



**Ciências
ULisboa**

Faculdade
de Ciências
da Universidade
de Lisboa

Cloud Computing

Final Report

Threadit API

Group 8 (GitHub)

Ricardo Costa	64371
Eduardo Proença	57551
Leonardo Fernandes	64597
Manuel Gonçalves	58555

Prof. Mário Calha

MSc in Computer Science and Engineering
Faculty of Sciences of the University of Lisbon

2024/2025

June 2025

Contents

1	Introduction	5
1.1	Motivation	5
1.2	Datasets	5
1.3	Business Capabilities	5
1.4	Use Cases	6
2	API Specification	7
2.1	Community Service	7
2.2	Thread Service	8
2.3	Comment Service	9
2.4	Voting Service	10
2.5	Search Service	11
2.6	Popular Content Service	11
3	Functional and Non-Functional Requirements	13
3.1	Functional Requirements	13
3.1.1	Communities	13
3.1.2	Threads	13
3.1.3	Comments	13
3.1.4	Voting System	14
3.1.5	Search & Discovery	14
3.1.6	Microservices Communication	14
3.2	Non-Functional Requirements	15
3.2.1	API Gateway Configuration	15
3.2.2	Liveness and Readiness Probes	15
3.2.3	Resource Limits and Horizontal Pod Autoscaling (HPA)	15
3.2.4	Secret Management	15
3.2.5	CI/CD Pipeline	15
3.3	Deployment Plan	16
4	Architecture	17
4.1	Application Architecture	17
4.2	Technical Architecture	18
5	Implementation	19
5.1	Repository organization	19
5.2	Microservices	19
5.3	Technical Details	20
5.3.1	Service Communication	20
5.3.2	Server Implementation	20

5.3.3	gRPC gateway	20
5.3.4	API gateway	21
6	Scripts	22
6.1	Deployment Scripts	22
6.2	Other Scripts	23
7	CI/CD Pipeline	24
7.1	Overview	24
7.2	Pipeline Steps	24
7.2.1	Build and Test	24
7.2.2	Build, Publish and Deploy	24
8	Cloudy Day	26
8.1	Target Group	26
8.2	Test and Evaluation Techniques	26
8.2.1	Smoke Testing	26
8.2.2	Stress Testing	27
8.3	Results	27
8.4	Feedback	28
9	Corrections After Feedback	29
10	Cost Analysis	30
11	Conclusion	32

Chapter 1

Introduction

Threadit is cloud native application that offers a set of services that provide users the ability to connect, share and engage in discussions within communities through a REST API. Its architecture follows a microservices model with gRPC communication and is deployed on Google Cloud Platform (GCP) with Kubernetes.

1.1 Motivation

The motivation behind *Threadit* is to demonstrate the practical implementation of scalable, microservices-based architectures in cloud environments. With this project, we aim to explore key concepts of cloud computing and technologies like Kubernetes, gRPC and MongoDB, as well as Go, one of the most trending programming languages right now, which comes with native gRPC support.

1.2 Datasets

We populated our MongoDB database using the Reddit Top 2.5 Million dataset from 2013, which contains 1.66 GB of the most popular Reddit posts. However, this dataset didn't include comments and did not fill our implementation requirements.

Therefore, we partitioned the original dataset into two separate JSON files which were then imported into the MongoDB collections: one containing community metadata and another with post (thread) data, removing extra unnecessary fields to reduce storage overhead.

In the end, we created three distinct MongoDB collections:

- **Communities collection:** Populated with community metadata
- **Threads collection:** Populated with post (thread) data
- **Comments collection:** Initialized as an empty JSON array

1.3 Business Capabilities

The business capabilities of our application include:

- **Communities** - for threads centered around specific topics

- **Threads** - for discussing a single topic
- **Comments** - for engaging in conversations within threads
- **Voting System** - to upvote or downvote threads and comments
- **Search & Discovery** - to search for communities and/or threads
- **Popular Content** - to get popular threads and comments based on votes

1.4 Use Cases

- Communities
 - Create, get, update or delete a community
 - List all communities
- Threads
 - Create, get, update or delete a thread
 - List all threads
 - List threads from a community
- Comments
 - Create a comment on a thread or on another comment
 - Get, update or delete a comment
 - List all comments
 - List all comments from a thread
- Voting System
 - Upvote or downvote a thread
 - Upvote or downvote a comment
- Search & Discovery
 - Search for threads by name
 - Search for communities by name
 - Search for both by name
- Popular Content
 - Get popular threads by votes
 - Get popular comments by votes

Chapter 2

API Specification

2.1 Community Service

GET /communities

- Retrieves a list of communities. Supports optional filtering and pagination.
- Query Parameters:
 - **name** (string, optional): Filter communities by name.
 - **offset** (int32, optional): Number of items to skip (for pagination).
 - **limit** (int32, optional): Maximum number of communities to return.

POST /communities

- Creates a new community with the given name.
- Request Body:
 - **name** (string): Name of the new community.

GET /communities/{id}

- Retrieves details of a specific community by ID.
- Path Parameters:
 - **id** (string, required): ID of the community.

