Voicefork

Cloud Computing course project
A.A. 2022-2023

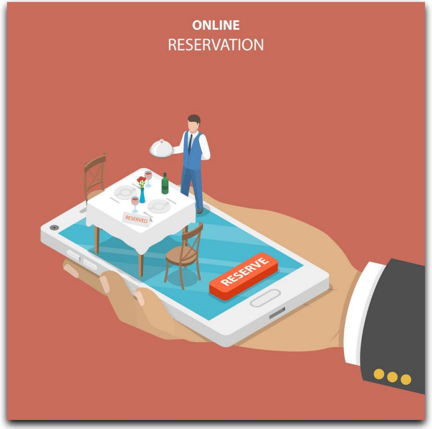
Faculty of Ingegneria dell'informazione, informatica e statistica
Department of Informatica

SAPIENZA
UNIVERSITÀ DI ROMA

Participants:
Corsi Danilo - 1742375
Lucciola Alessio - 1823638
Scarcelli Domiziano - 1872664

Introduction

- **Task:** Build a mobile application that allows users to manage their reservations at restaurants.
- **Goal:** Realize a structure based on several **microservices**, so that the application is **highly maintainable** and **scalable**.



Voicefork allows users to make online reservations at restaurants

Design of the solution

- Backend structured in different microservices
- Each microservice it's independent (has its own purpose and database)
- API built with **REST** technology used for single, well-defined tasks
- Additional technologies:
  - MinIO: To store any non relational data such as images
- **Docker**: Used in the local development environment

Minio Logo

Docker Logo

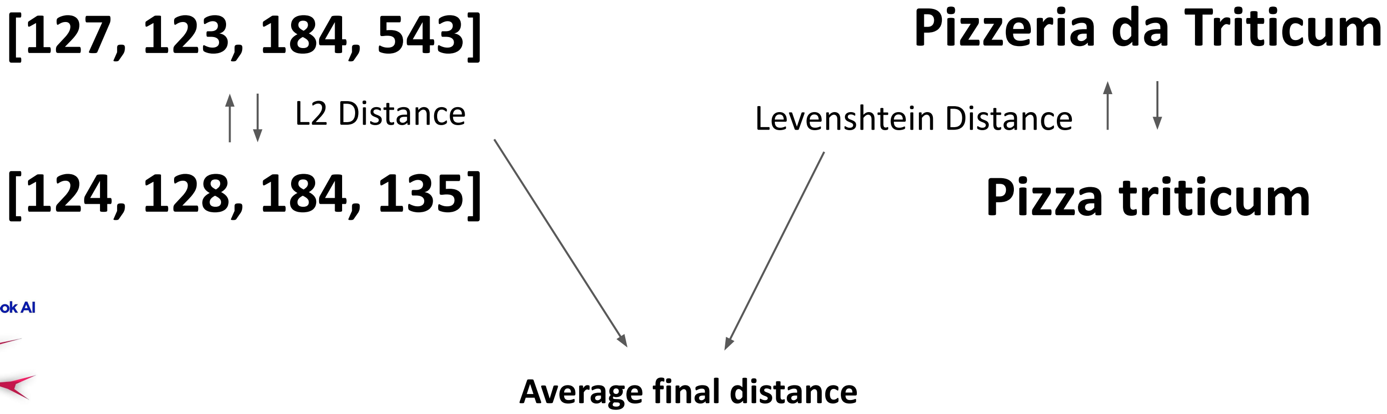
Design of the solution - Microservices

We implemented 5 microservices:

- **Users:** It handles users operations (registration and login)
- **Reservations:** It handles reservations operations (creation, visualization)
- **Restaurants:** It handles restaurants operations (searching)

Design of the solution - Microservices

- **Embeddings:** Used when searching for restaurants (encode restaurants characteristics, returns top k restaurants whose name is the most similar from the query)



[127, 123, 184, 543]

L2 Distance

[124, 128, 184, 135]

