambda functions. Initially in fact we had planned to use Python for the back-end by writing a Flask-based app, simply propagating our demo written in Jupyter Notebook on a Python file that would act as a search engine for the most similar images. Flask, however, immediately proved to be less intuitive and less suitable for handling lambdas

Vue.js is a JavaScript framework for building user interfaces. It builds on top of standard HTML, CSS, and JavaScript and provides a declarative and component-based programming model that helps you efficiently develop user interfaces, be they simple or complex. We need a technology that can grant dynamism to the current page, and the implementation choice fell on Vue.js because it is familiar and intuitive to us.

Finally, we can say that the site itself is not very complex and challenging. In fact, we desire a clean, simple and minimal design: the modules of our application are not many and the type of information/resource offered is not extremely wide and varied, but rather particularly specific. We therefore opt for a structure that is as basic as possible, which will allow yes the app to function properly and a good smoothness and fluidity in the graphical interface, but we direct our energies toward simplicity and mmediateness. We therefore simply exploit HTML and CSS without additional front-end frameworks (and avoid detailing the responsiveness of the pages since the project is aimed only at a deasktop audience with no special requirements).

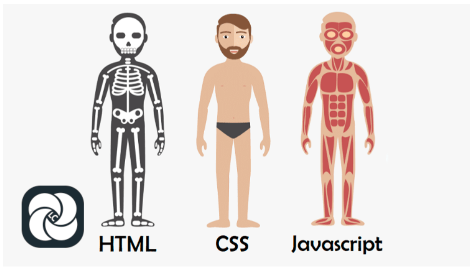


Figure 3: HTML provides the basic structure, JavaScript adds interactivity and functionality, and CSS defines the visual presentation, making them the skeleton, muscle, and body of a web page, respectively..

3.2 Containers and Dockerization

As mentioned above, Weaviate and our pre-trained neural network will lie in docker containers, contained by an Aws ECS service. The primary motivation for employing Docker containers is to ensure consistent and reliable deployment of our application across different environments. Docker provides a lightweight and portable solution that encapsulates the entire application stack, including dependencies and configurations.
We built the weaviate and NN images from a *.yaml* file as follows, where at line code 13 (semitechnologies/weaviate:1.19.8) and line 30 (semitechnologies/img2vec-pytorch:resnet50) are difned the images for our Database - ML tools. Note also that we connected the containers to the host rosurces (ours) in terms of [gpu] computational resources (see line 39):

```
version: '3.4'
services:
  weaviate:
    command:
    - --host
    - 0.0.0.0
    - --port
    - '8080'
```

```
      - --scheme
      - http
      image: semitechnologies/weaviate:1.19.8
      ports:
      - 8080:8080
      restart: on-failure:0
      environment
        IMAGE_INFERENCE_API: 'http://i2v-neural:8080'
        QUERY_DEFAULTS_LIMIT: 25
        AUTHENTICATION_ANONYMOUS_ACCESS_ENABLED: 'true'
        PERSISTENCE_DATA_PATH: '/var/lib/weaviate'
        DEFAULT_VECTORIZER_MODULE: 'img2vec-neural'
        ENABLE_MODULES: 'backup-filesystem,img2vec-neural,ref2vec-centroid'
        CLUSTER_HOSTNAME: 'node1'
        BACKUP_FILESYSTEM_PATH: '/tmp/backups'
      volumes:
      - /var/weaviate:/var/lib/weaviat
      - backups-weaviate:/tmp/backups
    i2v-neural:
      image: semitechnologies/img2vec-pytorch:resnet50
      environment:
        ENABLE_CUDA: '0'
      deploy:
        resources:
          reservations:
            devices:
              - driver: nvidia
                count: 1
                capabilities: [gpu]

volumes:
  backups-weaviate:
```

This approach enables us to create a reproducible and self-contained environment, eliminating any potential issues related to variations in underlying infrastructure. Docker containers enable efficient resource utilization. Each container can be allocated a specific amount of computing resources, such as CPU and memory, based on its requirements. This level of resource isolation ensures optimal performance and prevents one component from negatively impacting the others. It also allows us to easily scale our system by deploying multiple containers in parallel, ensuring high availability and load balancing.

Furthermore, it should be noted that our lambda function will only be triggered by the action of uploading an image by the user on our web app, so by decoupling the user interactions from the neural network and Weaviate, we achieve greater scalability and responsiveness. The Lambda function can be invoked as needed, providing real-time responses without the need for continuous resource allocation.
By leveraging docker containerization technology, we can create a robust and scalable system that delivers optimal performance while ensuring flexibility and ease of management.