DELETE /communities/{id}

- Deletes a community by ID.
- Path Parameters:
 - **id** (string, required): ID of the community.

PATCH /communities/{id}

- Updates a community's name or thread count offset.
- Path Parameters:
 - `id` (string, required): ID of the community.
- Request Body:
 - `name` (string, optional): New name of the community.
 - `numThreadsOffset` (int32, optional): Change in number of threads.

2.2 Thread Service

GET /threads

- Retrieves a list of threads. Supports filtering and pagination.
- Query Parameters:
 - `communityId` (string, optional): Filter threads by community ID.
 - `title` (string, optional): Filter threads by title.
 - `offset` (int32, optional): Number of items to skip.
 - `limit` (int32, optional): Maximum number of threads to return.
 - `sortBy` (string, optional): Sorting criteria.

POST /threads

- Creates a new thread.
- Request Body:
 - `communityId` (string): ID of the community the thread belongs to.
 - `title` (string): Title of the thread.
 - `content` (string): Content of the thread.

GET /threads/{id}

- Retrieves details of a specific thread by ID.
- Path Parameters:
 - `id` (string, required): ID of the thread.

DELETE /threads/{id}

- Deletes a thread by ID.
- Path Parameters:
 - `id` (string, required): ID of the thread.

PATCH /threads/{id}

- Updates fields of a thread.
- Path Parameters:
 - `id` (string, required): ID of the thread.
- Request Body:
 - `title` (string, optional): New title.
 - `content` (string, optional): New content.
 - `voteOffset` (int32, optional): Change in up/down votes.
 - `numCommentsOffset` (int32, optional): Change in number of comments.

2.3 Comment Service

GET /comments

- Retrieve a list of comments filtered by optional query parameters.
- Query Parameters:
 - `threadId` (string, optional): Filter comments by thread ID.
 - `offset` (integer, optional): Pagination offset.
 - `limit` (integer, optional): Pagination limit.
 - `sortBy` (string, optional): Sort order or field.

POST /comments

- Create a new comment.
- Request Body:
 - `content` (string): The text content of the comment.
 - `parentId` (string, optional): The ID of the parent comment or thread.
 - `parentType` (enum: THREAD, COMMENT): Type of the parent entity.

GET /comments/{id}

- Retrieve a specific comment by its ID.
- Path Parameters:
 - `id` (string, required): The comment ID.

DELETE /comments/{id}

- Delete a specific comment by its ID.
- Path Parameters:
 - `id` (string, required): The comment ID.

PATCH /comments/{id}

- Update fields of a specific comment.
- Path Parameters:
 - `id` (string, required): The comment ID.
- Request Body:
 - `content` (string, optional): New content for the comment.
 - `voteOffset` (integer, optional): Change in up/down votes.
 - `numCommentsOffset` (integer, optional): Change in number of comments.

2.4 Voting Service

POST /votes/comment/{commentId}/down

- Downvote a comment by its ID.
- Path Parameters:
 - `commentId` (string, required): The ID of the comment to downvote.
- Request Body:
 - Empty object (no fields required).

POST /votes/comment/{commentId}/up

- Upvote a comment by its ID.
- Path Parameters:
 - `commentId` (string, required): The ID of the comment to upvote.
- Request Body:
 - Empty object (no fields required).

POST /votes/thread/{threadId}/down

- Downvote a thread by its ID.
- Path Parameters:
 - `threadId` (string, required): The ID of the thread to downvote.
- Request Body:
 - Empty object (no fields required).

POST /votes/thread/{threadId}/up

- Upvote a thread by its ID.
- Path Parameters:
 - **threadId** (string, required): The ID of the thread to upvote.
- Request Body:
 - Empty object (no fields required).

2.5 Search Service

GET /search

- Search across threads and communities globally.
- Query Parameters:
 - **query** (string, optional): Search keyword or phrase.
 - **offset** (integer, optional): Pagination offset.
 - **limit** (integer, optional): Maximum number of results to return.

GET /search/community

- Search for communities matching the query.
- Query Parameters:
 - **query** (string, optional): Search keyword or phrase for communities.
 - **offset** (integer, optional): Pagination offset.
 - **limit** (integer, optional): Maximum number of results to return.

GET /search/thread

- Search for threads matching the query.
- Query Parameters:
 - **query** (string, optional): Search keyword or phrase for threads.
 - **offset** (integer, optional): Pagination offset.
 - **limit** (integer, optional): Maximum number of results to return.

2.6 Popular Content Service

GET /popular/comments

- Retrieve a list of popular comments.

- Query Parameters:
 - `offset` (integer, optional): Pagination offset for the results.
 - `limit` (integer, optional): Maximum number of comments to return.

GET /popular/threads

- Retrieve a list of popular threads.
- Query Parameters:
 - `offset` (integer, optional): Pagination offset for the results.
 - `limit` (integer, optional): Maximum number of threads to return.