Pizzeria da Triticum

Levenshtein Distance

Pizza triticum

Average final distance

FAISS
Scalable Search With Facebook AI

Design of the solution - Microservices

- **Nginx:** It connects all the microservices under the same URL

https://voicefork-api.com/

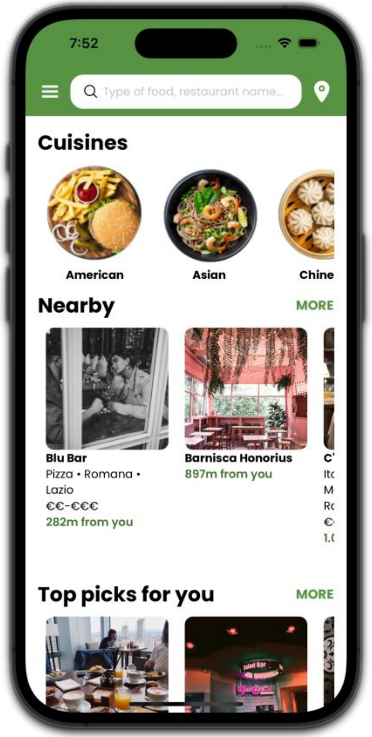
https://voicefork-api.com/restaurants/

https://voicefork-api.com/users/

https://voicefork-api.com/reservations/

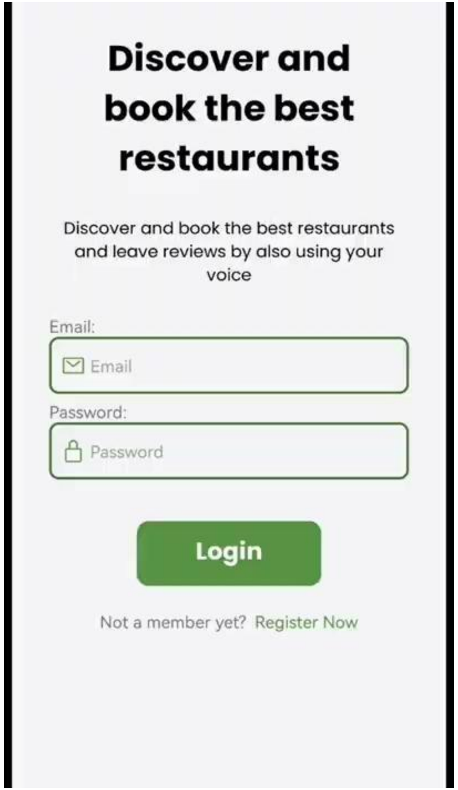
Frontend Application

- Built with **React Native**
- Cross-platform application available on **Android** and **IOS**
- The main features are:
  - Login and registration
  - Searching for restaurants
  - Making reservations
  - Managing reservations
- Each microservice is involved in a certain task



Voicefork homepage

Frontend Application - Login and Registration



Discover and book the best restaurants

Discover and book the best restaurants and leave reviews by also using your voice

Email:

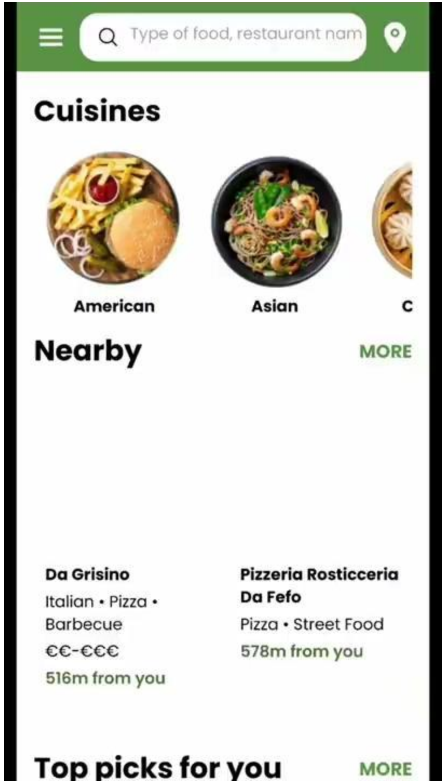
Email

Password:

Password

Not a member yet? Register Now

Frontend Application - Searching and reservation



Cuisines

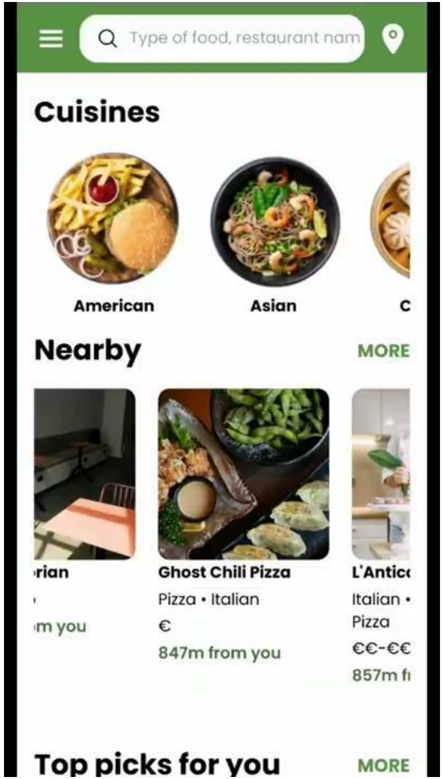
Nearby

Da Grisino
Italian • Pizza • Barbecue
€€-€€€
516m from you

Pizzeria Rosticceria Da Fefo
Pizza • Street Food
578m from you

Top picks for you

Frontend Application - Reservation list



Deployment

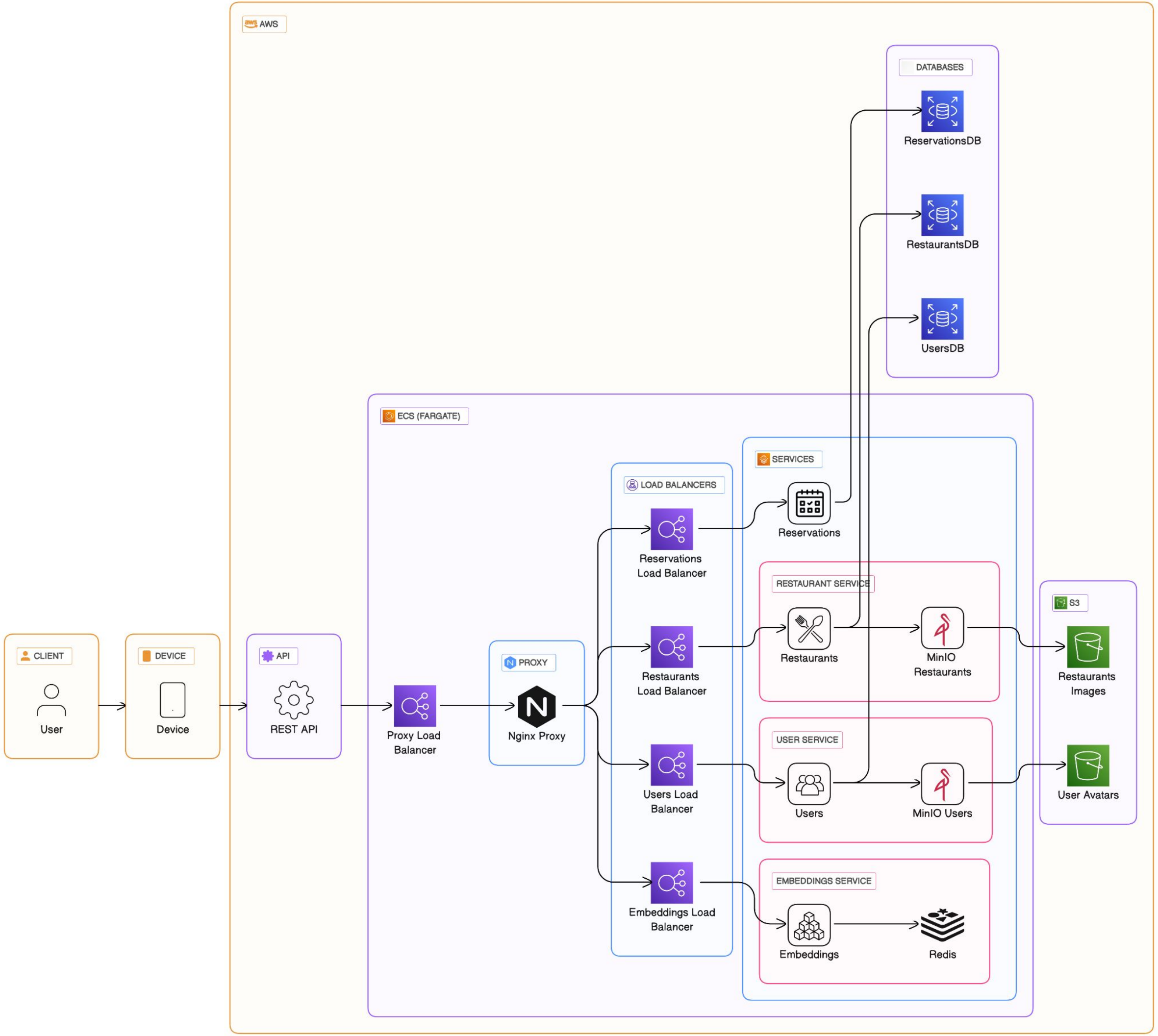
- Backend deployed on AWS:
  - **AWS ECS:** To deploy the microservices
    - **Task definition:** To replicate the docker configurations
    - Each microservice has its own container
  - **AWS Fargate:** To run containers
  - **AWS S3:** To store the unstructured data (images)
  - **MinIO:** To interface with AWS S3
  - **Amazon RDS:** To run the databases
- **Terraform:** To declare the entire cloud infrastructure through a set of files and initialize it with a simple command.