3.3 Amazon Web Services

Regarding the use of cloud services we plan a number of possible implementation alternatives, very different from each other according to different aspects such as: simplicity of design, performance and different trade-offs. In each case we are sure to use: an S3 bucket in which to ”deposit” the source code of the site; lambda functions to exploit user-directed triggers for activating cloud services and the layout of outputs on the user interface.

3.3.1 First solution: EC2

The simplest solution involves the use of an EC2 instance on which to directly upload the already containerized application. In this first, fairly trivial example, a call to lambda function via a specific event on the web page (in an S3 bucket) is then always expected, and the lambda code will then provide for using the EC2 computational resource to launch the two docker images. Finally, the computed output will be returned dynamically and asynchronously (with respect to the request from the client).
The following steps then power to good scalability performance:

1. Set up an Auto Scaling group: we create an Auto Scaling group using our Launch Configuration. We specify the desired capacity, minimum and maximum number of instances, and scaling policy. The Auto Scaling group will automatically launch and terminate instances based on the defined criteria.

2. Configure a load balancer: we set up an Elastic Load Balancer (ELB), in particular an Application Load Balancer (ALB) to distribute incoming traffic across our EC2 instances. We configure the load balancer to listen on the appropriate ports and protocols for our web app, i.e. port 80 for the HTTP protocol.

3. Add instances to the load balancer: we associate the instances launched by the Auto Scaling group with our load balancer. This ensures that traffic is evenly distributed among the instances (fairness performance).

4. Monitor and adjust scaling: finally we can monitor the performance of our web app and adjust the scaling policies as needed.

The problem here naturally arises: how the computational resources should scale should depend on the type of task. In fact, the type of resources used varies most critically depending on the specific goal, and and in our case we need different scaling policies for the neural network and the vector database. In addition, the two containers require very different performance and characteristics in terms of hardware: the former (the neural network) possibly requires a GPU, and for the latter, the most critically important variables are the storage and RAM usage levels.

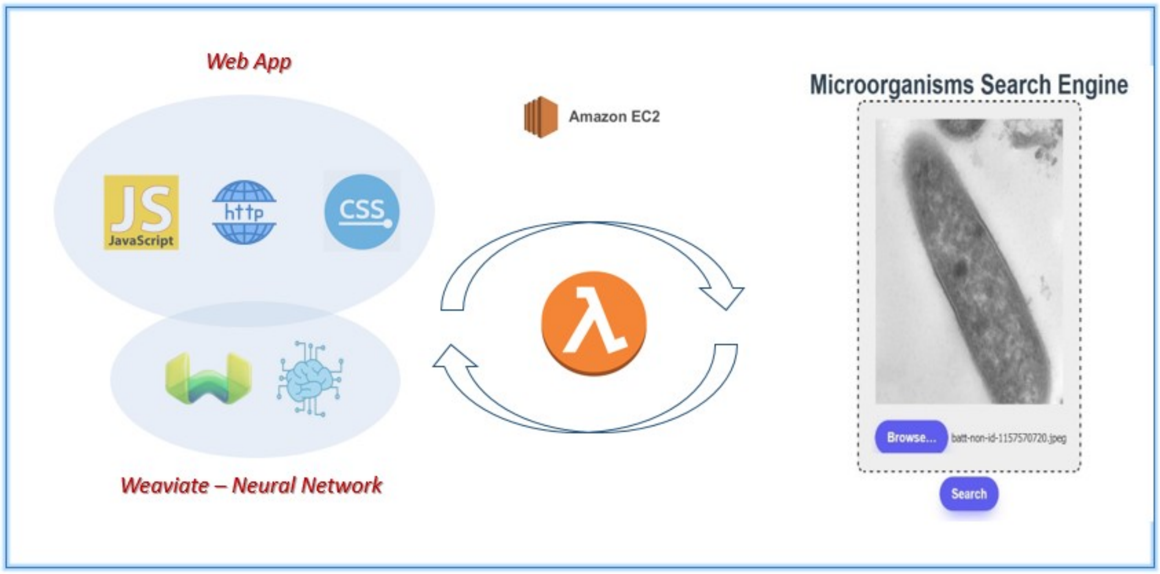

Web App
JS
JavaScript
http
CSS
Amazon EC2
Microorganisms Search Engine
Browse...
batt-non-id-1157570720.jpeg
Search
Weaviate – Neural Network

Figure 4: The picture explain the first hypothesis. Database, models and our app running on top of a EC2 instance. In this scenario we have our lambda function that is triggered by an event (upload of images by the user in the web app). The *architecture* here is minimal, we put everything inside an instance that communicate with the λ function.

