KAZAKH BRITISH TECHNICAL UNIVERSITY

REPORT

Title: Cloud-Based Smart Inventory Management System

Akhmetkan Azhar

20.12.2024

Table of Contents

Executive summary

The Cloud-Based Smart Inventory Management System project is a comprehensive solution that aims to change the approach to inventory management in businesses. Using the power of Google Cloud Platform (GCP), the system provides a scalable, efficient and intelligent platform for real-time inventory tracking, product management across multiple locations and analytics using predictive algorithms.

This project:

- Provide a tool for real-time inventory control to minimize the risks of overstocking or shortages.
- Simplify inventory management for companies with multiple locations.
- Use predictive models to analyze historical data and predict trends.
- Ensure a reliable and scalable system by using cloud architecture.

The project addresses key issues faced by companies like manual errors, which is automation of processes reduces the risk of accounting errors.Second is resource optimization, where cost reduction through optimal inventory management. Integrating analytics helps make more informed decisions in Data-driven decisions. Also the project system demonstrates increasing operational efficiency due to reduced accounting errors, increasing customer satisfaction due to timely order fulfillment, reduction of cost due to reduced storage costs and reliability and scalability due to the use of cloud technologies.

Introduction

The evolution of technology has significantly changed the way businesses operate, and cloud computing has become the cornerstone of today's digital transformation. Cloud computing provides on-demand access to shared computing resources such as data storage, processing power, and applications over the Internet. This allows companies to reduce operational costs, increase scalability, and improve overall efficiency.

In the area of inventory management, traditional approaches are often based on manual processes or legacy systems, which lead to inefficiencies such as poor inventory management, increased costs, and slower decision making. A cloud inventory management system addresses these issues by providing businesses with tools for automated real-time monitoring, data analysis, and forecasting, which helps them adapt to changing requirements and the competitive environment.

The Cloud-Based Smart Inventory Management System project is designed to leverage the power of cloud technologies to provide companies with a reliable, scalable, and intelligent solution. The system, hosted on Google Cloud Platform (GCP), combines real-time monitoring, data analytics, and forecasting to optimize inventory and improve operational efficiency.

Project Goals

The key goals of this project include:

- Real-time inventory monitoring: Provide companies with instant visibility into inventory levels, minimizing the risk of overstocking or understocking.
- Multi-location inventory management: Provide a centralized tool to monitor and manage inventory across multiple locations
- Predictive analytics: Use machine learning models to analyze historical sales data and predict future trends, enabling data-driven decisions.
- Scalability and reliability: Leverage the flexibility of the GCP platform to ensure stable system operation as workloads fluctuate and business demands increase.

Project Scope

The scope of this project includes:

- Development of a web application for managing inventory management processes.
- Implementation of a RESTful API for managing inventory, orders, and suppliers.
- Configuration of cloud storage for centralized storage of product data and backups.
- Using machine learning models to analyze sales trends and forecast demand.
- Ensuring reliable security measures using identity and access management (IAM).

While the system provides key inventory management features, some advanced capabilities, such as integration with third-party platforms and mobile apps, are beyond the scope of the current project. These features may be considered in future iterations. By combining the strengths of cloud technology and intelligent systems, the Smart Cloud Inventory Management System aims to transform approaches to inventory management and pave the way for a more efficient and data-driven future.

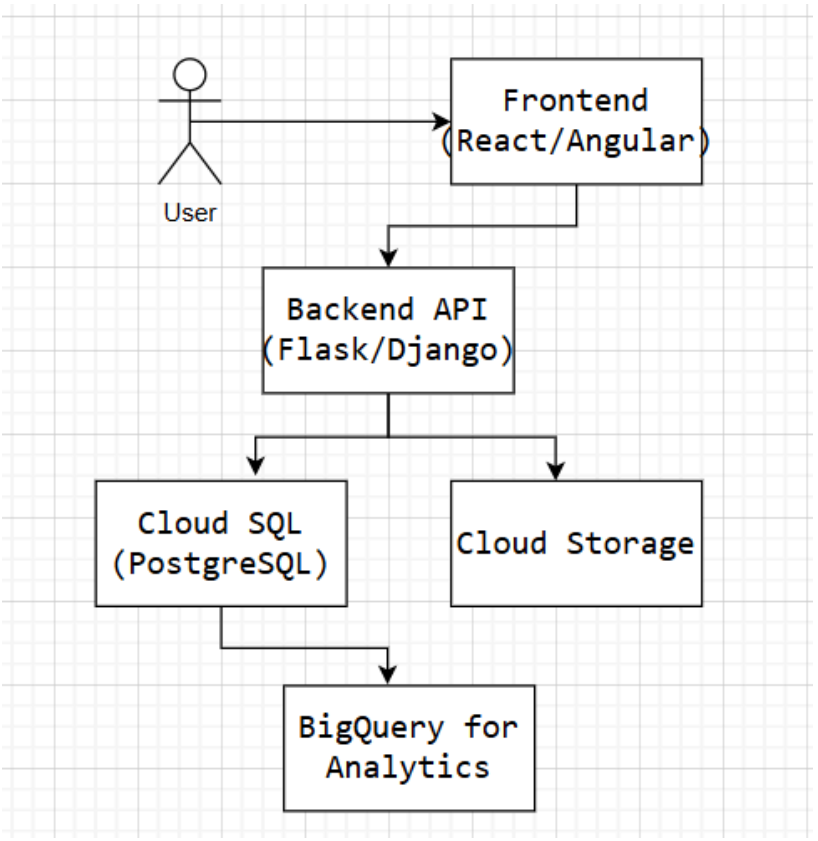
System Architecture

The architecture of the inventory management system is built on a modular principle, which ensures its flexibility, scalability and ease of maintenance. The main components include the user interface (frontend), the server part (backend), the database, cloud services for data storage, analytics and computing, as well as a notification and monitoring system.

Architecture diagram

- The user interacts with the interface via a browser.
- The interface sends requests to the server part (RESTful API).
- The server part interacts with the database to perform CRUD operations.
- Cloud services process data, store images, and manage logs.

Example diagram (ERD or system diagram):



Architecture Components:

Frontend (User Interface) is an intuitive web interface designed for user interaction with the system. The main functions of the interface include viewing and managing products, orders and categories, displaying analytics and reports, and providing notifications about errors and critical events. Modern technologies such as React or Angular are used to implement the interface, which allows you to create a Single Page Application (SPA) with high performance and ease of use.

Backend (Server Part) is responsible for processing requests from the user interface, executing business logic and interacting with the database. It implements a RESTful API, providing a unified approach to data processing. The server part performs key tasks, including user management, order processing and analytics. Python with Flask or Django, as

well as Node.js, are used to implement the backend, which provides flexibility and scalability.

Database is used to store data about users, products, orders, categories, and logs. Google Cloud SQL based on PostgreSQL was chosen, which ensures reliability, scalability, and performance. The main tables include:

- User: storing user data such as name, email, and role.
- Product: product information including name, quantity, and price.
- Order: order records associated with users and products.
- Category: product classification.
- InventoryLog: inventory change log reflecting events such as adding or writing off products.

Cloud Services (Google Cloud Platform) to ensure high availability and fault tolerance:

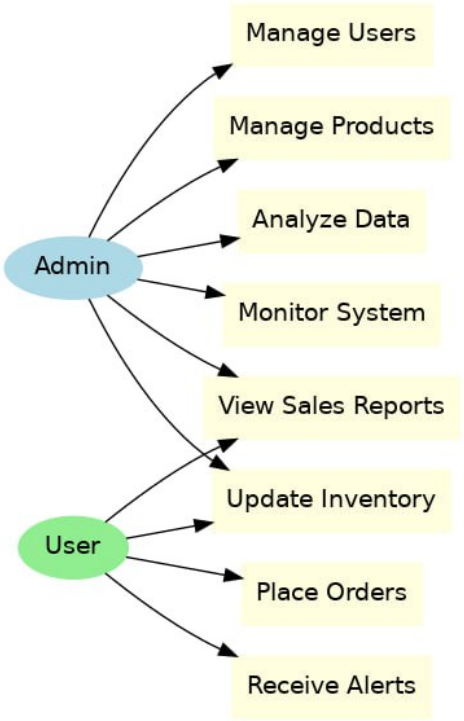
- Compute Engine: hosting the server part for processing requests.
- Cloud Storage: storing product images and data backups.
- BigQuery: performing big data analytics and building reports.
- Cloud Functions: Performing serverless tasks, such as automatically sending notifications when critical inventory levels are reached.

Notifications are responsible for informing users about critical events, such as low inventory levels, order errors, or other important events. Third-party services such as Twilio or Google Firebase are used to deliver notifications, allowing you to send notifications via email or SMS. This ensures timely information and prevention of potential problems.