Amazon ECS

Amazon Fargate

Amazon S3

Amazon RDS


AWS
DATABASES
ReservationsDB
RestaurantsDB
UsersDB
ECS (FARGATE)
SERVICES
LOAD BALANCERS
Reservations
Reservations Load Balancer
RESTAURANT SERVICE
Restaurants
MinIO Restaurants
S3
Restaurants Images
CLIENT
User
DEVICE
Device
API
REST API
Proxy Load Balancer
PROXY
Nginx Proxy
Restaurants Load Balancer
USER SERVICE
Users Load Balancer
Users
MinIO Users
User Avatars
EMBEDDINGS SERVICE
Embeddings Load Balancer
Embeddings
Redis

Real Scenario Experiments

- In order to simulate the application in a real scenario we used real restaurants:
  - “Tripadvisor European Restaurant” dataset
  - More than 200k Italian restaurants
  - Each restaurant contains real information such as name, address, average rating, type of cuisine, etc..
  - Dataset plugged into the database hosted on Amazon RDS

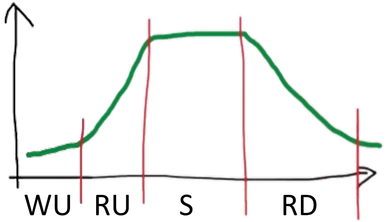
Tripadvisor Logo

Dataset source: https://www.kaggle.com/datasets/stefanoleone992/tripadvisor-european-restaurants

Design of experiments

- **K6**: tool that enables setup and run load tests
  - Simulate scenarios and application behavior
  - Provides real-time metrics and reports

- **Pipeline phases**



- **Microservices specs**

512 units (0.5 vCPU)
(Embeddings - 1 vCPU)

+ more resources

1024MB
(Embeddings - 4096MB)

+ more resources

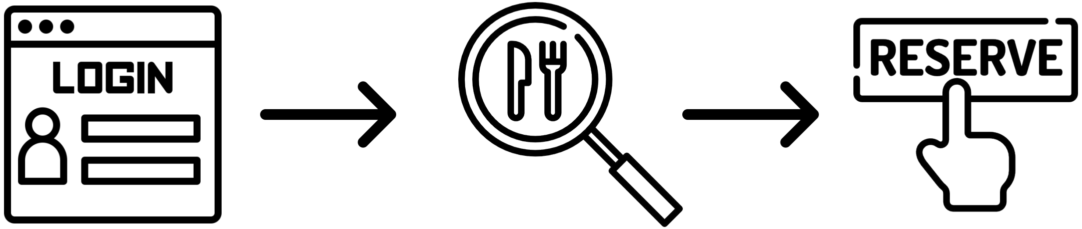
Load balancer
(distribute traffic)

- **Autoscaling target**
  - Min capacity: 1
  - Max capacity: 10

- **Autoscaling policy**
  - Target value: 60% (average percentage of CPU utilization)
  - Scale-out cooldown: 120 seconds
  - Scale-in cooldown: 180 seconds

Design of experiments - Scenario

- The user has a real interaction with the application (simulate the behavior of the actual system)
- **Example:**