3.3.2 Second solution: ECS

We therefore decide to opt with this second implementation, which seems to improve our overall performance and better reflects the context described above:

1. Push Docker images to Amazon ECR: Amazon Elastic Container Registry (ECR) is a fully-managed container registry that allows you to store, manage, and deploy Docker images. Push your Docker images to ECR so that ECS can pull them for deployment.

2. Create ECS task definitions: we define two ECS task definitions, each referencing one of our Docker images. Task definitions describe the container configuration, resources required, environment variables, and any other settings.

3. Create ECS services: we create two ECS services, each associated with a specific task definition. ECS services manage the desired number of tasks and ensure

that the specified number of tasks are running and healthy. Each service will run on a separate EC2 instance.

4. Configure scaling: For each ECS service, we configure scaling based on specific requirements of RAM and GPU utilization.

5. Configure load balancing: we set up an Application Load Balancer (ALB) in front of our ECS services.we configure the load balancer to distribute incoming traffic across the services running on different EC2 instances.

6. Monitor and adjust scaling: we continuously monitor the performance of our web app and adjust scaling policies for each service accordingly. We use both ECS-specific metrics and CloudWatch to gather and analyze relevant data for scaling decisions.

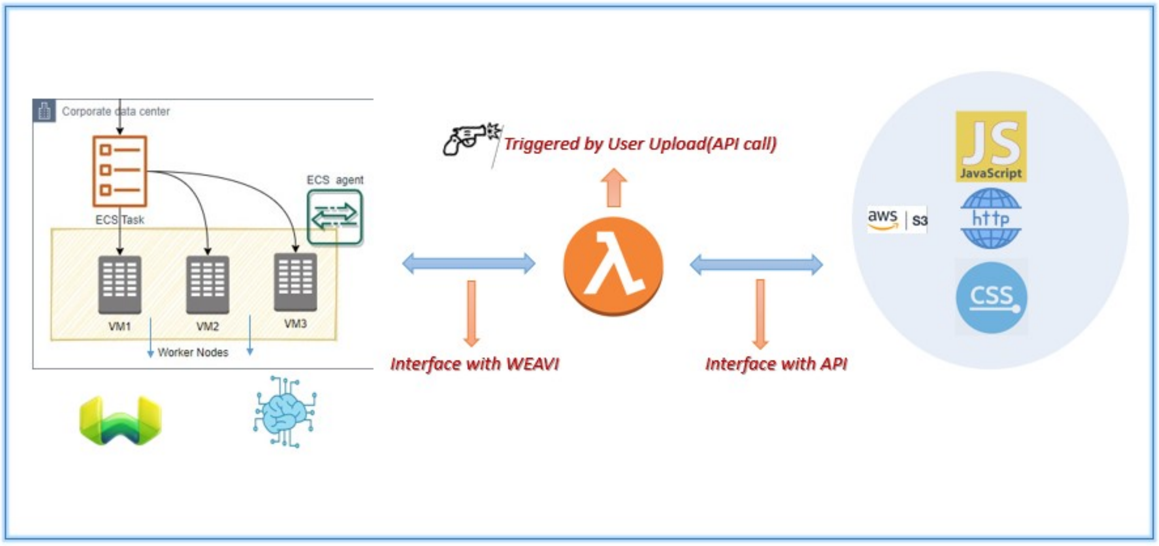


Figure 5: The picture explain the *modus operandi* of our project. Database and models built in a ECS service and our app onto a S3 bucket. In this scenario we have our lambda function that is triggered by an event (upload of images by the user in the web app). The λ communicate with Weaviate (and just with it for the left side architecture) and with the API for the web app side.

4 Deployment procedure

4.1 Virtual Private Cloud

After the launch of the EC2 instance (that acts as a web server) we need to create a Virtual Private Cloud (VPC) on AWS and configure various components, including subnets, security groups, and route tables. Notice that we also will make an AMI, that act like a screenshot of the ’checkpoint’ from which we want to resume the instance we are working with.
We can very quickly resume the steps that are involved as follow:

1. Create a VPC with a public subnet and a private subnet in one Availability Zone. Configure the VPC settings, including the CIDR blocks for the subnets and the number of availability zones.
2. Create additional subnets in a second Availability Zone, including a public subnet and a private subnet. Associate the appropriate route tables with the subnets.
3. Create a VPC security group to act as a virtual firewall. Configure inbound rules to allow HTTP access from anywhere.
4. Launch an EC2 instance named ”Weaviate-Search-Engine” using the Amazon Linux 2023 AMI: Select the appropriate VPC, subnet, and security group. Configure the instance to automatically interact with Weaviate(our web server) and a web application using our data script.

