ISEL
INSTITUTO SUPERIOR DE
ENGENHARIA DE LISBOA

Final Project:
Landmark Recognition System

Group 03

48280 André Jesus
48287 Nyckollas Brandão

Professor: José Simão

Report written for Cloud Computing
BSc in Computer Science and Computer Engineering

May 2023

Abstract

This report presents the development and implementation of a cloud-based system for landmark recognition. The system integrates various services from the Google Cloud Platform, including Cloud Storage, Firestore, Pub/Sub, Compute Engine, Cloud Functions, Vision API, and Static Maps API, to enable efficient storage, communication, and computation capabilities.

The primary objective of the system is to achieve elasticity by dynamically scaling the processing capacity of images to adapt to workload variations. It identifies landmarks and retrieves corresponding maps. Client applications can access the system through a gRPC interface.

The report provides an introduction to the system and outlines its design and architecture. It describes the integration of the Google Cloud Platform services and highlights their specific roles in enabling the system's capabilities. The implementation details, including the utilization of each service, are discussed to provide a comprehensive understanding of the system's functionality.

Challenges encountered during development are addressed, along with strategies employed to overcome them. The system's scalability and elasticity features are explored, demonstrating its ability to adapt to varying workloads.

Overall, this work demonstrates the successful design and implementation of an elastic cloud computing system using Google Cloud Platform services for landmark recognition, providing a scalable solution for cloud-based image processing.

Key Words: Cloud Computing; Scalability; Google Cloud Platform; Landmark recognition

Resumo

Este relatório apresenta o desenvolvimento e implementação de um sistema baseado na nuvem para reconhecimento de marcos históricos. O sistema integra vários serviços da plataforma Google Cloud, incluindo Cloud Storage, Firestore, Pub/Sub, Compute Engine, Cloud Functions, Vision API e Static Maps API, para permitir eficiente armazenamento, comunicação e computação.

O principal objetivo do sistema é alcançar elasticidade por meio do dimensionamento dinâmico da capacidade de processamento de imagens, adaptando-se a variações na carga de trabalho. Identifica marcos históricos e procura mapas correspondentes. Aplicações cliente podem aceder ao sistema através de uma interface gRPC.

O relatório apresenta uma introdução ao sistema, sublinhando o seu design e arquitetura. Descreve a integração dos serviços da plataforma Google Cloud e destaca os seus papéis específicos nas funcionalidades do sistema. Os detalhes de implementação, incluindo a utilização de cada serviço, são discutidos, fornecendo uma compreensão abrangente da funcionalidade do sistema.

Desafios encontrados durante o desenvolvimento são abordados, juntamente com as estratégias empregadas para superá-los. As características de escalabilidade e elasticidade do sistema são exploradas, demonstrando a sua capacidade de adaptação a cargas de trabalho variáveis.

No geral, este trabalho demonstra o design e implementação bem-sucedidos de um sistema de computação na nuvem utilizando os serviços da plataforma Google Cloud para reconhecimento de marcos históricos, fornecendo uma solução escalável para processamento de imagens baseado na nuvem.

Palavras-chave: Computação na Nuvem; Escalabilidade; Google Cloud Platform; Reconhecimento de Marcos Históricos.

Contents

List of Figures

List of Acronyms

API	Application Programming Interface
AWS	Amazon Web Services
GCP	Google Cloud Platform
gRPC	Google Remote Procedure Call
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
RPC	Remote Procedure Call
REST	REpresentational State Transfer
URL	Uniform Resource Locator
VM	Virtual Machine

Chapter 1

Introduction

The rapid growth of cloud computing [1] has revolutionized the way businesses and organizations leverage computational resources and services. Cloud platforms provide scalable and flexible infrastructures that enable efficient storage, communication, and computation capabilities.

Cloud-based systems offer a compelling solution for efficient image processing due to their scalability and elasticity. By leveraging cloud platforms, the computational resources can be dynamically allocated based on workload variations, resulting in optimized performance and cost-effectiveness. The use of cloud platforms also allows for the integration of various services and APIs that can enhance the functionality and capabilities of the system.

This report presents the development and implementation of a cloud-based system for landmark recognition, leveraging various services offered by the Google Cloud Platform (GCP) [2]. The system aims to address the challenges associated with processing image data at scale and achieving elasticity to adapt to varying workloads. By utilizing the capabilities of GCP, including Cloud Storage [3], Firestore [4], Pub/Sub [5], Compute Engine [6], Cloud Functions [7], Vision API [8], and Static Maps API [9], the system provides a scalable solution for cloud-based image processing.

A gRPC [10] interface was also defined, in order to implement a gRPC server that acts as a facade to the backend system by communicating to the client only what's required of the gRPC contract and hiding the inner implementation of the several services that constitute the server.

1.1 Outline

The report is structured as follows.

Chapter 2 describes the domain of our problem in more depth and our approach to solve it.

Chapter 3 presents the architecture of system, namely its components and how the goals pointed out in chapter 2 will be reached with this application.

Chapter 4 describes the implementation details about each of the components that compose the system, their functionalities and their interactions.

Chapter 5 concludes the report, summarizing the key findings, challenges overcome and contributions of this work.

By following the structure of the report, readers will gain a comprehensive understanding of the system's development and implementation.