Monitoring and logging to ensure reliable operation of the system, Google Cloud Monitoring is used, which allows you to track the performance of the server part (CPU load, API response time). Logs are collected and processed via Google Cloud Logging, which simplifies error analysis and system optimization.

This architecture ensures the flexibility, security, and scalability of the system, creating a reliable foundation for real-time inventory management.

Interaction of components



Order creation process:

- The user sends a request to create an order via the interface.
- The interface sends the request to the server part via API.
- The server part checks the availability of the product and updates the Order table
- The inventory update is recorded in the InventoryLog table.

Analytics process:

- Historical data from the database is transferred to BigQuery.
- SQL queries are executed to build sales and inventory reports.
- The results are transferred to the interface and displayed as graphs.

Notification process:

- The server part analyzes the inventory level.
- When a critical level is reached, a notification is created.
- The notification is sent to the user via Cloud Functions.

Key architectural features

The system has a number of key characteristics that ensure its reliability, scalability, security and flexibility.

Scalability is achieved through the use of the Google Cloud Platform (GCP) infrastructure, which allows for automatic scaling of resources depending on the load. This ensures efficient processing of API requests even under high loads and an increasing number of users.

Security is achieved through the use of Google Identity and Access Management (IAM), which provides strict control over access to data. Additionally, data encryption is implemented at all levels - both when stored in the database and when transferred between system components.

Fault tolerance is ensured by data backup in Google Cloud Storage, which minimizes data loss in the event of a failure. Database replication reduces the risk of downtime, ensuring data availability even if the primary node fails.

Flexibility is achieved through the modular structure of the system, which simplifies the addition of new functions. This allows the system to be easily integrated with mobile applications, ERP systems or other external platforms.

The system is built using cutting-edge technologies. On the Frontend side, React, Angular, HTML5, and CSS3 are used to create a user-friendly interface. The Backend is implemented in Python (Flask or Django) with a REST API that provides interaction between components. The database is based on Google Cloud SQL with PostgreSQL, providing reliable and scalable data storage. For cloud computing, GCP services such as Compute Engine, Cloud Storage, and BigQuery are used. DevOps tools, including Docker and Kubernetes, are used to manage deployment and maintain containers. System monitoring is carried out through Google Cloud Monitoring and Logging, which allows you to track performance, identify errors, and optimize system operation. This architecture provides a solid foundation for effective inventory management and scaling the system as business requirements grow.

Table Entities

This section describes the database that is the basis of The Cloud-Based Smart Inventory Management System. It includes key entities used to store and manage data.

Database structure overview

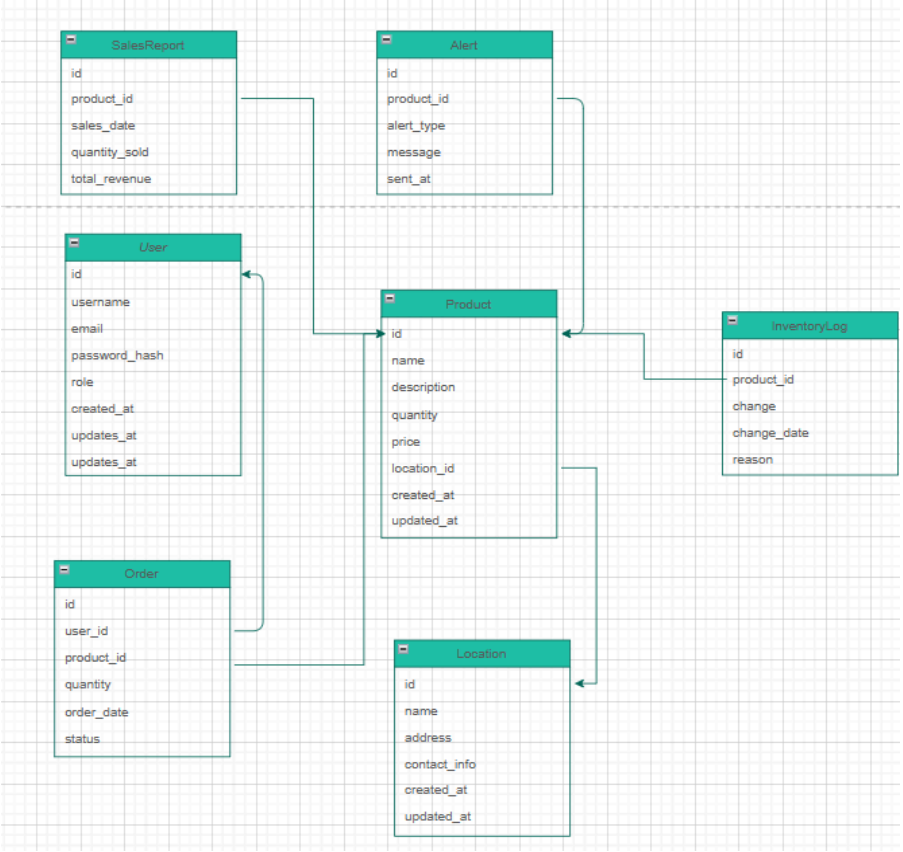
The system's functionality is implemented using a relational database model. The main tables and their relationships allow storing and processing data on users, products, orders, locations, suppliers, and events.

The database schema is designed to provide:

1. **Flexibility** — support for changes and adding new entities.
2. **Scalability** — ability to process large amounts of data.
3. **Data integrity** — use of keys and relationships to prevent errors.

Here relationships between tables like in Figure .

- User is related to Order via the user_id foreign key.
- Product is related to Order, InventoryLog, and Alert via product_id.
- Location is related to Product via location_id.



List of tables:

1. ***User*** **-** Stores information about system users.
   Fields:
   id: Unique user identifier (PRIMARY KEY).
   username: Username.
   email: Email.
   password_hash: Hashed password for security.

role: User role (e.g. "admin", "manager").
created_at: Date the record was created.
updated_at: Date the record was last updated.
SQL query example:

```
CREATE TABLE User (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    role VARCHAR(20) DEFAULT 'user',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP
);
```

2. ***Product -*** Contains information about products in stock.
   Fields:
   id: Unique product identifier (PRIMARY KEY).
   name: Product name.
   description: Product description.
   quantity: Product quantity in stock.
   price: Product price.
   location_id: Foreign key to the Location table.
   created_at: Date the record was created.
   updated_at: Date the record was last updated.
   SQL query example:

```
CREATE TABLE Product (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    description TEXT,
    quantity INT NOT NULL,
    price DECIMAL(10, 2) NOT NULL,
    location_id INT REFERENCES Location(id),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP
);
```

3. ***Location -*** Indicates the location of warehouses and offices.
   Fields:
   id: Unique location identifier (PRIMARY KEY).
   name: Location name (e.g. "Central Warehouse").
   address: Location address.
   contact_info: Contact information.
   created_at: Date the record was created.
   updated_at: Date the record was last updated.
4. ***Order*** **-** Stores information about orders.

Fields:
id: Unique order identifier (PRIMARY KEY).
user_id: Foreign key to the User table.
product_id: Foreign key to the Product table.
quantity: Quantity of products ordered.
order_date: Date the order was created.
status: Order status (e.g. "processing", "completed").
SQL query example:

```
CREATE TABLE Order (
    id SERIAL PRIMARY KEY,
    user_id INT REFERENCES User(id),
    product_id INT REFERENCES Product(id),
    quantity INT NOT NULL,
    order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    status VARCHAR(20) DEFAULT 'pending'
);
```

5. ***Supplier -*** Contains data on product suppliers.
   Fields:
   id: Unique supplier identifier (PRIMARY KEY).
   name: Supplier name.
   contact_info: Contact information (phone, email).
   created_at: Date the record was created.
   updated_at: Date the record was last updated.
6. ***InventoryLog -*** Records all changes in inventory (e.g. adding or removing items).
   Fields:
   id: Unique identifier (PRIMARY KEY).
   product_id: Foreign key to the Product table.
   change: Change value (e.g. +10 or -5).
   change_date: Date of change.
   reason: Reason for change (e.g. "order", "return").
   SQL query example:

```
CREATE TABLE InventoryLog (
    id SERIAL PRIMARY KEY,
    product_id INT REFERENCES Product(id),
    change INT NOT NULL,
    change_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    reason TEXT
);
```

7. ***SalesReport -*** Stores aggregated sales data.
   Fields:
   id: Unique identifier (PRIMARY KEY).
   product_id: Foreign key to the Product table.
   sales_date: Sale date.
   quantity_sold: Number of units sold.

total_revenue: Total revenue.