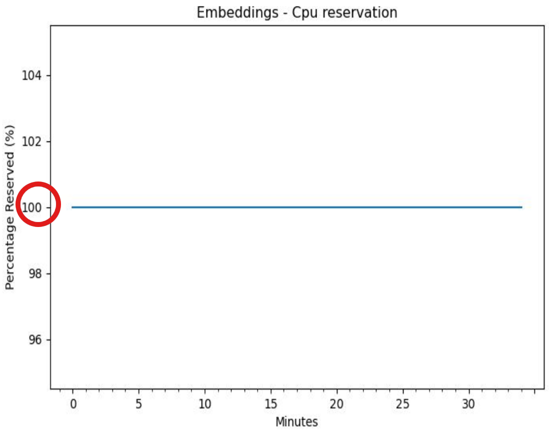


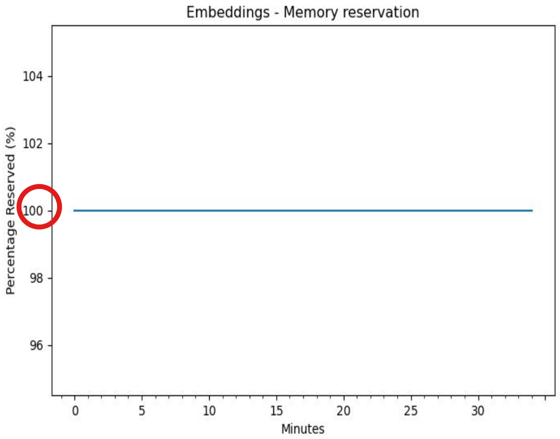
Access → Search → Reserve

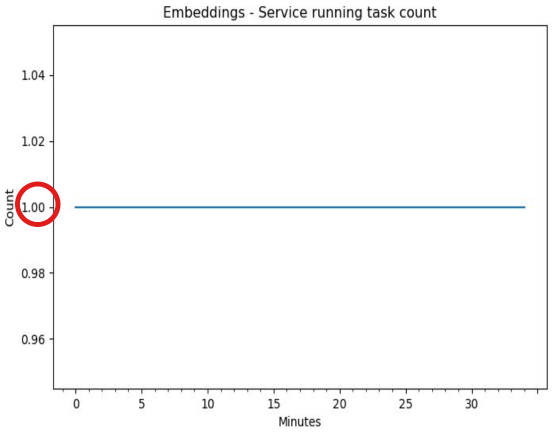
- All actions are randomized (users can perform any action at any time)
- Some actions are constrained by others (e.g. login before making reservation)
- A total of 10 tests were conducted (each lasting 30 minutes), the results were averaged

Results of experiments - Embeddings

- No scaling operation (CPU utilization never exceeds 60%)
- CPU and memory reservations are constants (as well as the running task count)


Embeddings - Cpu reservation

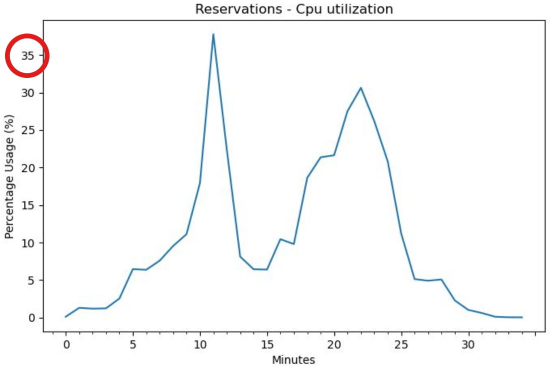

Embeddings - Memory reservation

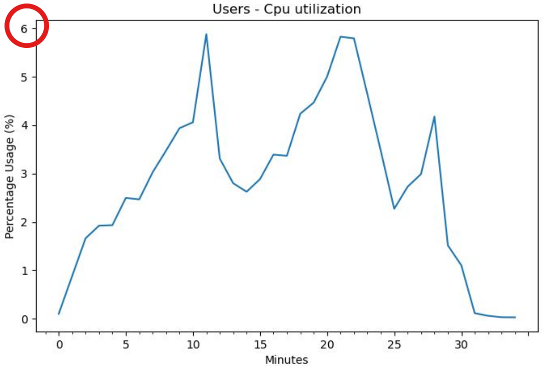

Embeddings - Service running task count

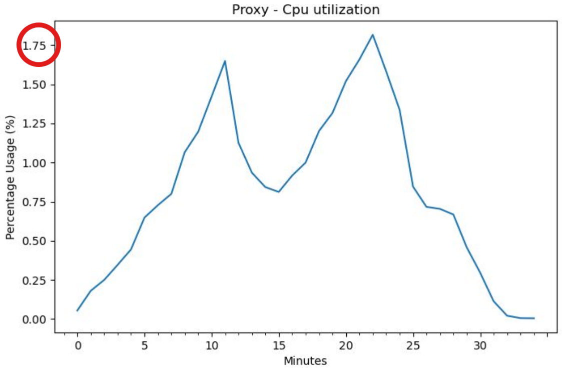
- **"Reservation"** refers to the portion of memory and cpu that is allocated for each individual task