After that the instance pass the status checks, copy the Public IPv4 DNS value and open it in a web browser to verify the web server is correctly running.

By following these steps, that are basically what is described in the Learner Lab number 2 on AWS, we can successfully use the EC2 instance for our purposes and access to the bacteria vectorial database (Weaviate). **Please notice that we never interact directly with the convolutional neural network with which Weaviate interfaces.**

4.2 Create Web Group & define EC2 rules

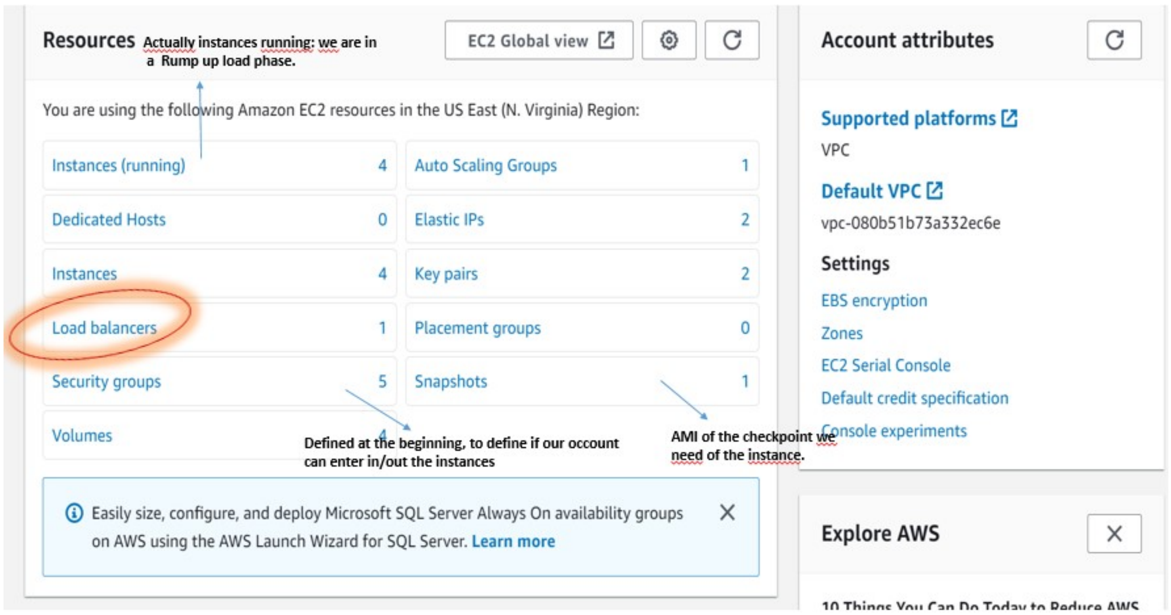
After the creation of the VPC as described above, we need to define the ’attributes’ of the instance through which we communicate with the Database.
First of all we need to add a script to generate the Weaviate image (claim that it lies on a docker container but we don’t see it, because we interact just with the EC2 instance). We are using the following script:

```
#!/bin/bash

# update and install docker
sudo yum update -y
sudo yum install -y docker
sudo service docker start
sudo usermod -a -G docker ec2-user

# install docker-compose
sudo curl -L https://github.com/docker/compose/releases/latest/download/docker
compose-$(uname -s)-$(uname -m) -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose

# download YAML
cd /home/ec2-user
curl -o docker-compose.yml
"https://configuration.weaviate.io/v2/docker-compose/docker-compose.yml?
generative_cohere=false&generative_palm=false&image_neural_model=keras-

resnet50&media_type=image&modules=modules&ref2vec_centroid=true&reranker_cohere=
false&runtime=docker-compose&weaviate_version=v1.20.1"

sudo docker-compose up

some rows of this code are cutted due to too long row.
```

Furthermore we 'customize' the EC2 tool with Linux 2023 (as described above), 30G of storage and the latter VPC. We enable also the Cloud-Watch to check usage of resources. An important stuff to notice in this section is the definition of *Web Groups* in the inbound rules: we add SSH and HTTP protocols to be able to communicate with web application(http) and entering in the EC2 virtual machine provided by the AWS cloud service.