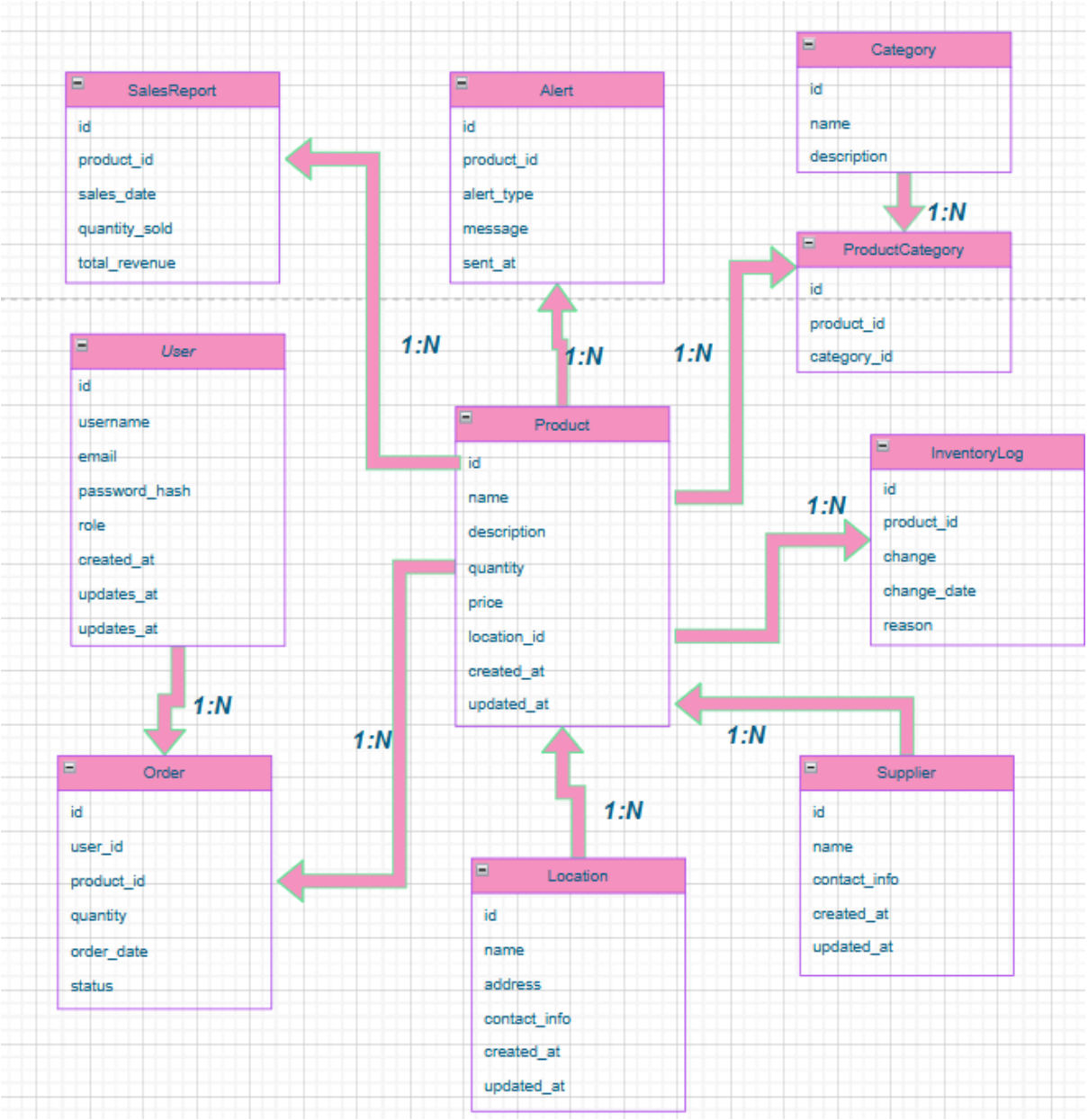
8. ***Alert -*** Contains data about notifications for users.
   Fields:
   id: Unique identifier (PRIMARY KEY).
   product_id: Foreign key to the Product table.
   alert_type: Alert type (e.g. "low stock").
   message: Alert text.
   sent_at: Time the alert was sent.
9. ***Category -*** The Category table is used to classify products. Each category has a unique ID, name, and description, which simplifies the management of product groups.
   Fields:
   id: Unique identifier (PRIMARY KEY).
   name: Category name (e.g. "Electronics", "Furniture").
   description: Category description.
   SQL query example:

```
CREATE TABLE Category (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    description TEXT
);
```

10. ***ProductCategory -*** The ProductCategory table creates a many-to-many relationship between Products and Categories. For example, one product can be assigned to several categories at once, such as "Electronics" and "New Products".
    Fields:
    id: Unique identifier (PRIMARIES KEY).
    product_id: Forage Key to the Product tablet.
    category_id: Forage Key to the Category tablet.
    SQL query example:

```
CREATE TABLE ProductCategory (
    id SERIAL PRIMARY KEY,
    product_id INT REFERENCES Product(id) ON DELETE CASCADE,
    category_id INT REFERENCES Category(id) ON DELETE CASCADE
);
```

Example of an ERD diagram is shown :


SalesReport
id
product_id
sales_date
quantity_sold
total_revenue
Alert
id
product_id
alert_type
message
sent_at
Category
id
name
description
1:N
ProductCategory
id
product_id
category_id
User
id
username
email
password_hash
role
created_at
updates_at
updates_at
1:N
1:N
1:N
Product
id
name
description
quantity
price
location_id
created_at
updated_at
1:N
InventoryLog
id
product_id
change
change_date
reason
1:N
1:N
1:N
Order
id
user_id
product_id
quantity
order_date
status
1:N
Location
id
name
address
contact_info
created_at
updated_at
Supplier
id
name
contact_info
created_at
updated_at

Development Process

The development of the "Smart Cloud Inventory Management System" system was organized in several stages to ensure logical and consistent execution of all tasks. Each stage included the use of modern technologies, tools and approaches to achieve the set goals.

Technologies Used

Programming Languages:

- Backend: Python using Flask/Django frameworks to implement RESTful API.
- Frontend: JavaScript using React/Angular to build Single Page Application (SPA).

Database:

- Google Cloud SQL based on PostgreSQL for reliable data storage.

Cloud Services:

- Google Cloud Platform (GCP): Compute Engine, Cloud Storage, BigQuery.

DevOps Tools:

- Docker for application containerization.
- Kubernetes for container management and scaling.

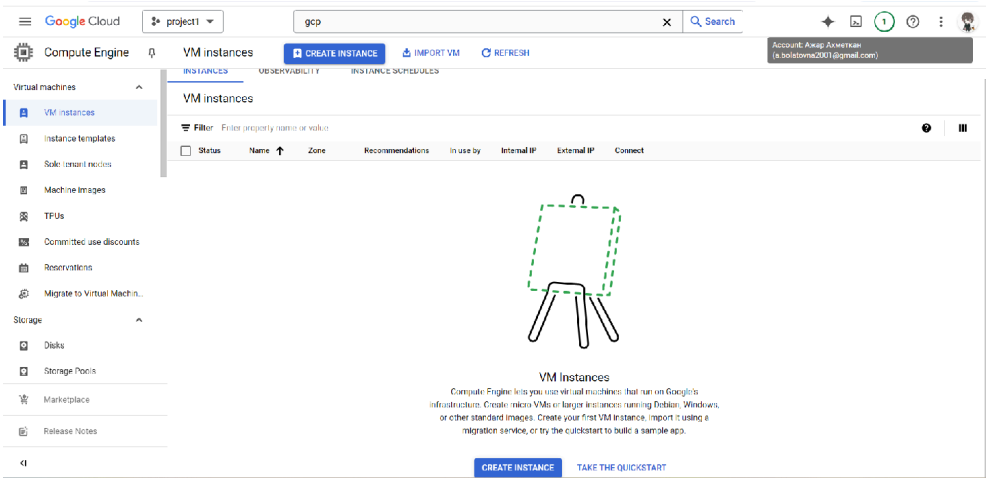
Monitoring Systems:

- Google Cloud Monitoring and Logging for performance monitoring and error analysis.

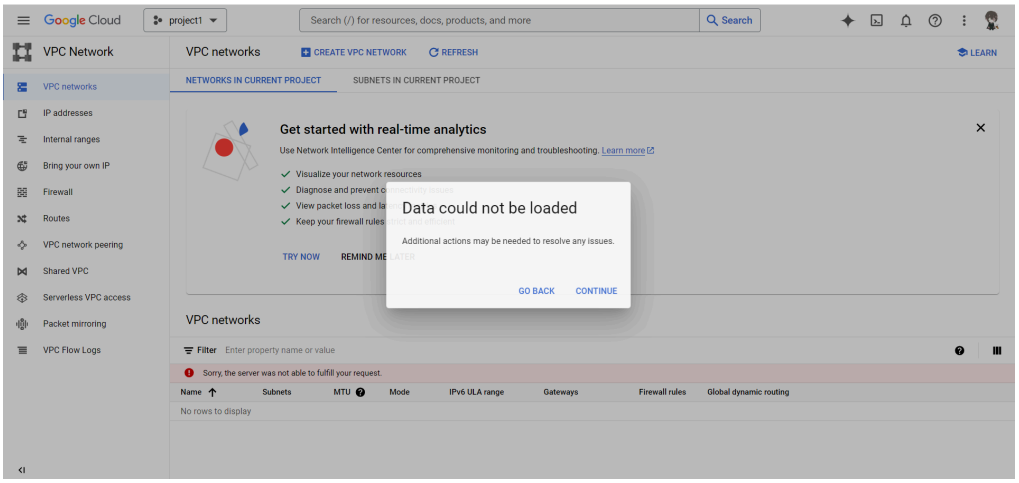
Main stages of development:

Each component of the system was implemented using relevant technologies and Google Cloud Platform (GCP) cloud services, ensuring reliability, scalability, and security. This article describes how the main elements of the system were implemented and how GCP services were used.