Results of experiments - Other microservices

- Have a similar trend (they don’t perform any computationally heavy operations even if under a substantial load)



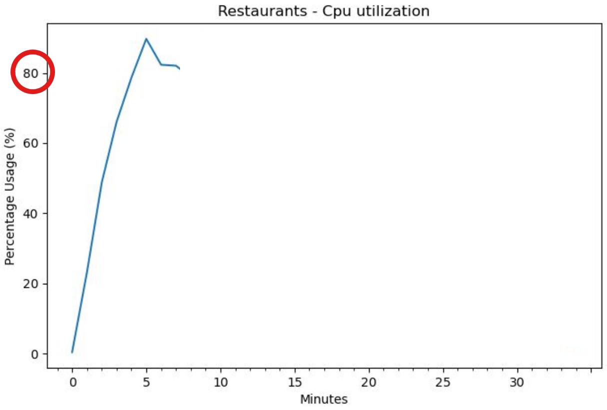


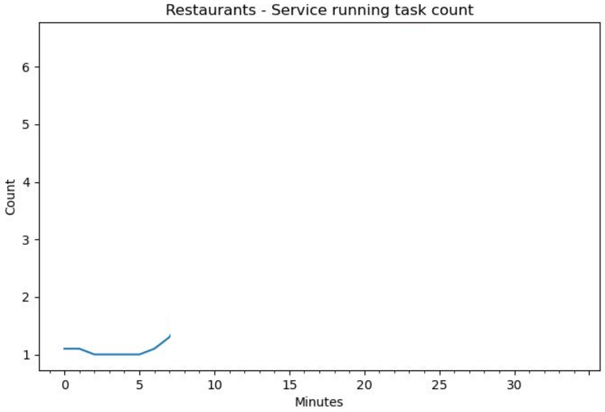


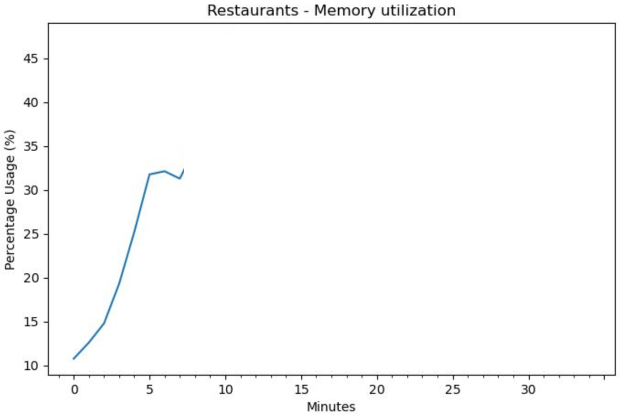
Results of experiments - Restaurants

Results of experiments - Restaurants

- There is a sudden increase with a peak of 80% of cpu utilization (target value was 60%)