Chapter 3

Functional and Non-Functional Requirements

3.1 Functional Requirements

3.1.1 Communities

- The system allows to list all communities, optionally by name, offset and limit.
- The system allows to create a new community with a given name.
- The system allows to get a community by id, with its name and number of threads.
- The system allows to update a community's name or number of threads by id.
- The system allows to delete a community by id.

3.1.2 Threads

- The system allows to list all threads, optionally by community id, title, offset, limit and sort order.
- The system allows to create a new thread with a title and content in a specific community.
- The system allows to get a thread by id, with its title, content, community id, upvotes, downvotes and number of comments.
- The system allows to update a thread's title, content, votes or number of comments by id.
- The system allows to delete a thread by id.

3.1.3 Comments

- The system allows to list all comments, optionally by thread id, offset, limit and sort order.
- The system allows to create a new comment with content in a thread or another comment.

- The system allows to get a comment by id, with its content, upvotes, downvotes, parent id, parent type (thread or comment) and number of replies.
- The system allows to update a comment's content, votes or number of replies by id.
- The system allows to delete a comment by id.

3.1.4 Voting System

- The system allows to upvote or downvote a thread.
- The system allows to upvote or downvote a comment.
- The system allows to change or remove vote on a thread or comment.

3.1.5 Search & Discovery

- The system allows to search for threads by keyword.
- The system allows to search for communities by keyword.
- The system allows to search for both by keyword.
- The system supports pagination (offset and limit) for searches.
- The system supports sort order for searches.

3.1.6 Microservices Communication

- The system enforces that all inter-service communication between microservices uses gRPC.
- The system exposes a REST API for external clients to interact with the system.

3.2 Non-Functional Requirements

3.2.1 API Gateway Configuration

The application uses a Traefik instance installed and configured via Helm. An Ingress-Route and some Middlewares needed to be defined to ensure that Traefik routes the incoming requests to the gRPC Gateway.

3.2.2 Liveness and Readiness Probes

To improve fault tolerance and enable better self-healing behavior in Kubernetes, we defined:

- **Liveness probes** to detect and restart crashed pods.
- **Readiness probes** to ensure that traffic is only sent to pods that are ready to handle requests.

3.2.3 Resource Limits and Horizontal Pod Autoscaling (HPA)

We benchmarked the services to determine ideal values for:

- **CPU and memory resource requests/limits.**
- **Horizontal Pod Autoscaling (HPA)** thresholds based on real traffic patterns to ensure scalability.

3.2.4 Secret Management

To further improve security we explored Google Secret Manager for managing sensitive configuration data such as API keys, credentials and tokens. This approach provides:

- Centralized and secure secret storage.
- IAM-based access control.
- Seamless integration with GKE via workload identity.
- Versioning and audit logging for secret access.

3.2.5 CI/CD Pipeline

The Continuous Integration and Continuous Deployment (CI/CD) pipeline was implemented using GitHub Actions and GKE in order to automatically:

- Build and test images.
- Deploy services to the cluster.

3.3 Deployment Plan

Step	Tool/Technology
Containerization	Docker
Cluster Orchestration	Kubernetes (GKE)
Ingress Management	Traefik Ingress Controller
CI/CD	GitHub Actions
Autoscaling	Kubernetes HPA

Chapter 4

Architecture

4.1 Application Architecture

The application architecture is driven by the functional requirements and shows the services and databases of our system, including the interactions between them as well as the type of connections, whether REST or gRPC.