4.3 Scale and Load Balance the Architecture

In order to develop our web app architecture and be able to scale it in base of the different load, we need to define an "Application Load Balancer".
It helps in achieving high availability and scalability for our applications and will distribute incoming application traffic across EC2 instances.
After a trial and error procedure we arrive at the conclusion that the best settings for LB is a minimum of 2 and a maximum of 6 EC2 instances.



Resources

Actually instances running: we are in a Rump up load phase.

EC2 Global view

You are using the following Amazon EC2 resources in the US East (N. Virginia) Region:

Instances (running)	4	Auto Scaling Groups	1
Dedicated Hosts	0	Elastic IPs	2
Instances	4	Key pairs	2
Load balancers	1	Placement groups	0
Security groups	5	Snapshots	1
Volumes	4		

Defined at the beginning, to define if our occount can enter in/out the instances

AMI of the checkpoint we need of the instance.

Easily size, configure, and deploy Microsoft SQL Server Always On availability groups on AWS using the AWS Launch Wizard for SQL Server. **Learn more**

Account attributes

Supported platforms

VPC

Default VPC

vpc-080b51b73a332ec6e

Settings

EBS encryption

Zones

EC2 Serial Console

Default credit specification

Console experiments

Explore AWS

10 Things You Can Do Today to Reduce AWS

Figure 6: This is our resources summary of the EC2 Global view.
As you can see we are in a rump up phase, thanks to the Load Balancer work that scale our application in base of the increasing load.

Basically what we want is a tool that makes intelligent routing decisions based on the content of the HTTP/HTTPS client requests.

5 Results

5.1 Web Application

The construction of the website, as explained in section number two, was done using the tools of HTML, CSS, JavaScript. The site was constructed in such a way as to have a user-friendly interface. The following demonstration shows how the user can simply enter a query image and get back all the most similar images.

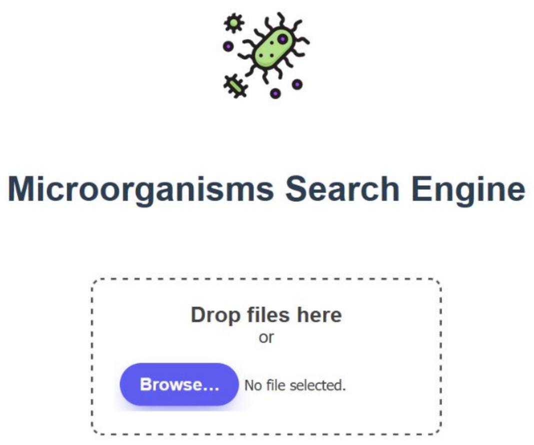
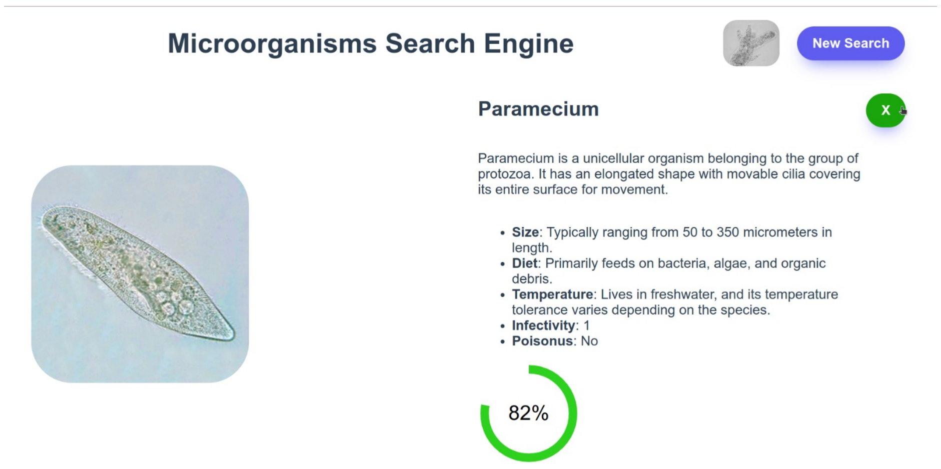


Figure 7: User directly interface of the web application.

In the example shown, given a bacterium of the class ’Amoeba’ as input, the most similar images correspond to the same battery species, underlining the excellent performance of our ML models in terms of accuracy. In addition, a description of the bacteria was included for each image, giving fuetaures of the bacteria such as a brief description, size, temperatures at which they live, what they feed on and so on. The site is static and served by an S3 bucket so it grants low latency.