1. Backend - The backend is implemented in Python using Flask/Django to create a RESTful API. It is responsible for processing business logic, interacting with the database, and managing data flows.
    - Google Cloud Compute Engine (VM): The backend was deployed on a GCP virtual machine (VM) with Ubuntu pre-installed. Python, Flask/Django, PostgreSQL, and the necessary libraries were installed. The API server is hosted on Gunicorn/Nginx to manage requests.



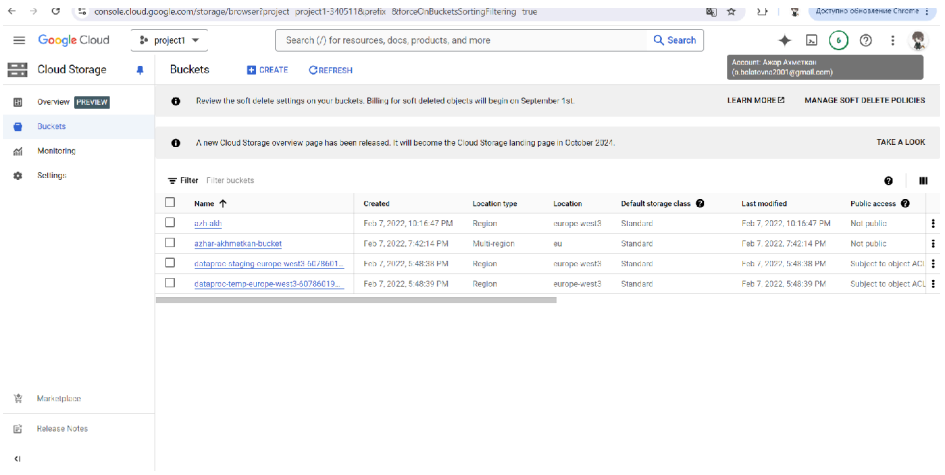
- Networking: Firewall rules were configured to securely access only the necessary ports (e.g. 80 for HTTP, 443 for HTTPS). GCP Load Balancer was used to distribute the load across multiple VM instances.



VM configuration example:

- Machine type: e2-medium (2 vCPU, 4 GB RAM).
- Disks: 50 GB SSD for fast reading/writing.

2. Frontend (User Interface) - Frontend is built on React/Angular and provides an intuitive interface for working with the system.
   - Google Cloud Storage: The web application is uploaded to cloud storage as static files (HTML, CSS, JS), accessible via HTTPS. CORS configuration to ensure interaction with the Backend API. Configured automatic file updates via CI/CD.



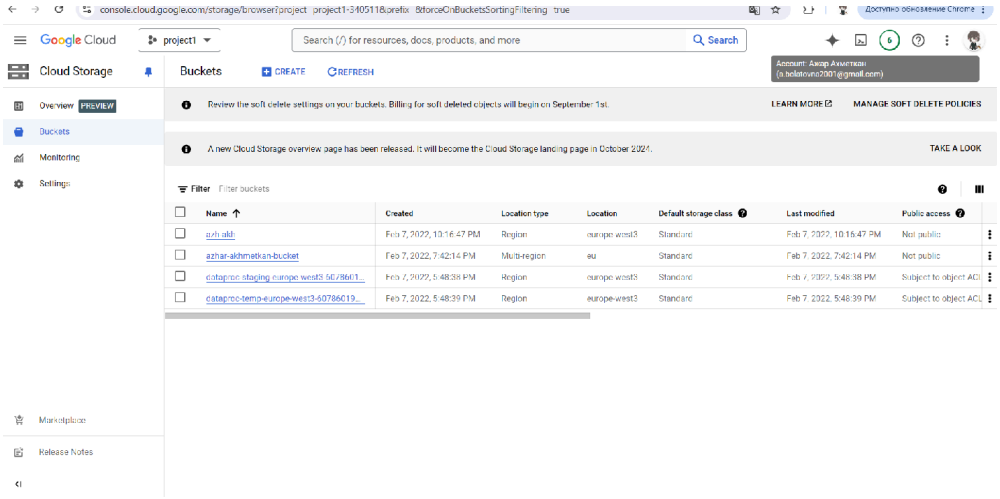
   - Cloud CDN: Connected to speed up the delivery of static resources to users around the world.

3. Database - A relational database is used to store data about users, products, orders, and categories. Google Cloud SQL (PostgreSQL): PostgreSQL database is deployed. Data replication is configured to ensure fault tolerance. Automatic backup is enabled.
Database Configuration:
   - Instance Type: db-f1-micro.
   - Disk Space: 20 GB SSD.

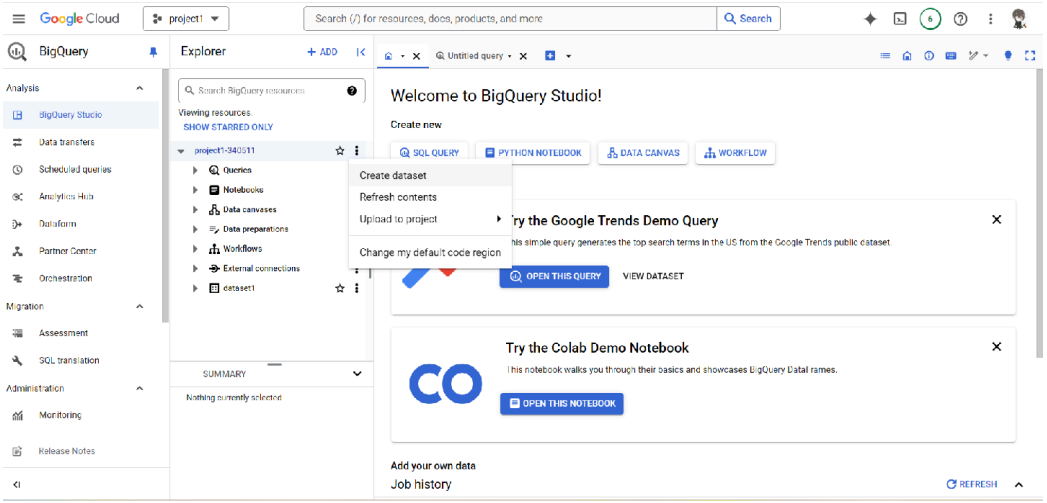
- IP Access is Limited to VPC Virtual Machines Only.

4. Data Storage - Product images, backups, and big data storage for analytics are provided. Several buckets are configured to store different types of data:
   - product-images: product images.
   - backup-data: database backups.

   Automatic object lifecycle management is enabled to optimize costs.



5. Analytics - The system uses analytics tools to process big data and generate reports. Sales data is loaded from Cloud SQL to BigQuery to perform analytics queries.



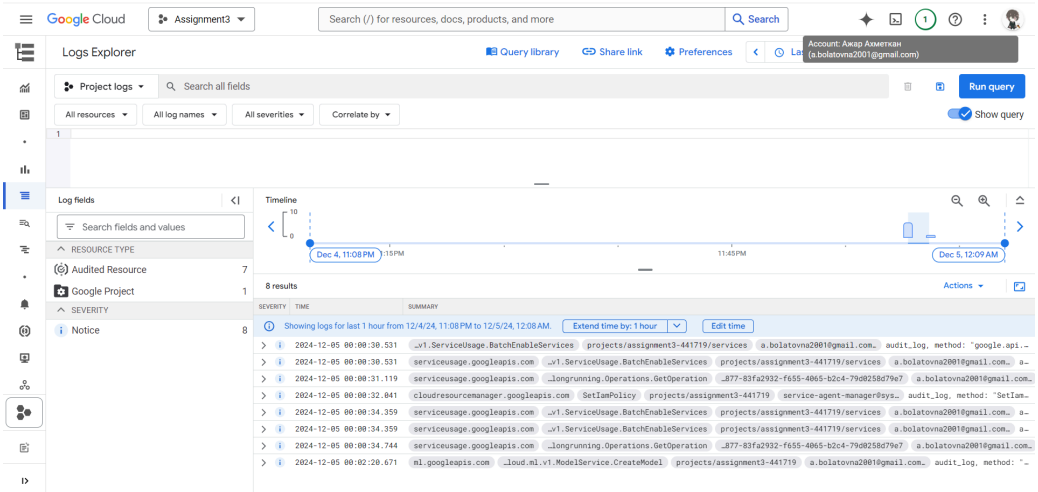
Predictive machine learning models are implemented using SQL queries and built-in BigQuery ML functions.