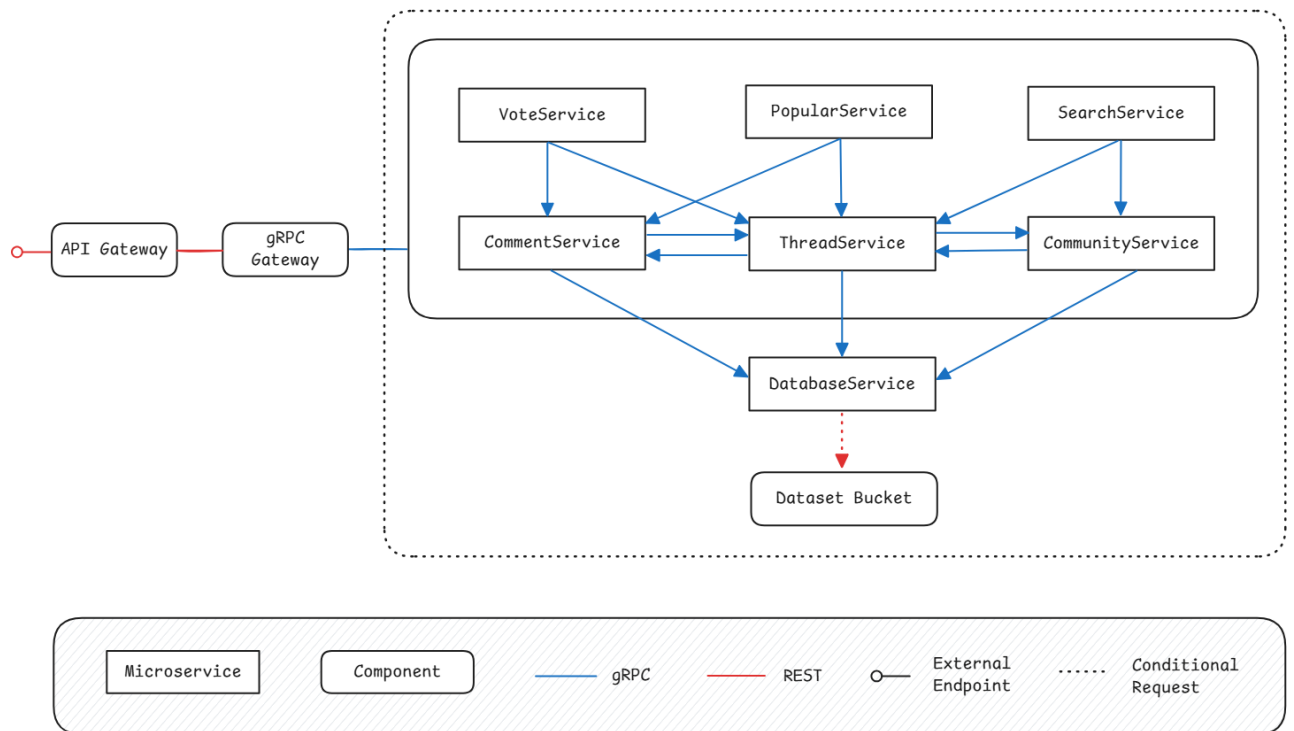


Figure 4.1: Application Architecture Diagram

4.2 Technical Architecture

The technical architecture is driven by the non-functional requirements and shows the cloud provider services and tools developed to support the fulfillment of those requirements.

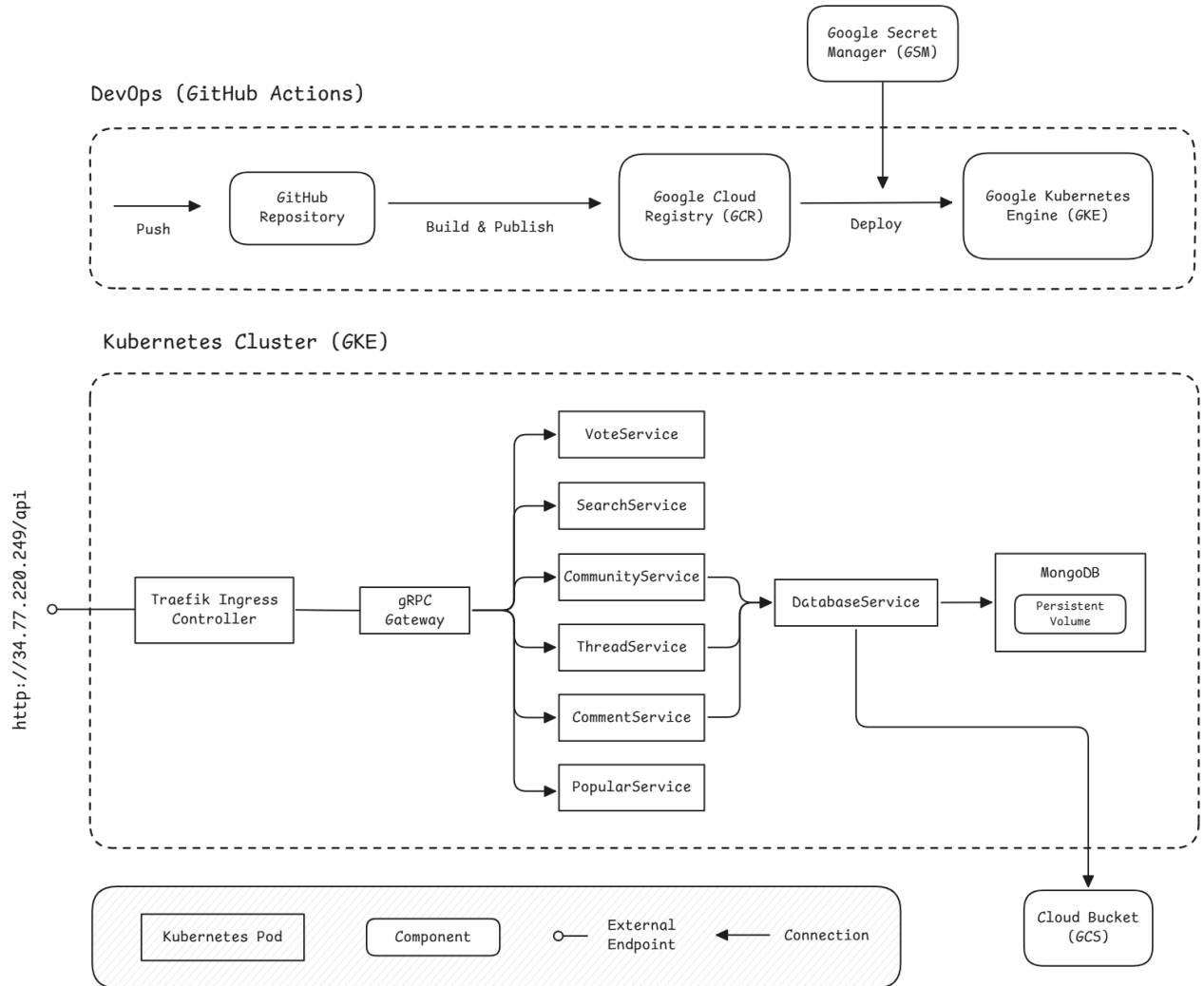


Figure 4.2: Technical Architecture Diagram

Chapter 5

Implementation

5.1 Repository organization

The implementation of the application was divided into different modules:

- **dataset** - the dataset (stored with Git LFS) used to populate the database
- **gen** - the generated code from the protobufs
- **grpc-gateway** - the implementation and configuration for the gRPC gateway used for exposing the gRPC services as a REST API
- **kubernetes** - the kubernetes deployment configuration
- **proto** - the protocol buffers used for gRPC communication between the microservices
- **services** - the microservices implementation that make up the backend of the system
- **traefik** - the configuration for the Traefik reverse proxy used for routing requests to the microservices (for local use only)

5.2 Microservices

The application was implemented as a set of microservices using Go and gRPC. The system contains the following microservices:

- **Database Service**
 - Handles all database operations using the MongoDB Driver API
 - **Depends on:** MongoDB database
- **Community Service**
 - Handles community operations
 - **Depends on:** Database service
- **Thread Service**
 - Handles thread operations
 - **Depends on:** Database and community services

- **Comment Service**
 - Handles comment operations
 - **Depends on:** Database and thread services
- **Vote Service**
 - Handles upvotes and downvotes
 - **Depends on:** Thread and comment services
- **Search Service**
 - Handles search operations
 - **Depends on:** Community and thread services
- **Popular Service**
 - Handles the retrieval of the most upvoted threads and comments
 - **Depends on:** Thread and comment services

5.3 Technical Details

5.3.1 Service Communication

All microservices use gRPC for inter-service communication and each one implements both client and server interfaces. The communication layer is implemented using a consistent pattern across all services, where each service maintains its own set of client connections to its dependencies. The communication is handled through generated protobuf code, which provides type-safe interfaces for all service interactions.

The service discovery mechanism is implemented through environment variables, where each service reads its dependencies' host and port information from the environment. This approach allows for flexible deployment configurations and easy service location in different environments. The connection establishment is handled through a common `connectGrpcClient` function that's shared across all services, ensuring consistent connection handling.

5.3.2 Server Implementation

Each service in the system follows a consistent server implementation pattern. The server implementation is organized into two main components: the main entry point (`main.go`) and the server implementation (`server.go`). The main entry point is responsible for setting up the service, establishing connections to dependencies and starting the gRPC server. The server implementation contains the actual business logic and handles the incoming requests with the injected dependencies.

5.3.3 gRPC gateway

We integrated the gRPC-Gateway Go package, which is a plugin of the Google protocol buffers compiler protoc. It reads protobuf service definitions and generates a reverse-proxy server which translates a RESTful HTTP API into gRPC. This server is generated according to the `google.api.http` annotations in your service definitions.

5.3.4 API gateway

In order to expose our RESTful HTTP API to the client, a Traefik to server as a ingress controller and load balancer to the application. It redirects the client requests to the gRPC-Gateway which then sends them to the appropriate microservice.

Chapter 6

Scripts

To ensure a streamlined and reproducible deployment process for our microservices architecture, several Bash scripts were developed. These scripts simplify tasks such as generating code from protocol buffers and OpenAPI specifications, managing Docker containers, and deploying services to a Kubernetes cluster hosted on Google Cloud Platform (GCP).

6.1 Deployment Scripts

create-cluster.sh

Provisions a Kubernetes cluster in GCP using the `gcloud` CLI. It automates the creation of the necessary infrastructure for deploying services.

deploy.sh

Used after the cluster has been created. It builds Docker images for all microservices, pushes them to the Google Cloud Registry (GCR) and applies the corresponding Kubernetes configuration files to deploy the services.

delete-cluster.sh

Deletes the Kubernetes cluster previously created in GCP. It is used to tear down the infrastructure and release associated resources when no longer needed.

cluster-info.sh

Displays information about the current state of the Kubernetes cluster, including pods, services, and deployments. It is useful for monitoring and debugging purposes.

update-image.sh

Performs a rolling update of one or more microservices in the Kubernetes cluster. The image tag must be specified via a parameter. To update all services, the `--all` flag must be

used. To update a specific microservice, its name should be provided as a parameter. The script leverages Kubernetes' built-in rollout mechanism.

rollback-image.sh

Rolls back the most recent image update for one or more microservices in the Kubernetes cluster. To rollback all services, the `--all` flag must be used. To rollback a specific microservice, its name should be provided as a parameter. The script leverages Kubernetes' built-in rollback mechanism.