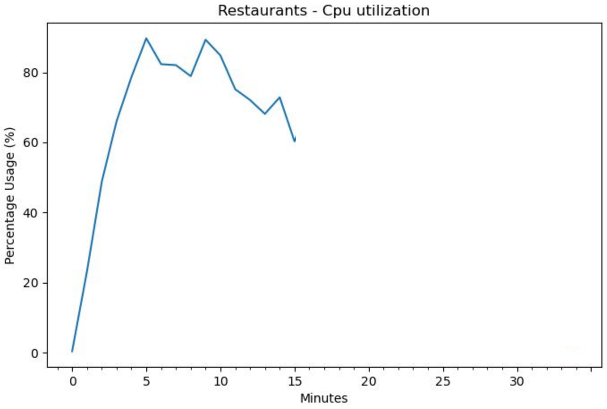

Restaurants - Cpu utilization


Restaurants - Service running task count

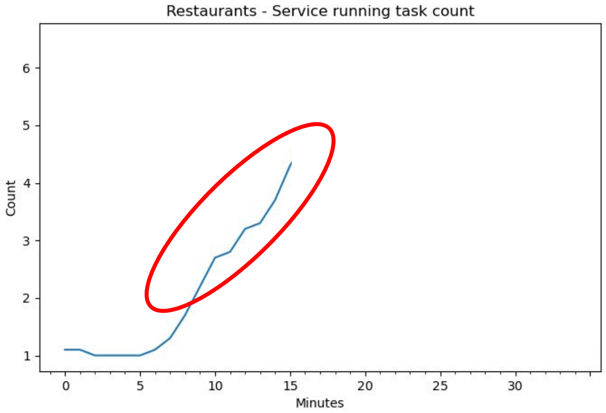

Restaurants - Memory utilization

Results of experiments - Restaurants

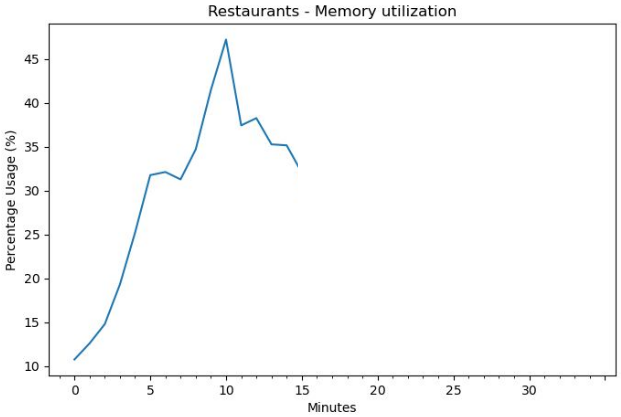
- There is a sudden increase with a peak of 80% of cpu utilization (target value was 60%)
- After the scale-out cooldown the scaling system starts working by adding new running tasks



Restaurants - Cpu utilization



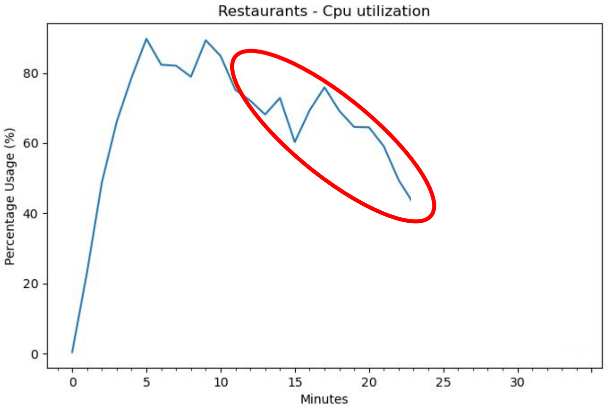
Restaurants - Service running task count

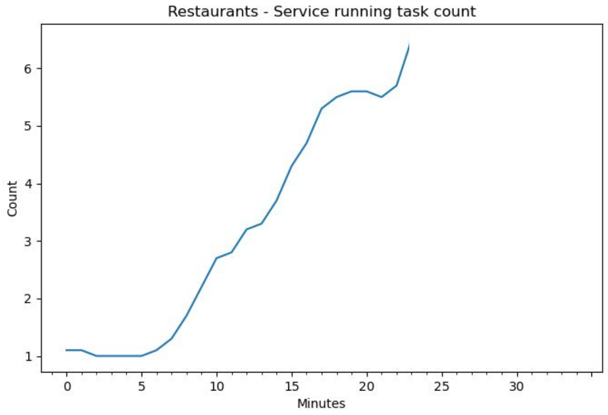


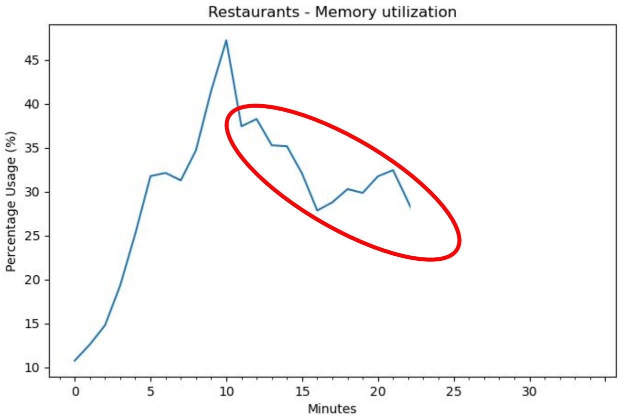
Restaurants - Memory utilization

Results of experiments - Restaurants

- There is a sudden increase with a peak of 80% of cpu utilization (target value was 60%)
- After the scale-out cooldown the scaling system starts working by adding new running tasks
- The usage percentage starts decreasing slowly