Example of an analytical query:

```
SELECT
    product_id,
    SUM(quantity_sold) AS total_sales,
    SUM(total_revenue) AS total_revenue
FROM
    `project_id.dataset.sales_report`
GROUP BY
    product_id
```

```
ORDER BY
    total_revenue DESC
LIMIT 10;
```

6. Notifications - Implemented a notification system to inform users about important events.
   Implementation:
   - Cloud Functions: Used to generate notifications about low inventory levels. Functions are called when triggers are fired from the database or Cloud Storage.
   - Google Firebase: Integration for sending push notifications and SMS.
7. Monitoring and logging - The monitoring system allows you to track performance and identify errors.
   Implementation:
   - Cloud Monitoring: Metrics are configured to monitor CPU load, memory usage, and API performance.
   - Cloud Logging: Request and error logs are recorded for analysis and troubleshooting. Alerts are created for critical events (e.g., exceeding the API response time).



Implementation of system components using Google Cloud Platform services ensured the stability, security, and performance of the system. The modular structure allows you to easily scale the system and add new features, providing reliable inventory management in real time.

API Design and Implementation

This section describes the RESTful APIs created to interact with the database and provide functionality for the inventory management system. The API is built on modern REST principles to be scalable, secure, and easy to integrate.

API Design Principles

1. Clean and Readable:
    - Endpoints represent entities (e.g. /products, /orders).
    - Standard HTTP methods are used (GET, POST, PUT, DELETE).
2. Security:
    - Authorization via tokens (e.g. JWT).
    - Restricting access via roles (e.g. administrator, manager).
3. Data Structure:
    - Responses are returned in JSON format.
    - Standard HTTP codes are used for error handling (200, 404, 500).

Description of API endpoints:

Method	**URL**	**Description**	**Request**	**Result**

GET	/api/products	Getting a list of all products.	`GET /api/products HTTP/1.1`
			`Host: example.com`
			`Authorization: Bearer <access_token>`
				`{ "id": 1, "name": "Phone", "quantity": 50, "price": 999.99, "category": "Electronic" }`

POST	/api/products	Creating a new product.	`POST /api/products HTTP/1.1`
			`Host: example.com`
			`Authorization: Bearer <access_token>`
			`Content-Type: application/json`
				`{ "message": "Product created successfully!", "product_id": 3 }`

PUT	/api/products/{id}	Updating information about a specific product.	`PUT /api/products/1 HTTP/1.1`
			`Host: example.com`
			`Authorization: Bearer <access_token>`
			`Content-Type: application/json`
			`{ "quantity": 60, "price":`
				`{ "message": "Product deleted successfully!" }`

			899.99 }	

DELETE	/api/products/{id}	Removing product.	DELETE /api/products/1 HTTP/1.1
			Host: example.com
			Authorization: Bearer <access_token>
				{
				"message": "Product deleted successfully!"
				}

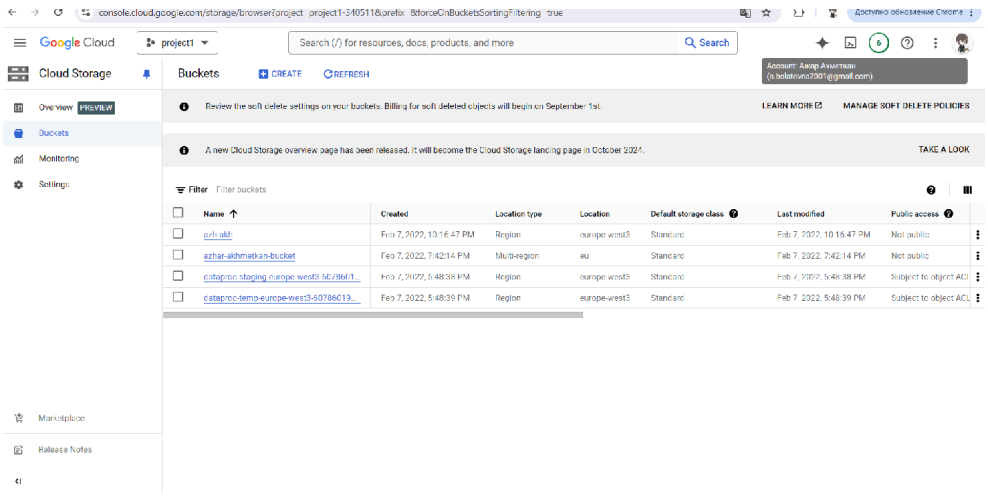
POST	/api/auth/login	User authorization.	POST /api/auth/login HTTP/1.1
			Host: example.com
			Content-Type: application/json
			{
			"email": "user@example.com",
			"password": "password123"
			}
				{
				"token": "<access_token>"
				}

Cloud Storage Solutions

Learn how to store, manage, and access data using Google Cloud Storage.

Create a new Cloud Storage bucket

- Go to the **Google Cloud Console** and navigate to the **Cloud Storage** section.
- Click **Create** next to "Buckets."
- Set up the bucket by choosing:
    - **Name**: A unique name for the bucket (e.g., exercise_5).
    - **Location**: Choose the region for data storage (e.g., Multi-region for global access or Region for localized storage).
    - **Storage class**: Select Standard, Nearline, Coldline, or Archive, depending on the use case.
    - **Access control method**: Choose either Uniform (bucket-level access) or Fine-grained (object-level access).
- Click **Create** to finalize the bucket creation.



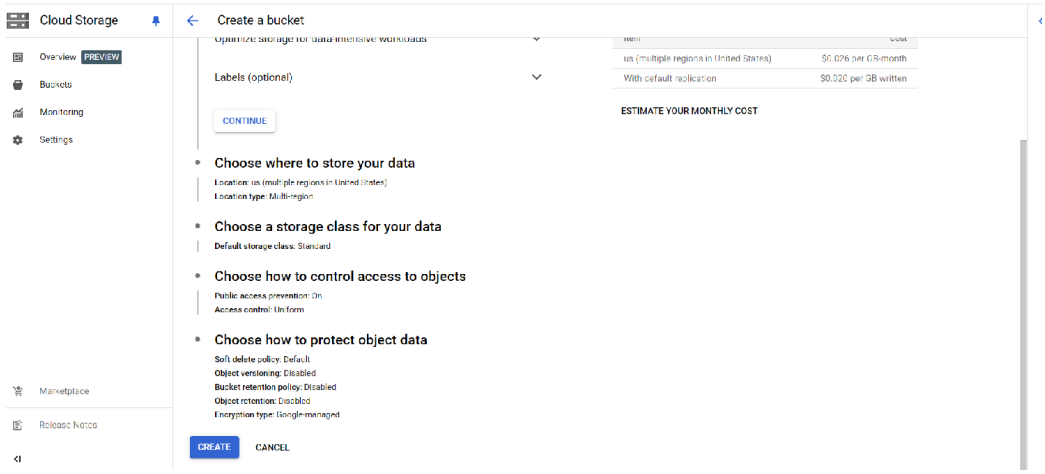
Upload files to the bucket

- Use the **Upload files** option in Google Cloud Console to add files, such as:
    - Text files (e.g., .txt).
    - Images (e.g., .jpg, .png).
    - Videos (e.g., .mp4).
- Drag and drop files or select them manually from your system to upload them to the bucket.

Set access permissions and test file access

- Open the bucket properties in **Cloud Console**.
- Adjust access settings:
    - **Public access**: Add "allUsers" with the Viewer role for public accessibility.
    - **Private access**: Leave the default settings to restrict access to authenticated users.
- Test access by opening the object URL:
    - Public files should open without authentication.