6.2 Other Scripts

generate-proto.sh

Generates Go code from protobuf definitions. It ensures that the gRPC and protobuf stubs are up-to-date across the services.

generate-openapi.sh

Generates OpenAPI specifications from the existing protobuf definitions.

run.sh

Starts the project locally using Docker Compose. It brings up all the defined services as containers.

stop.sh

Stops and removes all containers associated with the project. It ensures a clean shutdown of services and removes associated resources.

Chapter 7

CI/CD Pipeline

7.1 Overview

Continuous Integration and Continuous Deployment (CI/CD) were employed in the application, through GitHub Actions to automate its building, testing and deployment, achieving a modern DevOps workflow.

With it, we were able to streamline the process from code commit to deployment, with the following automated tasks:

- **Building and Testing Docker Images:** containerizing each microservice for reliable and consistent deployment
- **Pushing Images to GCR:** storing versioned images for easy distribution and deployment in the Google Container Registry
- **Deploying to GKE:** rolling out updated services to the Kubernetes cluster with minimal downtime

7.2 Pipeline Steps

7.2.1 Build and Test

With each push or pull request to the `dev` branch of the repository, a GitHub Actions workflow is triggered to:

- Checkout the source code
- Build Docker images for each microservice using their respective Dockerfiles
- Run the unit tests in each microservice

7.2.2 Build, Publish and Deploy

Then, when the code is pushed to the `main` branch of the repository, a different workflow is triggered to:

- Checkout the source code
- Authenticate with Google Cloud

- Build and push the Docker images of each microservice to the Google Container Registry (GCR)
- Deploy or update Kubernetes resources for all microservices, Traefik, configuration maps and secrets

Chapter 8

Cloudy Day

8.1 Target Group

- **Group:** 11
- **Topic:** *LinkedIn Job Postings and Job Reviews*

8.2 Test and Evaluation Techniques

8.2.1 Smoke Testing

Methodology

To ensure the core functionalities of the system were operational, our group conducted smoke testing using **Postman**. Each team member was responsible for manually testing a set of critical API endpoints by sending HTTP requests and analyzing the responses. The primary goal was to verify that essential endpoints were accessible, returned appropriate status codes, and performed their intended operations without unexpected errors.

Tested Endpoints and Results

- GET `/jobs/search/average-salary` - Correct functionality and returning 200 OK
- GET `/jobs/search/remote` - Correct functionality and returning 200 OK
- GET `/jobs/search/best-cities` - Taking too long to respond and returning 500 Internal Server Error
- GET `/jobs/search/jobs-with-rating` - Correct functionality and returning 200 OK
- GET `/jobs/search/best-companies` - Taking too long to respond and returning 500 Internal Server Error
- GET `/jobs/search/jobs-in-biggest-companies` - Taking too long to respond and returning 500 Internal Server Error
- GET `/jobs/search/best-paying-companies` - Correct functionality and returning 200 OK

- POST /jobs/post/job-posting – Correct functionality and returning 201 CREATED
- POST /jobs/post/review-posting – Correct functionality and returning 200 OK
- PUT /jobs/put/review-update – Correct functionality and returning 200 OK
- DELETE /jobs/delete/review-deletion – Correct functionality and returning 200 OK

8.2.2 Stress Testing

Methodology

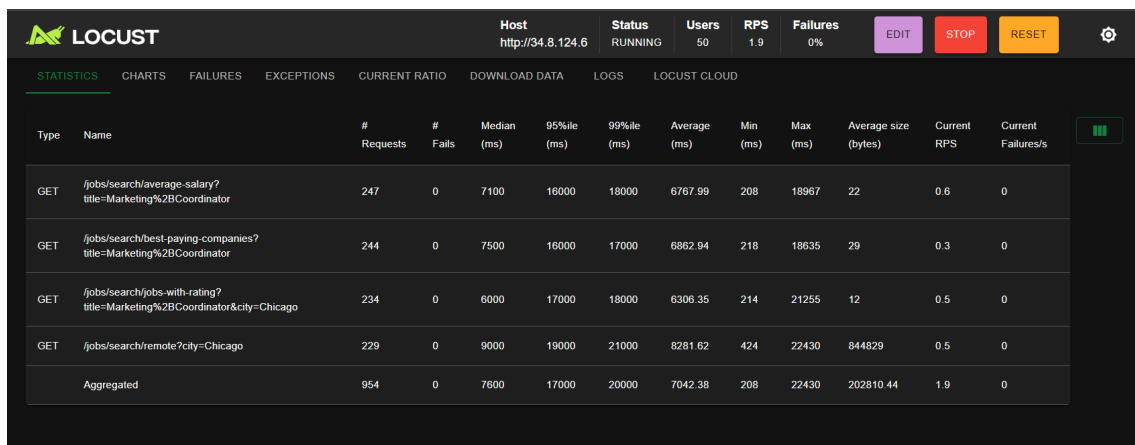
To evaluate the system's performance and stability under load, we employed **Locust**, an open-source load testing tool. We defined user behavior scripts to simulate realistic traffic patterns and executed tests by gradually increasing the number of concurrent users. This allowed us to measure the system's response times, throughput, and failure rates under various stress levels. The primary objective was to identify performance bottlenecks and ensure the tested endpoints could handle high request volumes without degradation of service.