Chapter 2

Problem Description

The development of a cloud-based system for landmark recognition has several significant implications. Firstly, it enables efficient and accurate recognition of landmarks from large volumes of image data. This has applications in fields such as tourism, where travelers can use mobile applications to instantly identify landmarks and access relevant information. Furthermore, it can aid in cultural heritage preservation by automating the identification and documentation of historical landmarks.

Secondly, the scalability and elasticity features of the system allow for efficient resource utilization. The processing capacity can be dynamically scaled up or down based on the workload, ensuring optimal performance and cost-effectiveness. This scalability is particularly important when dealing with large datasets and fluctuating demand.

Thirdly, the integration of Google Cloud Platform services provides a comprehensive and robust framework for the system's implementation. GCP offers a wide range of services, such as Cloud Storage, Firestore, and Vision API, that can be leveraged to handle different aspects of the system's functionality. Understanding the integration and utilization of these services contributes to the broader knowledge and utilization of cloud computing technologies.

2.1 Problem and Solution

The problem addressed in this work is the development and implementation of a cloud-based system for landmark recognition that achieves elasticity and scalability. The primary objective is to design and build a system that can efficiently process large volumes of image data, identify landmarks, and retrieve corresponding maps. The system should adapt to varying workloads by dynamically scaling the processing capacity.

In order to fulfil this objective, we have developed the *Landmark Recognition System.*

2.1.1 Functional Requirements

Functional requirements state what the system must do, and how it must behave or react to. The functional requirements that were identified for the Landmark Recognition System are as follows:

- Design a gRPC interface, that allows the client to interact with the system;
- Develop a gRPC console client;
- Develop a gRPC server;
- Develop an application that performs landmarks detection;
- Integrate various services from the Google Cloud Platform to enable efficient storage, communication, and computation capabilities.

2.1.2 Non-Functional Requirements

Non-functional requirements or quality attributes requirements are qualifications of the functional requirements or of the overall product. The non-functional requirements that were identified for the Landmark Recognition System are as follows:

- Performance: The system should be capable of efficiently processing large volumes of image data and providing timely responses to client requests.
- Reliability: The system should be highly reliable, ensuring minimal downtime and data loss.
- Scalability: The system should be able to scale up or down based on workload variations, optimizing resource utilization.
- Maintainability: The system should be designed and implemented in a modular and maintainable manner, allowing for easy updates and enhancements.

2.1.3 Use Cases

With the requirements listed above, we have identified the use cases that the platform shall support. A use case is a written description of how users will perform tasks on a system. It outlines, from the user perspective, the behaviour of the system as it responds to a request. The use cases identified shall allow to reason about who are the users of the application, their objectives, the actions they are able to perform, and how the platform shall respond to each action.

The system use cases are presented in Figure 2.1.

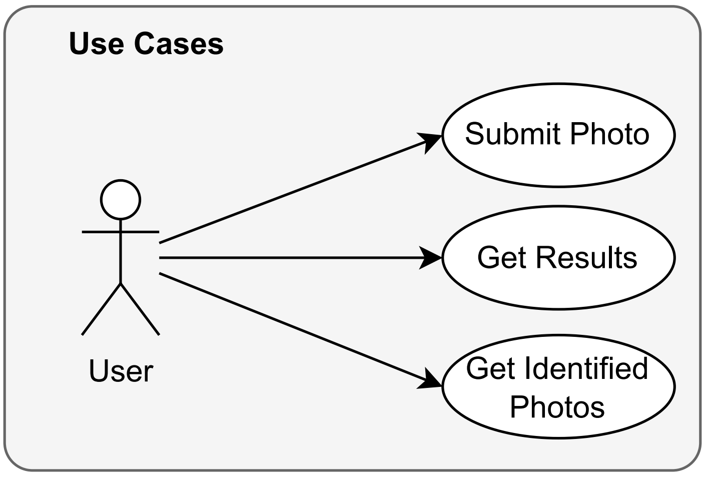

Figure 2.1: System use cases.

The following is a description of each use case:

1. Submit Photo: The client submits a photo (image file) for landmark detection. The system receives the photo, stores it as a blob in the Cloud Storage, and returns a unique identifier for the submission.

2. Get Results: The client requests the results for a specific identification request using the identifier received in the previous step. The system returns a list of identified landmarks, their geographic location (latitude and longitude in decimal format), and the associated confidence level. A static map image of one of the identified locations is also returned to the user. We made the decision to send the map of the identified landmark with a higher confidence level.

3. Get Identified Photos: The client requests the names of all photos where monuments were identified with a confidence level above a certain threshold (e.g., above 0.6) and the corresponding name of the identified location.

Please note that this use case diagram provides a high-level overview of the interactions between the actors and the system. It does not include detailed steps or data flows for each use case. Typically, use case diagrams are accompanied by textual descriptions or activity diagrams to provide a more comprehensive understanding of the system's behavior.

2.2 Summary

This chapter introduced the context and motivation for the development of a cloud-based system for landmark recognition. The significance of the system was discussed, highlighting its implications in various domains, such as tourism and cultural heritage preservation. The research problem and objectives were defined, focusing on achieving elasticity and scalability in image processing. The scope and structure of the report were outlined, providing a roadmap for readers to navigate through the subsequent chapters.

Chapter 3

Architecture

In this chapter, we will explain the components of the system and how they interact. This chapter will also clarify what the project can accomplish, as well as the architecture, entities, and implementation blueprint that we have designed and developed for it.