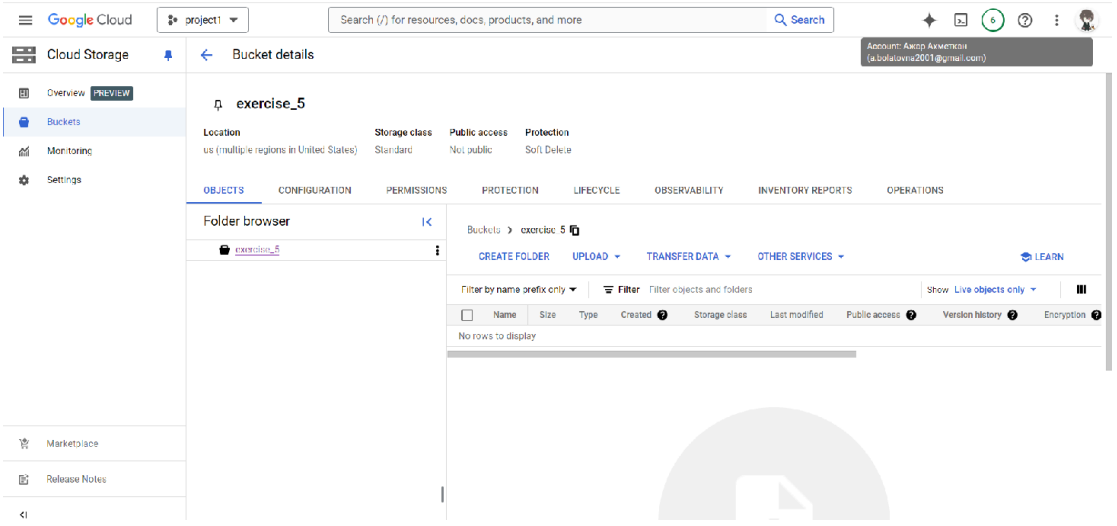
- Private files will require user authentication.



Manage files in the bucket

- Use Cloud Console to perform the following operations:
  - **Download files**: Select a file and click **Download**.
  - **Move files**: Use **Move** to transfer files between buckets.
  - **Delete files**: Select a file and click **Delete** to remove it.

And got exercise_5 bucket:



Identity and Security Management

The application has implemented the Identity and Access Management (IAM) system in Google Cloud to properly manage access and protect data. First, the user creates an IAM role, which defines access rights for different users and groups. In the Google Cloud Console, he goes to the IAM and Administration section,

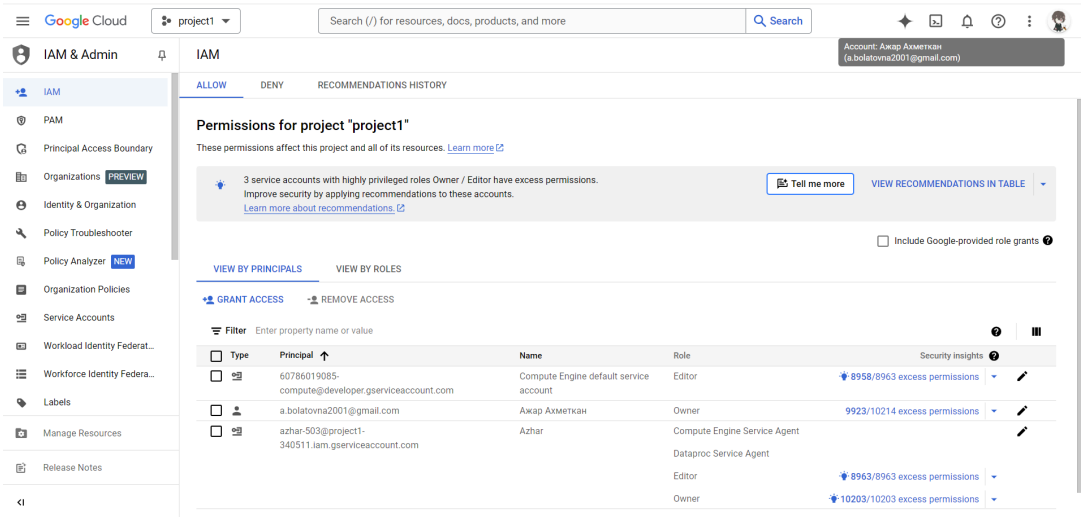


Image 4.1

adding new users and assigning them the appropriate roles, such as "Viewer", "Editor" or "Owner". This allows you to set clear levels of access to resources and manage user rights depending on their roles. There are user roles for more granular access control with specific permissions required to perform specific tasks. This helps avoid excessive rights and supports the principle of least privilege, which significantly reduces security risks.

In addition to setting up IAM, additional measures have been implemented to protect data. To ensure the security of information both during transmission and at the storage stage, encryption has been used. All data transmitted between clients and servers is encrypted using the HTTPS protocol, which protects it from interception. Built-in encryption is also used for data stored in Google Cloud Storage, providing additional protection.

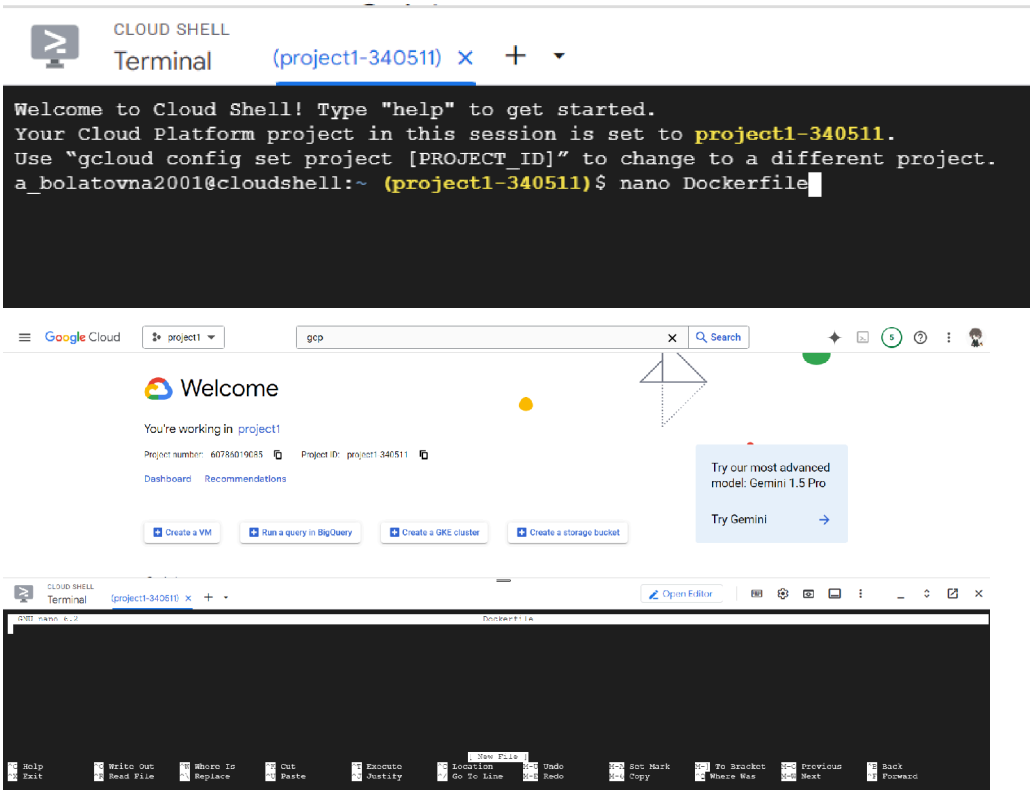
Additionally, the user has set up data access control using security policies that restrict access to certain resources based on user roles. This allows precise control over who has the right to view, modify, or delete data, protecting sensitive information. Thus, implementing IAM and applying comprehensive security measures ensures reliable access control and data protection, which is critical to the stability and security of an application in Google Cloud.

Kubernetes and Container Management

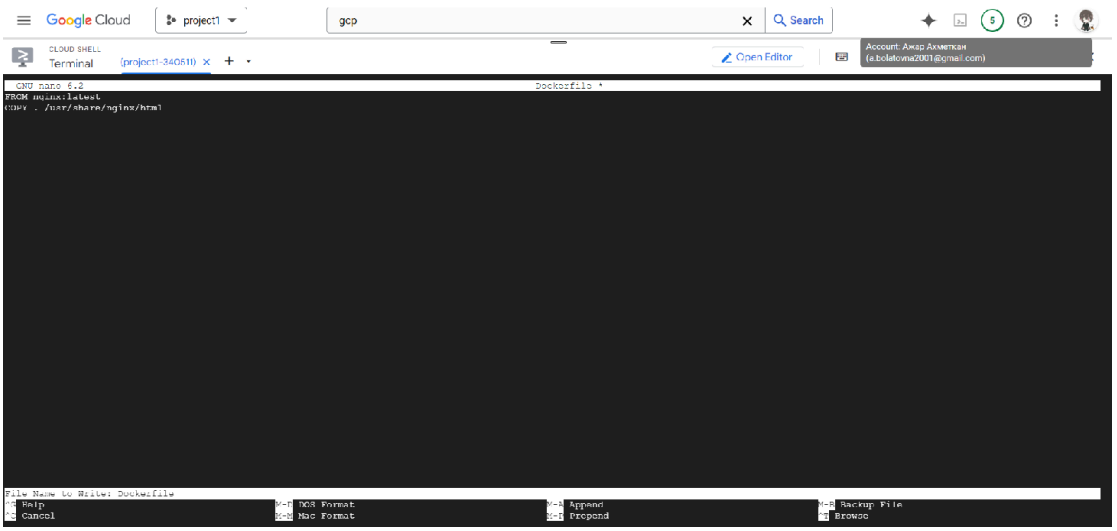
Objective: Learn how to deploy and manage containerized applications using Google Kubernetes Engine.