Tested Endpoints

- GET /jobs/search/average-salary
- GET /jobs/search/remote
- GET /jobs/search/jobs-with-rating
- GET /jobs/search/best-paying-companies

8.3 Results

We established that the maximum amount of concurrent users was 50. When trying 60, failures started to arise, with internal server error status codes.



LOCUST		Host	Status	Users	RPS	Failures						
		http://34.8.124.6	RUNNING	50	1.9	0%	EDIT STOP RESET			LOCUST CLOUD		
Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/jobs/search/average-salary?title=Marketing%2BCoordinator	247	0	7100	16000	18000	6767.99	208	18967	22	0.6	0
GET	/jobs/search/best-paying-companies?title=Marketing%2BCoordinator	244	0	7500	16000	17000	6862.94	218	18635	29	0.3	0
GET	/jobs/search/jobs-with-rating?title=Marketing%2BCoordinator&city=Chicago	234	0	6000	17000	18000	6306.35	214	21255	12	0.5	0
GET	/jobs/search/remote?city=Chicago	229	0	9000	19000	21000	8281.62	424	22430	844829	0.5	0
Aggregated		954	0	7600	17000	20000	7042.38	208	22430	202810.44	1.9	0

Figure 8.1: Locust Web Interface

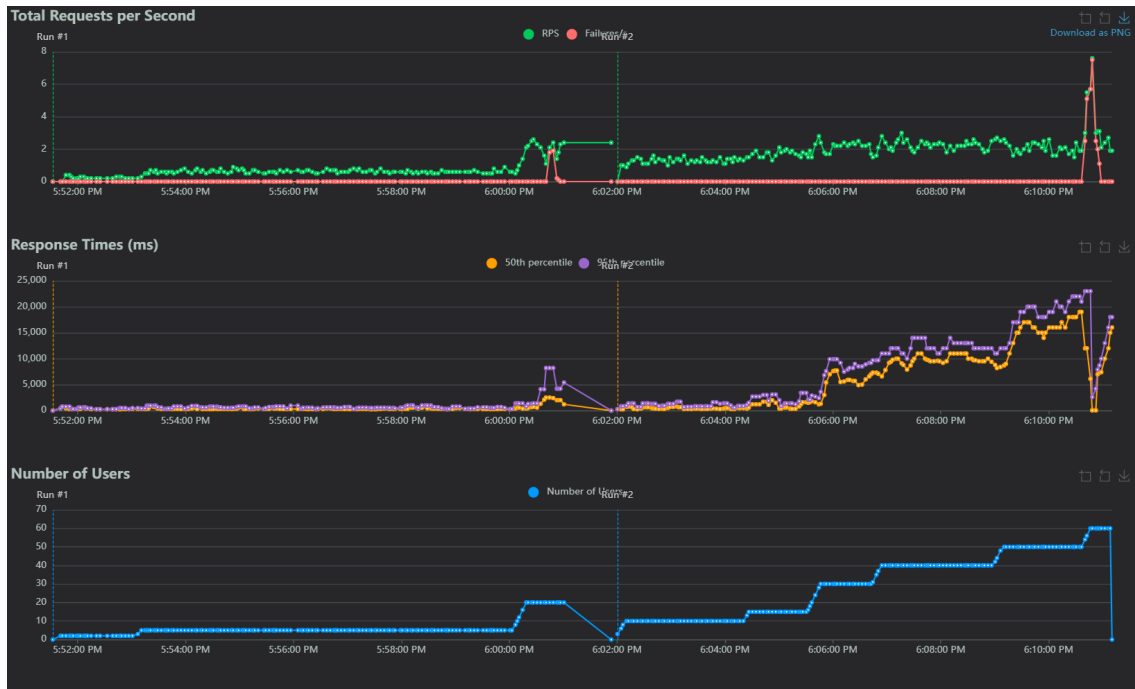


Figure 8.2: Locust Graphs

8.4 Feedback

- Endpoints that perform large queries of the database should have a default query limit.
- The POST requests should return the id of the object created.
- Better scaling configuration to allow for more concurrent users, such as HPA.
- Improve REST API endpoints:
 - Divide paths by domain, instead of all starting with `/jobs`
 - Include resource ids in the paths, not in the queries or bodies
 - Remove redundancy, by not including the HTTP method in the path itself
 - For example, in the delete review endpoint:
 - * Use `DELETE /reviews/{id}`
 - * Instead of `DELETE /jobs/delete/review-deletion?review_id={id}`

Chapter 9

Corrections After Feedback

The feedback given by the group was mainly about performance issues when the system is under load and about a data consistency bug.

The bug was that when a thread was deleted, the comments in that thread were not deleted. We fixed this, as it was a simple overlooked detail that we had forgotten about. Moreover, we found also that if the user deletes a comment which has other comments attached to it, those were also not deleted, which we also fixed to maintain data consistency.