Figure 3.1 is a diagram that shows the system's main components and their interactions.

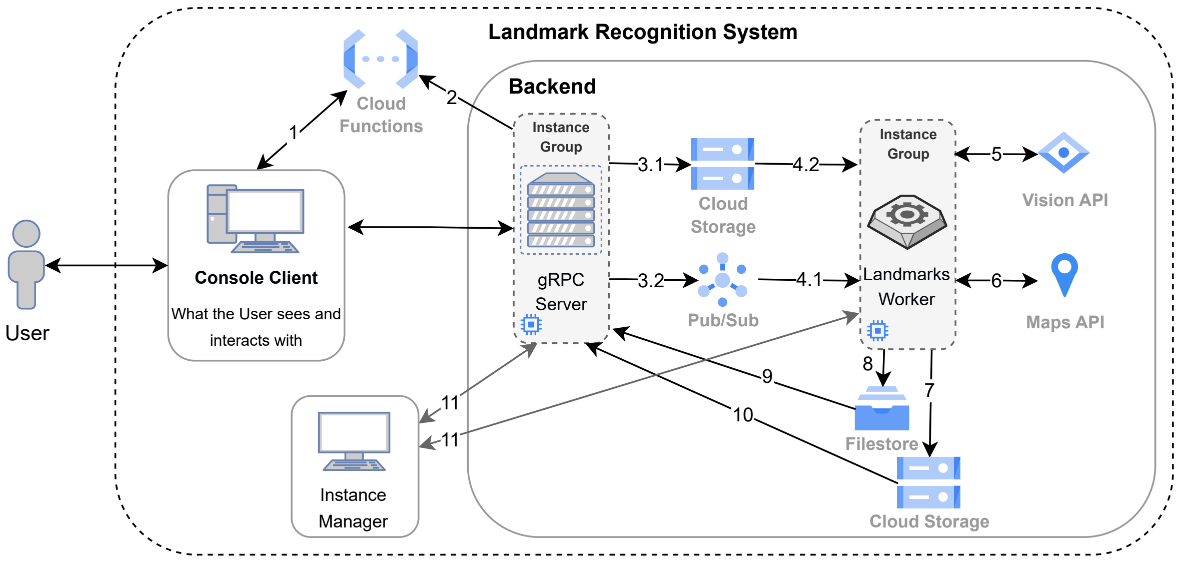

Figure 3.1: System architecture.

3.1 Operations Flow

Considering the sequence numbers of actions presented in Figure 3.1, the following list describes each functionality:

- **Lookup Function Service:** Used by the client application (1 and 2) to obtain the IP addresses of the gRPC servers, this service should be developed as a Cloud

Function. It retrieves (2) the IP addresses of the VMs that are part of the instance group. The client application randomly selects an IP from the returned list by the function. In case of a connection failure to the chosen gRPC server IP, the client application will attempt another IP or repeat the lookup process to update the IP list and establish a new connection.

- **Photo Submission:** After submitting a photo, it is stored in the Cloud Storage (3.1), and a unique identifier is returned to the gRPC client for future queries. Subsequently, a message containing the request identifier, timestamp, bucket name, blob name and photo name, is sent to a Pub/Sub topic (3.2) for landmark detection processing.

- **Worker and Landmarks App:** A shared subscription exists for the aforementioned topic, enabling multiple workers to process messages (work-queue pattern). For each message, a Landmarks App worker responsible for photo analysis receives the bucket and blob names of the photo to be processed (4.1). This allows obtaining a global reference (gs:// URI) to the Cloud Storage (4.2). The worker then interacts with the Vision API service (5) for location identification and the map service (6) for obtaining static maps.

- **Result Storage:** After the photo processing, the identified location maps are stored in the Cloud Storage (7), and relevant information regarding the request and analysis results is saved in Firestore (8).

- **Client Application Interaction:** The client application can request information about the submitted photos at any time using the request descriptor, as described in Section 1. To retrieve this information, the gRPC server consults Firestore (9) and/or the Cloud Storage (10).

- **Instance Groups Management:** A simple console application is responsible for the scaling of the instance groups (11).

3.2 GCP Services and APIs

The system architecture leverages several GCP services and the Google Static Maps API to fulfill its functionalities:

- **Cloud Storage:** This service is used to store the photos to be processed as well as the static maps.

- **Firestore:** Firestore is employed to store relevant information about photo processing, including request identifiers, references to stored blobs in Cloud Storage, names, and locations of identified landmarks in the photos, and any other pertinent data.
- **Pub/Sub:** Pub/Sub facilitates decoupled message exchange between the system components, enabling asynchronous communication.
- **Compute Engine:** The Compute Engine service is utilized to host the VMs where the gRPC server replicas and Landmarks App replicas, responsible for monument identification in photos, are executed.
- **Vision API:** The Vision API is employed for monument detection (landmarks) in the submitted photos.
- **Static Maps API:** The Static Maps API is used to obtain map images at specific latitude and longitude coordinates.

The deployment architecture comprises multiple gRPC server replicas running within VMs, instance groups for load balancing, and separate VMs for the Landmarks App replicas. The Cloud Storage and Firestore services store and manage the relevant data, while Pub/Sub enables communication among the system components. The Vision API performs monument detection, and the Static Maps API provides static map images.