Steps:

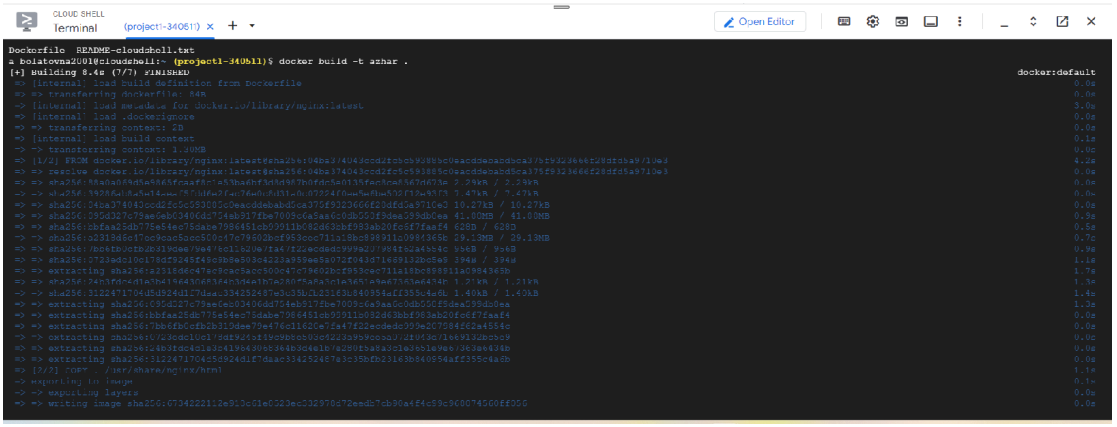
I started by opening the Terminal through Google Cloud Shell, which is accessible from the Google Cloud Console. Using the nano text editor in the terminal, I created a Dockerfile to define the container environment. After configuring the Dockerfile, I built and pushed the container image to Google Container Registry (GCR).



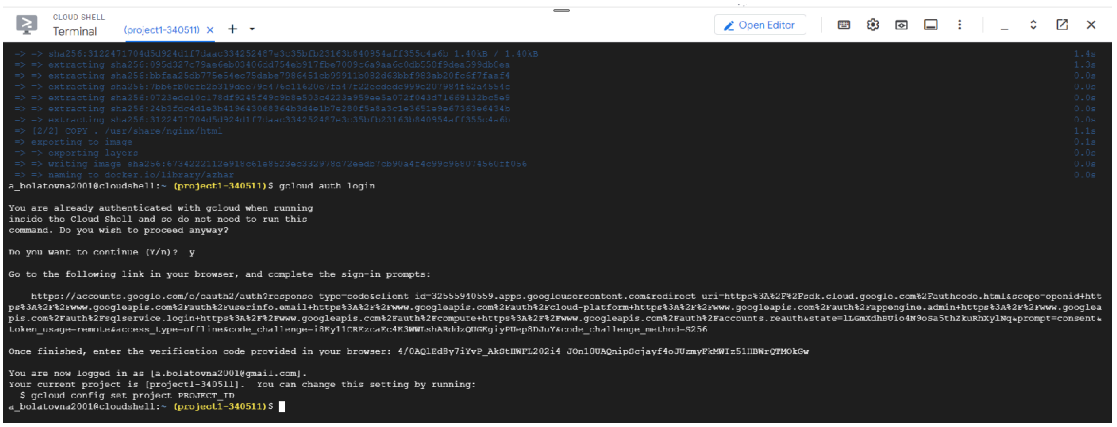
Create the required instructions for a simple web application using Nginx in the Dockerfile. Write the configuration to define the environment for the application. Once completed, save the file by pressing **Ctrl + O** to write the changes to disk. Additionally, ensure to include the necessary port exposure and entry point commands for Nginx to serve the application.



After saving the file, I exited the Nano editor by pressing **Ctrl + X**. I then proceeded to build the Docker image named "azhar" by running the following command in the terminal:
docker build -t azhar .



The next step was to authenticate with Google Cloud. Still in the terminal, I logged in to my Google Cloud account by running the following command:
gcloud auth login
This command opened a window prompting me to enter my Google account credentials and complete the authentication process.



Using the terminal I tagged image for Google Container Registry and finally, pushed tagged image to Google Container Registry by commands below:

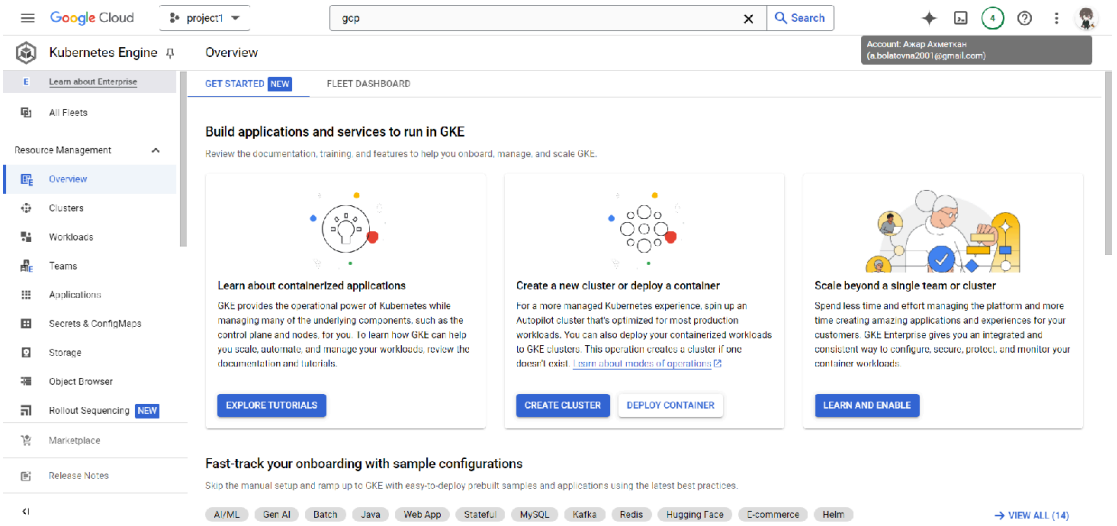
--docker tag azhar gcr.io/340511/azhar

--docker push gcr.io/340511/azhar

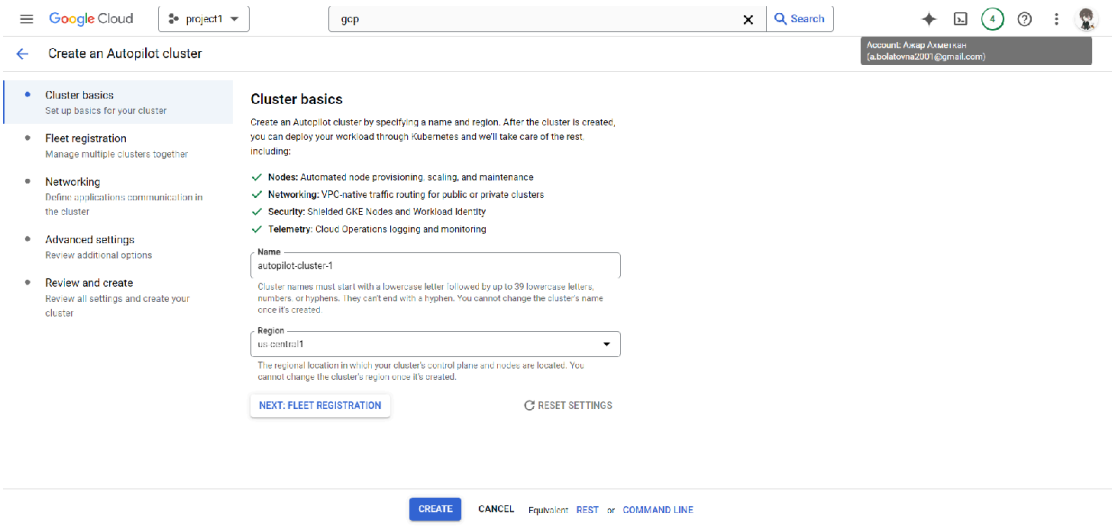
```
Once finished, enter the verification code provided in your browser: 4/0AQlEd8y7iYvP_AkStHWFI

You are now logged in as [a.bolatovna2001@gmail.com].
Your current project is [project1-340511].  You can change this setting by running:
  $ gcloud config set project PROJECT_ID
a_bolatovna2001@cloudshell:~ (project1-340511)$ docker tag azhar gcr.io/340511/azhar
a_bolatovna2001@cloudshell:~ (project1-340511)$ docker push gcr.io/340511/azhar
Using default tag: latest
The push refers to repository [gcr.io/340511/azhar]
41be875c576e: Preparing
11de3d47036d: Preparing
16907864a2d0: Preparing
2bdf51597158: Preparing
0fc6bb94eec5: Preparing
eda13eb24d4c: Waiting
67796e30ff04: Waiting
8e2ab394fabf: Waiting
```