Regarding the performance issue, we also detected this when stress testing our own application with Locust, as the response times began to rise significantly once the number of clients exceeded 500 users. Despite the cluster limitations to minimize expenses, we set up a robust configuration of resource limits and scaling (HPA), which in our point of view helped preventing what could have been a much worse performance scenario of our system. In order to try and boost the performance of our system, we also added an extra configuration in each microservice, to enable them to use the maximum number of CPUs available. However, with the limited testing performed, despite seeing some improvement, we do not have enough data to conclude that this has had a positive effect.

Finally, we believe that the performance issue reported by the group is due to our cluster limitations and mainly the bottleneck that occurs between the microservices and the database, which is something impossible for us to change at this stage of development. In spite of this, we consider that a number of 500 clients in a stress test environment is acceptable for the project requirements of this course.

Chapter 10

Cost Analysis

The cost for this project is divided, in the Google Cloud Platform, between the several tools used for its development - Compute Engine, Kubernetes Engine, Networking, Cloud Monitoring and Cloud Storage.

In total, between April and June our group spent \$18.52, split between two billing accounts (\$7.44 + \$11.08).

During the development of the project we tried to be as much cost effective as possible, achieving what we believe to be a very acceptable overall cost.

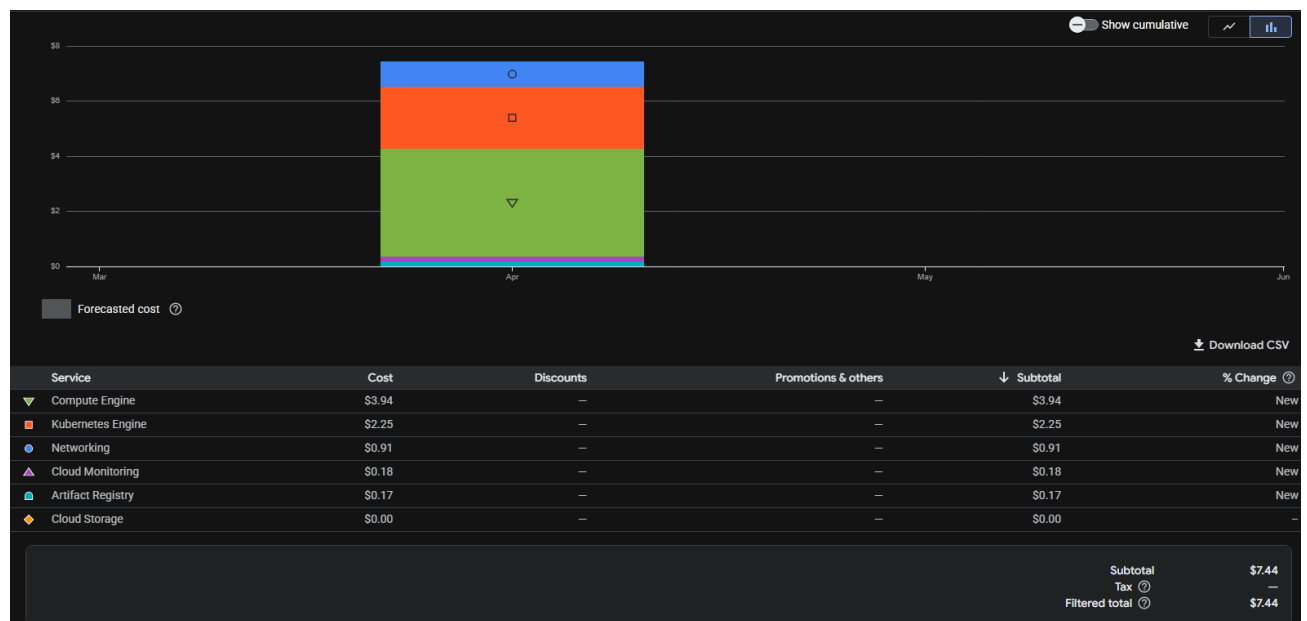


Figure 10.1: Project Cost (Eduardo's billing account)



Figure 10.2: Project Cost (Ricardo's billing account)

Chapter 11

Conclusion

Throughout the development of Threadit, we successfully designed and implemented a cloud-native application leveraging microservices, gRPC communication and Kubernetes orchestration. Our main objective was to gain some hands-on experience with cloud computing concepts, including scalable service architectures, CI/CD pipelines and containerized deployments in a managed cloud environment.

We adopted modern DevOps practices using GitHub Actions for continuous integration and Google Kubernetes Engine for deployment, which enabled a streamlined development lifecycle from code to production. The integration of OpenAPI and protobuf specifications further ensured consistency and maintainability across our services.

The most significant challenge we encountered was performance tuning and scalability. Ensuring the system could handle high traffic volumes with minimal latency required careful resource allocation, autoscaling configuration and microservice-level optimizations.

During the project, we also observed that operating a cloud-native architecture using managed services such as GKE and GCR can become expensive, particularly when dealing with multiple microservices and continuous deployments.

Overall, Threadit served as a practical and valuable case study in the application of distributed systems principles and cloud-native technologies, preparing us for real-world challenges in modern cloud-based application development.