This architecture ensures scalability, availability, and fault tolerance by leveraging the capabilities of the GCP services and distributing the system's components across VM instances.

3.3 Data Model

The data in our system is categorized into two categories: metadata and data. This is observable in Figure 3.1 with the Firestore and Cloud Storage services. Metadata describes the properties and relationships of the data, while data contains the actual data. In this chapter, we will explain all the entities that make up our data model, and then how they are stored and connected to each other by the metadata-data system.

Photos and Maps

The Cloud Storage service securely stores both the photos awaiting processing and the static maps. These visual assets, in the form of blobs, are stored in separate buckets. To ensure organizational clarity, we have established dedicated buckets for each resource. One bucket is designated for user-uploaded photos, while another is dedicated to static maps linked to identified landmarks.

Requests

The photos and maps mentioned above are directly tied to a landmark detection request initiated by the server and processed by the worker. To manage these requests efficiently, we utilize the Firestore service, which includes a dedicated collection specifically designed for this purpose. Within this collection, each document represents an individual landmark detection request. These documents comprehensively capture all pertinent information related to the request, such as the user-uploaded photo's name and URL, the request's status and timestamp, as well as a detailed list of the identified landmarks.

The Figure 3.2 shows an example of a document with a landmark detection request.

```
requestId: "7f89afcd-b092-4780-9802-5d5a6d61886c"
photoName: "hello"
photoUrl: "gs://landmarks-photos/05f31584-52ec-451e-b8eb-d172ab110485"
status: "DONE"
timestamp: May 22, 2023 at 12:21:37PM UTC+1
landmarks:
    0:
        name: "Belém Tower"
        confidence: 0.7491545677185059
        location:
            latitude: 38.691583699999995
            longitude: -9.215977299999999
        mapBlobName: "6928d7ba-400d-4730-8acb-f041d5bb3f9d"
    1:
        name: "Belém Tower Garden"
        confidence: 0.6960328221321106
        location:
            latitude: 38.6927205
            longitude: -9.215710399999997
        mapBlobName: "71efcd07-1190-4f03-a562-22340cae9a9b"
```

Figure 3.2: Example of a document representing a landmark detection request.

As shown, the request consists of a photo titled "hello" along with its corresponding URL. It includes an ID, a status of "DONE", indicating the completion of the landmark detection process, a timestamp, and an array of detected landmarks. The photo in question captures the magnificent Belém Tower (Torre de Belém), and the system successfully identified two landmarks associated with the photo: the Belém Tower itself and its picturesque garden.

Chapter 4

Implementation

In this chapter we describe in more detail each component of the system, how they interact, their functionalities, the technologies that support them and implementation details.

For the sake of decoupling and modularity, in *Landmarks App* and the *gRPC Server*, we have built abstractions between the different components of the systems, using a layered architecture pattern [11, 12]. Components within this architecture pattern are organized into horizontal layers, each layer performing a specific role within the application (e.g., presentation logic or business logic). Although the layered architecture pattern does not specify the number and types of layers that must exist in the pattern, most layered architectures consist of four standard layers: presentation, business, persistence, and database. The Landmarks App and gRPC Server sections below will describe these abstractions and their benefits in more detail.

4.1 Components

4.1.1 gRPC Contract

The gRPC [10] contract defines the structure and operations available in the Landmarks service. The contract is defined using Protocol Buffers (protobuf) [13] with syntax version 3 (proto3). The protobuf definition of the contract is available in Appendix A.

The gRPC contract includes a service called `LandmarksService`, which provides three methods: `submitPhoto`, `getResults`, and `getIdentifiedPhotos`.

submitPhoto

The `submitPhotoRequest` message is used for submitting a photo, with the photo being represented as a byte array (`bytes`). In every request of the stream, along with the bytes, the name of the photo is sent (`photo_name`).

The `submitPhotoResponse` message contains a request ID (`request_id`) as a string.

getResults

The `GetResultsRequest` message is used to request the results for a submitted photo using the corresponding request ID.

The `GetResultsResponse` message includes a list of `Landmark` messages, representing the identified landmarks, and an `ImageMap` message, which contains the image data for a static map.

The `Landmark` message represents a landmark or famous location identified in a photo, with properties such as the name (`name`), latitude (`latitude`), longitude (`longitude`), and confidence (`confidence`).

The `ImageMap` message represents an image of a static map for a specific location, with the image data stored as a byte array (`image_data`). The returned image map could be the one for the landmark with the highest confidence among all landmarks identified in the photo.

getIdentifiedPhotos

The `GetIdentifiedPhotosRequest` message is used to request the names of photos where a landmark was identified with a confidence level above a specified threshold (`confidence_threshold`). The `GetIdentifiedPhotosResponse` message contains a list of `IdentifiedPhoto` messages, representing the identified photos and their corresponding landmarks.

Each `IdentifiedPhoto` message includes the name of the photo (`photo_name`), the name of an identified landmark (`landmark_name`) and the associated confidence (`confidence`). The returned landmark could be the one with the highest confidence among all landmarks identified in the photo.

This gRPC contract defines the structure and operations that the client and server must adhere to in order to communicate effectively in the Landmarks service.

4.1.2 gRPC Client

The gRPC Client is the application the user interacts with directly. The use of gRPC hides the inner workings of the backend solutions of the server, providing the client application with only the contract to which it must adhere to.