Paramecium

Paramecium is a unicellular organism belonging to the group of protozoa. It has an elongated shape with movable cilia covering its entire surface for movement.

- **Size**: Typically ranging from 50 to 350 micrometers in length.
- **Diet**: Primarily feeds on bacteria, algae, and organic debris.
- **Temperature**: Lives in freshwater, and its temperature tolerance varies depending on the species.
- **Infectivity**: 1
- **Poisonus**: No

Figure 8: Details of a query results after the action of a user that upload a bacteria query image. In this case is shown a Paramecium

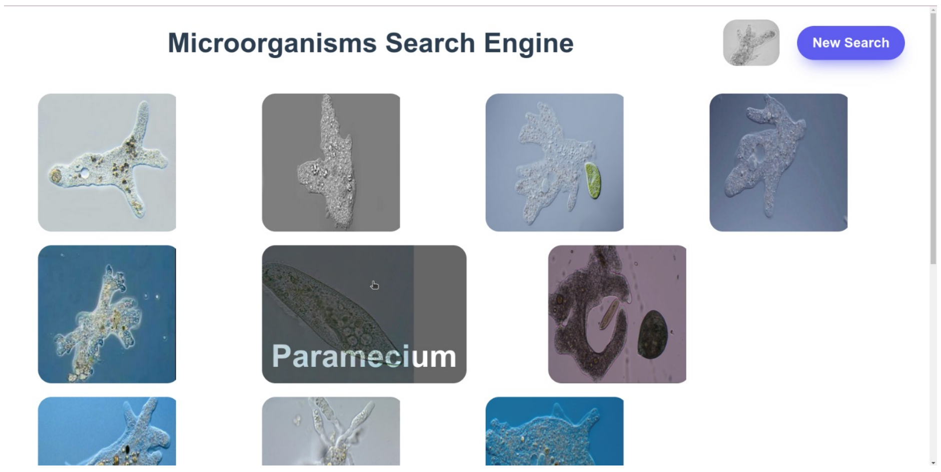


Figure 9: This is the most similar images retrieved by our search engine.

5.2 Scalability results

We used the CloudWatch service to monitor the scalability of our EC2 instance, using the AutoScaling Groups. As expected, as the load of mean CPU usage increases above the 60% of the max available CPU, the cloud services (a combination of AutoScaling Groups, Launch Template, Target Group and an Application Load Balancer) allow the correct satisfaction of users' needs (an high number of queries determines an high usage of computing resources).

We can note 2 peaks for each instance as they derive from stready periods that are when the application is stressed.

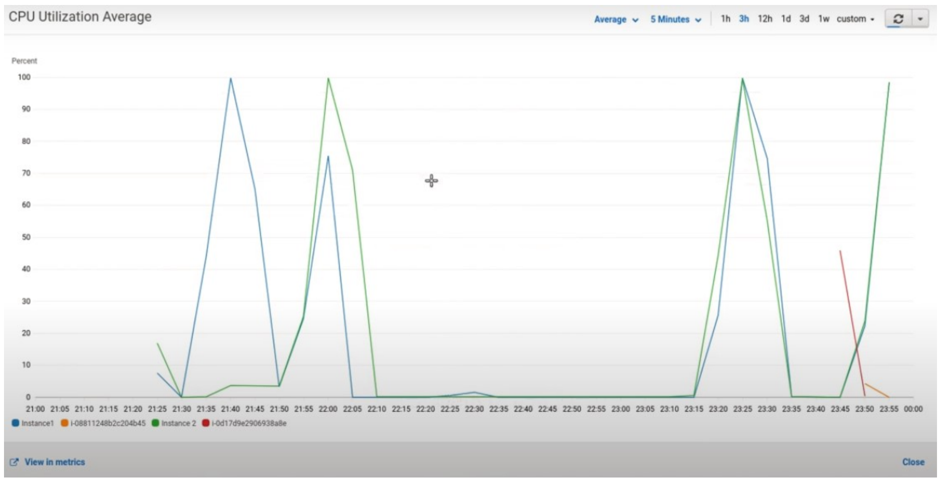


Figure 10: CPU summary utilization from our CloudWatch Monitor.

References

[1] Leslie Lamport. *LaTeX: a Document Preparation System.* Addison Wesley, Massachusetts, 2 edition, 1994.