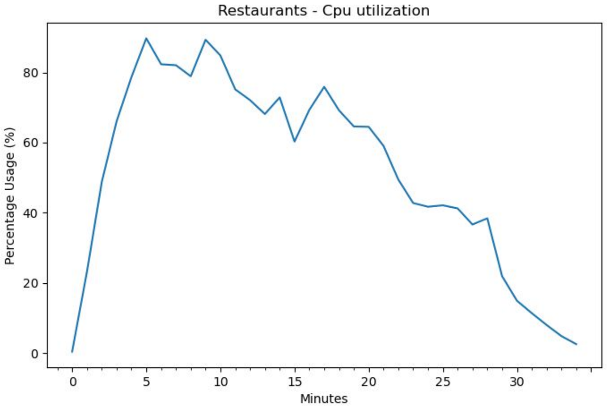


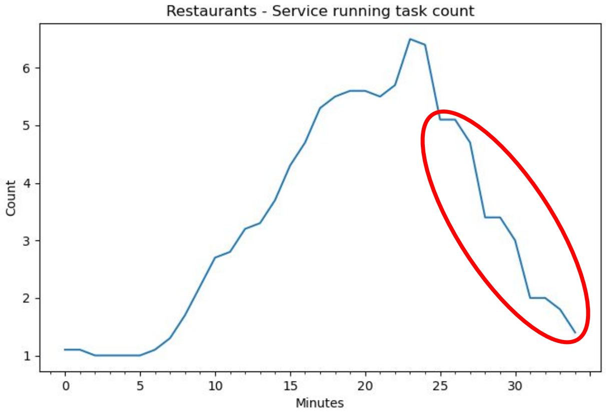


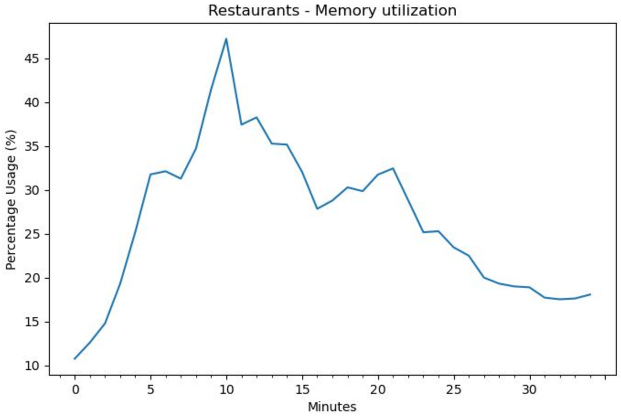


Results of experiments - Restaurants

- There is a sudden increase with a peak of 80% of cpu utilization (target value was 60%)
- After the scale-out cooldown the scaling system starts working by adding new running tasks
- The usage percentage starts decreasing slowly
- After the rump-down period also the running tasks number decreases


Restaurants - Cpu utilization


Restaurants - Service running task count


Restaurants - Memory utilization

Conclusions

Conclusions

- We built the application using the **microservices architecture**

Conclusions

- We built the application using the **microservices architecture**
- Taking advantages of **various benefits**
  - Independently deployable (we built and deployed each microservice in a parallel way)
  - Loosely coupled (each microservice is completely independent)
  - Organized by a small team (each member of the group focused on a specific service)

Conclusions

- We built the application using the **microservices architecture**
- Taking advantages of **various benefits**
  - Independently deployable (we built and deployed each microservice in a parallel way)
  - Loosely coupled (each microservice is completely independent)
  - Organized by a small team (each member of the group focused on a specific service)
- Each service was containerized using **Docker Compose**
  - Enabling fast deployment on **Amazon ECS**
  - Infrastructure managed by **Terraform**

Conclusions

- We built the application using the **microservices architecture**
- Taking advantages of **various benefits**
  - Independently deployable (we built and deployed each microservice in a parallel way)
  - Loosely coupled (each microservice is completely independent)
  - Organized by a small team (each member of the group focused on a specific service)
- Each service was containerized using **Docker Compose**
  - Enabling fast deployment on **Amazon ECS**
  - Infrastructure managed by **Terraform**
- Tests on the deployed application demonstrated the cloud's ability to manage workload efficiently (automatically adding and removing resources based on chosen policy)

Thank you for your attention