The client provides a console interface with a menu shown in the Figure 4.1.

```
########## Landmark Recognition System ##########
 0: Submit Photo
 1: Get Results
 2: Get Identified Photos
 3: Exit
Enter an Option:
```

Figure 4.1: gRPC Client console menu.

The IP of the server to connect to is retrieved using an IP lookup Cloud Function that returns multiple server IPs. The client connects to one of these IPs at random, and tries another one if connection failed.

Two stubs are set-up for the `LandmarksService` of the gRPC contract, a blocking and a non-blocking stub. The blocking stub blocks the thread waiting for the response, while the non-blocking handles the response asynchronously using a stream observer.

The implemented gRPC client is accessible through a terminal and shows a menu to the user. The menu is composed of the following options, each one directly related to the methods of the `LandmarksService` of the gRPC contract:

- Submit Photo - Submit a photo from their computer and receives a request ID.
- Get Results - Receive the results associated to the request ID.
- Get Identified Photos - Receive information of photos with identified landmarks within a confidence threshold provided by the user.
- Exit - Exit the client application.

Submit Photo

This option calls the method `submitPhoto`, which performs the following steps:

- Asks for user input for the name of the photo and the path of the photo file.
- Verifies that the file exists in the provided path.
- Creates a `submitPhotoResponseObserver` for handling the server response.
- Calls the method `submitPhoto` of the non-blocking stub with the observer, which returns the stream observer from the server, `requestObserver`.
- Sends the photo as a stream of bytes, by calling successive `requestObserver.onNext` with a `submitPhotoRequest` object containing the name of the photo and the current block of bytes.

- Calls `requestObserver.onCompleted` to signal the end of the stream to the server.

The `submitPhotoResponseObserver` receives the `submitPhotoResponse` from the server via the method `onNext`, which is invoked by the server when it acknowledges the photo upload is complete, displaying the request ID to the user.

Get Results

This option calls the method `getResults`, which performs the following steps:

- Asks for user input for the request ID.
- Calls the method `getResults` of the blocking stub with a `GetResultsRequest` object containing the request ID, returning a `GetResultsResponse`.
- If no landmarks were identified, tells the user.
- Otherwise, displays the identified landmarks to the user:
  - Landmark Name
  - Latitude
  - Longitude
  - Confidence
- Creates an image file with the received map in the user's computer.

Get Identified Photos

This option calls the method `getIdentifiedPhotos`, which performs the following steps:

- Asks for user input for the confidence threshold.
- Verifies that the confidence threshold is between 0 and 1.
- Calls the method `getIdentifiedPhotos` of the blocking stub with a `GetIdentifiedPhotosRequest` object containing the confidence threshold, returning a `GetIdentifiedPhotosResponse`.
- If no identified photos were found within the confidence threshold, tells the user.
- Otherwise, displays the information of the photos to the user:
  - Photo Name

  - Landmark Name
  - Confidence

4.1.3 gRPC Server

The gRPC Server interacts with the gRPC client, acting as a facade to the backend system by communicating to the client only what's required of the gRPC contract and hiding the inner implementation of the several services that constitute the server.

This way, the client does not need to know about the details and complexity of the server's services, and can rely on a consistent and simple interface. To further reinforce the idea of the client-server facade, several abstractions were created to decouple the services used in the server, showing that the server can change its implementation without affecting the client.

The server's entry point is the `Main` class, which is responsible for creating the object instances used for dependency injection and starting the gRPC server with the `LandmarksServer` as its service. Configuration for the server like the names of cloud storage buckets can be done in the `Config` class.

LandmarksServer

The `LandmarksServer` class extends the base implementation of the `LandmarksService` of the gRPC contract. It is composed of each one of the three methods of the service: `submitPhoto`, `getResults` and `getIdentifiedPhotos`.

In this layered architecture, the `LandmarksServer` class serves as the controller using gRPC, while the `Service` class provides the business logic methods to be used by the controller, hiding the implementation of these. The service methods throw checked exceptions to be handled appropriately by the controller, which in this case may send an error to the client with a certain gRPC status code [14].

The `submitPhoto` method creates a new `PhotoRequestStreamObserver`, with the client's stream observer in the constructor, and returns it to the client.

The `PhotoRequestStreamObserver` is a stream observer of `submitPhotoRequest`, and is responsible for handling the bytes the client sends. With each call to `onNext`, the received bytes are written to the `ByteArrayOutputStream` created for the purpose of storing the photo. When `onCompleted` is called:

- a request ID is generated with a random UUID.
- `responseObserver.onNext` is called with a `submitPhotoResponse` containing the request ID.

- the photo bytes are submitted along with request ID and the name of the photo.

The `getResults` method receives a `GetResultsRequest` with the request ID and performs the following steps:

- Gets the results of the request associated with the request ID.
- If the request does not exist, sends an error to the client with `Status.NOT_FOUND`.
- If the request has still not finished processing, sends an error to the client with `Status.INVALID_ARGUMENT`.
- Otherwise, sends to the client a `GetResultsResponse` containing the list of identified landmarks in the photo as `Landmark` objects, and, if any landmarks were identified, an `ImageMap` with the image of the map of the landmark with the highest confidence.

The `getIdentifiedPhotos` method receives a `GetIdentifiedPhotosRequest` with the confidence threshold and performs the following steps:

- If the confidence threshold is not between 0 and 1, sends an error to the client with `Status.INVALID_ARGUMENT`.
- Otherwise, sends to the client a `GetIdentifiedPhotosResponse` with a list of `IdentifiedPhoto`, each containing information of a photo where at least one landmark was identified with a confidence within the threshold.

Service

For the business logic layer, the `Service` class was implemented. It provides methods to be used by the controller, communicating with the cloud data storage, metadata storage, and the landmarks detector without exposing the implementation to the controller. It exposes three public methods, each corresponding to a `LandmarksServer`'s method: `submitPhoto`, `getResults` and `getIdentifiedPhotos`.

This class uses a `CloudDataStorage` for handling cloud data, a `MetadataStorage` for handling request metadata, and a `LandmarksDetector` to notify about the request (start the detection). These three are **interfaces** for abstraction purposes. For instance, `LandmarksDetector`, instead of notifying an external service through pub/sub, could have a implementation for the asynchronous detection of the landmarks, being locally in the server.

In this implementation:

- To notify the (external) landmarks detector (*Landmarks App*), Google Pub/Sub is used. `PubSubLandmarksDetector` creates a temporary publisher for a topic and publishes the message.
- For the cloud storage, Google Cloud Storage is used. `GoogleCloudDataStorage` implements the cloud operations for uploading/downloading of blobs in buckets.
- For the metadata storage, Firestore is used. `FirestoreMetadataStorage` implements the firestore database queries for the request metadata document.

The `submitPhoto` method receives a request ID, the bytes of the photo and the name of the photo and performs the following steps:

- Generates a blob name with a random UUID.
- Uploads the photo to the cloud storage.
- Make the photo blob public.
- Notify the landmarks detector about the request, being provided the request ID, name of the photo and blob name.
- If at any point an exception occurs, an `PhotoSubmissionException` is thrown.

The `getResults` method receives a request ID and performs the following steps:

- Obtains the metadata of the request of request ID.
- If the request does not exist, throw a `RequestNotFoundException`.
- If the request has still not finished processing, throw a `RequestNotProcessedException`.
- Otherwise, return the list of identified landmarks and the image of the map of the landmark with the highest confidence.
- If the map image could not be retrieved, throw a `MapImageRetrievalException`.

The `getIdentifiedPhotos` method receives a confidence threshold and performs the following steps:

- If the confidence threshold is not between 0 and 1, throw an `InvalidConfidenceThresholdExcep`
- Otherwise, gets all request metadata and filters only those containing at least a landmark with the confidence within the threshold.
- Returns the list of identified photos.

4.1.4 Landmarks App

The Landmarks App interacts with the gRPC Server through Pub/Sub messages and is entirely dependent on the existence of the server, writing data to the databases that are accessible by the server.

The app's entry point is the `Main` class, which is responsible for creating the object instances used for dependency injection and starting the `LandmarksApp`. Configuration for the app like the names of cloud storage buckets can be done in the `Config` class.

LandmarksApp

The `LandmarksApp` class is composed of a method `start` which starts the app. It has as dependencies:

- `StorageService` for handling storage of photos and map images and request metadata.
- `LandmarksGooglePubsubService` for handling pub/sub messages.
- `LandmarkDetectionService` for detecting landmarks in a photo.
- `MapsService` for retrieving a map image based on location.

`LandmarkDetectionService` and `MapsService` are **interfaces** for abstraction purposes. For instance, for the detection of landmarks in a photo, another service could be used instead of Google Vision API. This is useful if it's considered that another service does the job better. As such, it's good to not be dependent on a specific implementation. In this implementation:

- For the landmark detection, Google Vision API is used. `GoogleVisionLandmarkDetectionService` implements the `detectLandmarks` operation, allowing both an URL and a byte array as possible inputs.
- For the map image retrieval, Google Maps Static API is used. `GoogleMapsStaticService` implements the `getStaticMap` operation.

The `start` method subscribes to the landmarks pub/sub subscription, passing a message handler that performs the following steps upon receiving the message:

- Gets the photo URL based on the bucket name and blob name.
- Creates and stores the metadata for the request, using request ID, name of the photo and the URL of the photo.

- Calls the landmark detection service using the URL of the photo, that returns a list of identified landmarks in the photo.
- For each of the landmarks, calls the map service to obtain the map image based on the landmark location (latitude, longitude) and stores it in the cloud storage.
- Store the landmarks in the metadata of the request with request ID.

4.2 GCP Services and APIs

In this section, we will discuss the various configurations and services utilized within the Google Cloud Platform (GCP) for our project. We will cover the creation of service accounts, the usage of Cloud Storage, Firestore, Pub/Sub and the other services and APIs used throughout the development of the project. These components play a crucial role in the deployment and functioning of our application.

It is important to note that we are utilizing the project created at the beginning of the semester, named "CN2223-T1-G03". This project serves as the foundation for our development and integration with GCP services.

During the development process, we encountered certain services and APIs that were not activated by default. Two notable examples include the Vision API and the Static Maps API. We needed to enable these services manually within the GCP console to leverage their functionalities for our project.

Additionally, to utilize the Static Maps API, we had to generate an API key. This key serves as the authentication mechanism for accessing the Static Maps service.

Service Accounts

Service accounts in Google Cloud Platform (GCP) are utilized to authenticate and authorize applications and services for accessing platform resources. For this project, we have established two service accounts: one for the gRPC server and Landmarks App, and another for the instance manager. By segregating the service accounts, we ensure that each component has appropriate access levels while adhering to the principle of least privilege.

The first service account, dedicated to the gRPC server and Landmarks App, is assigned the following roles with their respective explanations:

- Cloud Datastore Owner: This role provides full control over Cloud Datastore, allowing the service account to manage data and perform administrative tasks.
- Firestore Service Agent: By assuming this role, the service account can access Firestore and carry out necessary operations on behalf of the application.

- Pub/Sub Admin: This role grants the service account the authority to manage Pub/Sub resources, such as topics and subscriptions, enabling communication and event-driven workflows.
- Storage Admin: With this role, the service account obtains administrative access to Cloud Storage, allowing it to manage buckets, objects, and related permissions.

The second service account, dedicated to the instance manager, is assigned a single role: Compute Admin. This role equips the service account with the essential permissions to handle instances, including creating, deleting, starting, and stopping virtual machine instances.

Cloud Storage

Cloud Storage is an integral component of our project, used for storing and managing the landmarks photos and maps. We have created two separate buckets within Cloud Storage to cater to these different data requirements:

- `landmarks-photos` - This bucket is utilized for storing user-uploaded photos associated with landmarks.
- `landmarks-maps` - This bucket is dedicated to storing the static maps retrieved from the Static Maps API.

The separation of these buckets allows us to organize and manage the different types of data efficiently, ensuring optimal performance and access control.

The buckets were created with the following configuration:

- Location type: `Region`
- Location: `europe-southwest1 (Madrid)`
- Default storage class: `Standard`
- Access control: `Fine-grained`
- Public access prevention: `Not enabled`

Firestore

Firestore serves as our NoSQL document database, providing real-time data synchronization and scalability. In this project, we have created a collection named `landmarks` within Firestore to store the landmark detection requests. This collection is configured with appropriate security rules and indexes to ensure data integrity and efficient queries.

Pub/Sub

Pub/Sub is utilized as a messaging service within our project, facilitating asynchronous communication and event-driven interactions between the gRPC server and the Landmarks App. We have created a topic named "landmarks" within Pub/Sub, which serves as a central communication channel for relevant events and notifications related to landmarks.

The `landmarks` topic is configured to ensure reliable and durable message delivery. Additionally, we have created a single subscription to the topic, named `landmarks-sub`, which uses the `Pull` delivery method. This subscription is responsible for receiving and processing messages from the `landmarks` topic. It plays a crucial role in handling events such as landmark updates, user notifications, and other relevant triggers within our application.

4.3 Deployment

In this section, we will discuss the deployment process of our system. Our deployment strategy involves utilizing the Compute Engine service to host the virtual machines (VMs) where the gRPC server replicas and Landmarks App replicas are executed in two separate instance groups.

To ensure the smooth operation of our system, we conducted initial testing on our local machines. During this testing phase, we did not utilize the IP lookup function and only used a single instance of the gRPC server. The IP address of this server was hard-coded in the client application.

Once we successfully tested the system locally, we proceeded to generate artifacts for the gRPC client, server, and landmarks app. These artifacts were prepared to facilitate the deployment process on the Compute Engine VMs.

Instance Groups

To create the instance groups, we performed the following steps:

1. Created two VM instances, one named `landmarks-server` and the other named `landmarks-app`. These instances were created with the following configuration:

   - Series: `E2`
   - Type: `e2-small (2 vCPU, 2 GB memory)`
   - Operating System: `CentOS com versão 8 stream`
   - Zone: `europe-southwest1-a (Madrid)`

- Firewall: `Allow HTTP traffic`; `Allow HTTPS traffic`

Our SSH keys were added in order to connect to the VM using SSH. After the VM started, using the SSH client:

- Installed Java 11: `sudo yum install java-11-openjdk-devel`;
- Copied the jar with the application to the home directory;
- Tested if the application runs.

2. Created an image for each VM instance created in the previous step.

3. Created an instance template for each image created in the previous step, with a startup script to start the Java application. Figure 4.2, shows an example of a startup script configured in the template for the server. As shown, an environment variable named `GOOGLE_APPLICATION_CREDENTIALS` is defined with the key associated with the service account, and then the application is launched.

```
#! /bin/bash
export GOOGLE_APPLICATION_CREDENTIALS=/home/CN2223-T1-G03/cn2223-t1-g03-key.json
java -jar /home/CN2223-T1-G03/LandmarksServer-1.0-jar-with-dependencies.jar
```

Figure 4.2: Example of startup script.

4. Created two managed stateless instance groups: `instance-group-landmarks-server` and `instance-group-landmarks-app`, with the autoscale turned off. The minimum number of instances for the gRPC Server is 1, and for the workers is 0. The maximum number of instances for the gRPC Server is 3, and for the workers is 2.

4.3.1 Instance Manager

To efficiently manage our deployed instances, we developed an Instance Manager console application. This application provides a console interface for managing the instance groups on the Compute Engine. It allows us to control the number of replicas for both the gRPC server and the Landmarks App, ensuring scalability and high availability.

The Instance Manager application provides the following key functionalities:

- Listing the VMs in the instance groups;
- Scaling the instance groups.

By using the Instance Manager application, we can efficiently manage the scaling and maintenance of our system, thereby ensuring its reliable and smooth operation.

The Figure 4.3 shows the console menu of the instance manager.

```
########## Landmarks Instance Manager ##########
gRPC Server Instance Group:
 0 - List gRPC Server VM instances
 1 - Resize gRPC Server VM instances

Landmark App Instance Group:
 2 - List Landmark App VM instances
 3 - Resize Landmark App VM instances

 4 - Exit
Enter an Option:
```

Figure 4.3: Instance Manager console menu.

4.4 Work Distribution

In this section, we will outline the distribution of work among the members of the team and specify the areas in which each member had more or less responsibility. The following is a breakdown of the work distribution:

- Both members of the team:
  - gRPC Contract;
  - gRPC Client;
  - Deployment;
  - Documentation;
- André Jesus:
  - Landmarks App;
- Nyckollas Brandão:
  - gRPC Server.

It is important to note that both members of the team discussed and collaborated on major decisions, architecture design, and overall project management. Regular communication was held to ensure effective coordination between the team members. The breakdown of work distribution outlined above represents the primary areas of responsibility for each member, but it does not imply exclusive ownership or lack of contribution in other aspects of the project.

Chapter 5

Final Remarks

In summary, the main objective of this project was successfully achieved. We were able to implement and test the system with the proposed architecture. After this project, we are now more familiar with the concepts of cloud computing, scalability, and elasticity. Additionally, we have gained hands-on experience in utilizing various services and APIs provided by the Google Cloud Platform, such as Cloud Storage, Firestore, Pub/Sub, Compute Engine, Cloud Functions, Vision API, and Static Maps API. This project has deepened our understanding of leveraging cloud platforms for efficient resource allocation, cost-effectiveness, and the integration of services to enhance system functionality.

We have also acquired practical knowledge in building a scalable solution, which can be valuable for future projects and applications. We also believe we have successfully used knowledge from other courses, namely programming techniques like abstraction and modularity to ensure a more robust and decoupled backend solution.

The main challenges we faced were related to the deployment of the solution and the implementation of the IP lookup cloud function, since we had less experience working with the services used to finish these tasks.

Concluding, we believe that the project was a success, and we are satisfied with the results obtained.

References

[1] T. Dillon, C. Wu, and E. Chang, "Cloud computing: issues and challenges," in *2010 24th IEEE international conference on advanced information networking and applications.* Ieee, 2010, pp. 27–33.

[2] J. J. J. Geewax, *Google Cloud platform in action.* Simon and Schuster, 2018.

[3] Google, "Cloud storage," Google Cloud. [Online]. Available: https://cloud.google.com/storage

[4] "Cloud firestore," Google Cloud. [Online]. Available: https://cloud.google.com/firestore

[5] "Cloud pub/sub," Google Cloud. [Online]. Available: https://cloud.google.com/pubsub

[6] Google, "Compute engine: Virtual machines (vms) — google cloud," Google Cloud, 2012. [Online]. Available: https://cloud.google.com/compute

[7] "Cloud functions," Google Cloud. [Online]. Available: https://cloud.google.com/functions

[8] "Vision ai — derive image insights via ml — cloud vision api," Google Cloud. [Online]. Available: https://cloud.google.com/vision

[9] "Google maps platform documentation — maps static api," Google for Developers. [Online]. Available: https://developers.google.com/maps/documentation/maps-static

[10] K. Indrasiri and D. Kuruppu, *gRPC: up and running: building cloud native applications with Go and Java for Docker and Kubernetes.* O'Reilly Media, 2020.

[11] M. Richards, *Software architecture patterns.* O'Reilly Media, Incorporated 1005 Gravenstein Highway North, Sebastopol, CA . . . , 2015, vol. 4.

[12] baeldung, “Layered architecture — baeldung on computer science,” www.baeldung.com, 11 2021. [Online]. Available: https://www.baeldung.com/cs/layered-architecture

[13] “Protocol buffers,” Protocol Buffers Documentation. [Online]. Available: https://protobuf.dev/

[14] “Grpc core: Status codes and their use in grpc,” grpc.github.io. [Online]. Available: https://grpc.github.io/grpc/core/md_doc_statuscodes.html

Appendices

Appendix A

gRPC Contract

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "landmarks";

package landmarks;

service LandmarksService {
  rpc submitPhoto(stream SubmitPhotoRequest) returns (SubmitPhotoResponse) {}
  rpc getResults(GetResultsRequest) returns (GetResultsResponse) {}
  rpc getIdentifiedPhotos(GetIdentifiedPhotosRequest)
        returns (GetIdentifiedPhotosResponse) {}
}

// submitPhoto
message SubmitPhotoRequest {
  string photo_name = 1;
  bytes photo = 2;
}

message SubmitPhotoResponse {
  string request_id = 1;
}


// getResults
message GetResultsRequest {
  string request_id = 1;
}

message GetResultsResponse {
  repeated Landmark landmarks = 1;
  ImageMap map = 2;
}

message Landmark {
  string name = 1;
  double latitude = 2;
  double longitude = 3;
  float confidence = 4;
}

message ImageMap {
  bytes image_data = 1;
}


// getIdentifiedPhotos
message GetIdentifiedPhotosRequest {
  float confidence_threshold = 1;
}

message GetIdentifiedPhotosResponse {
  repeated IdentifiedPhoto identified_photos = 1;
}

message IdentifiedPhoto {
  string photo_name = 1;
  string landmark_name = 2;
  float confidence = 3;
}
```