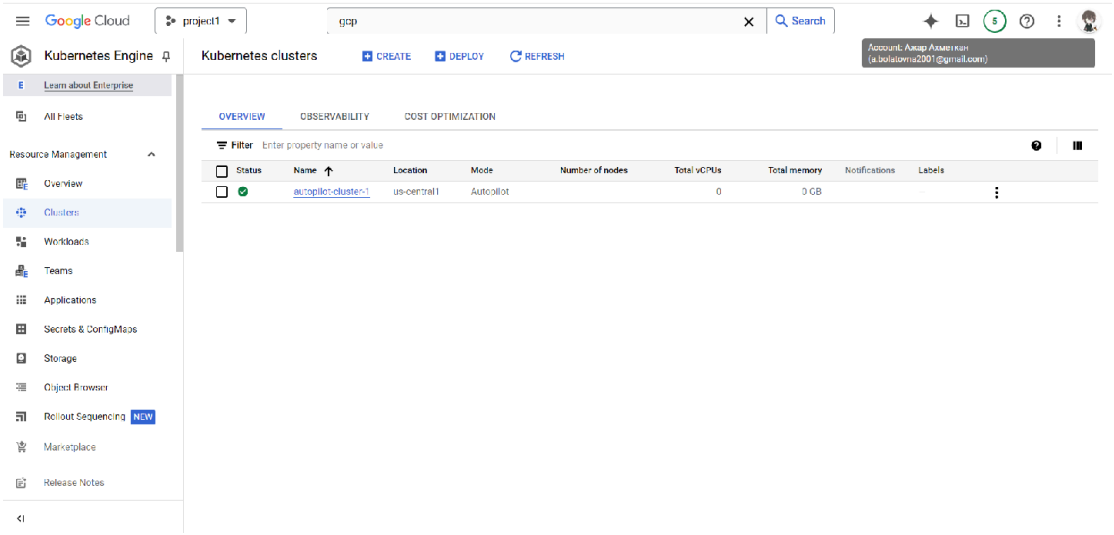
To set up the GKE cluster, I navigated to the Google Cloud Console and selected Kubernetes Engine. I clicked on “Create Cluster”



configured the settings, such as choosing the cluster type, specifying the number of nodes, and selecting the region.

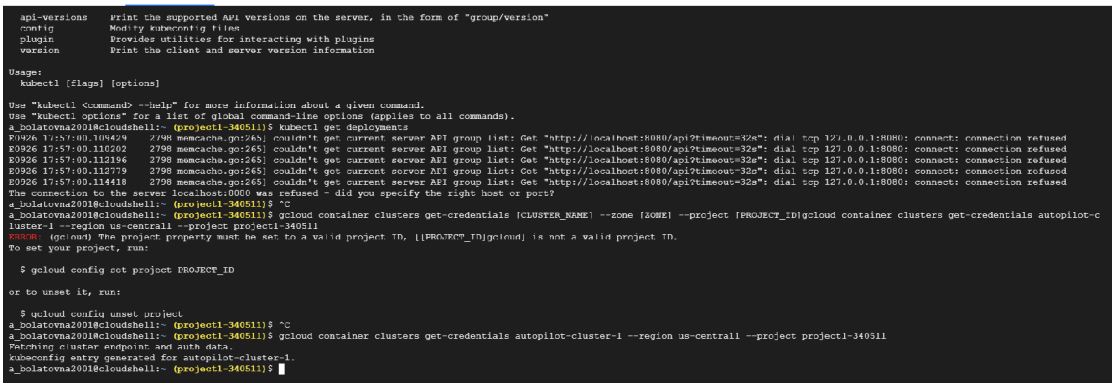


After reviewing the settings, I clicked “Create” to launch the cluster.



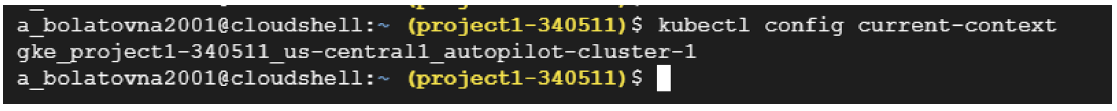
- How did you verify that your application was successfully deployed and accessible?
  Open terminal Google Cloud Shell and I connected to GKE cluster

gcloud container clusters get-credentials autopilot-cluster-1 --region us-central1 --project project1-340511



After running the command, I checked the current context

kubectl config current-context:



Used the command

kubectl create deployment azhardeployment --image=gcr.io/340511/azhar

to create a new deployment

To see the status of your deployments,I runned:

kubectl get deployments

```
a_bolatovna2001@cloudshell:~ (project1-340511)$ kubectl get deployments
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
azhardeployment   0/1     1            0           3m38s
a_bolatovna2001@cloudshell:~ (project1-340511)$
```

Next, I checked the service associated with application to see how it's exposed to the internet.

```
a_bolatovna2001@cloudshell:~ (project1-340511)$ kubectl get services
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   34.118.224.1   <none>        443/TCP   62m
a_bolatovna2001@cloudshell:~ (project1-340511)$
```

I identified that it is ClusterIP: This type is only accessible within the cluster and won't work for external access. So i decided create new service with LoadBalancer.

kubectl expose deployment azhardeployment --type=LoadBalancer --name=kubernetes2 --port=80 --target-port=8080

```
a_bolatovna2001@cloudshell:~ (project1-340511)$ kubectl expose deployment azhardeployment --type=LoadBalancer --name=kubernetes2 --port=80 --target-port=8080
service/kubernetes2 exposed
a_bolatovna2001@cloudshell:~ (project1-340511)$
```

Check again services and get kubernetes2

```
service/kubernetes2 exposed
a_bolatovna2001@cloudshell:~ (project1-340511)$ kubectl get services
NAME          TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)        AGE
kubernetes    ClusterIP      34.118.224.1    <none>          443/TCP        4m32s
kubernetes2   LoadBalancer   34.118.232.129  34.41.112.201   80:30668/TCP   56s
a_bolatovna2001@cloudshell:~ (project1-340511)$
```

So my external IP is 34.41.112.201 and can use to access application.

Cloud Monitoring and Logging

Monitoring was set up to ensure stable operation of the model. Logs tracking API calls were enabled in Cloud Logging like in Figure 2.8,

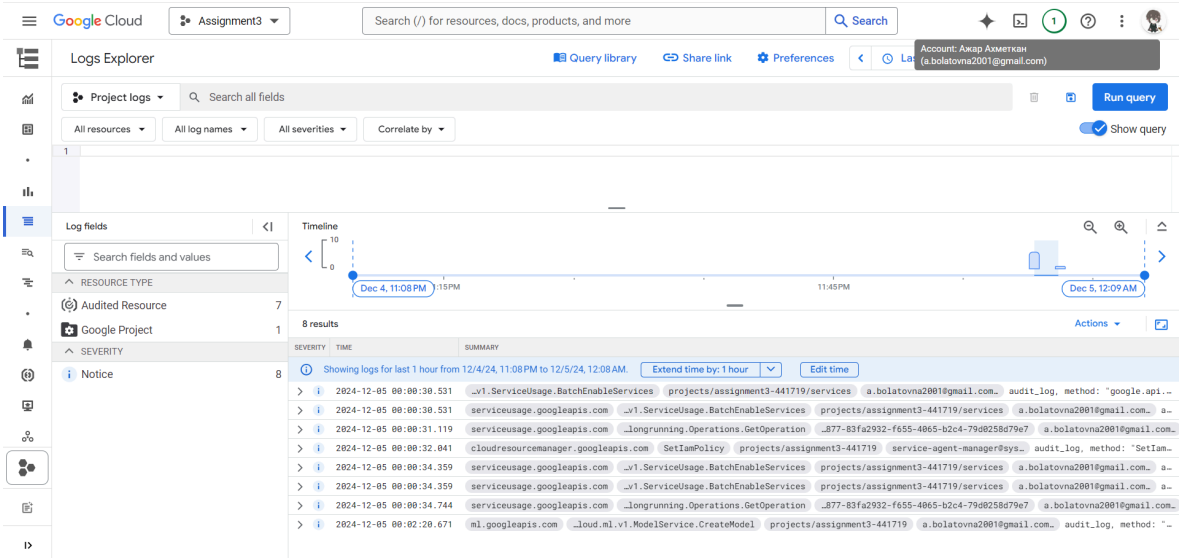


Figure 2.8 Logs Explorer

and dashboards were created in Cloud Monitoring to analyze performance metrics such as response time and percentage of successful requests. Notifications were set up to notify about exceeding critical metric values.

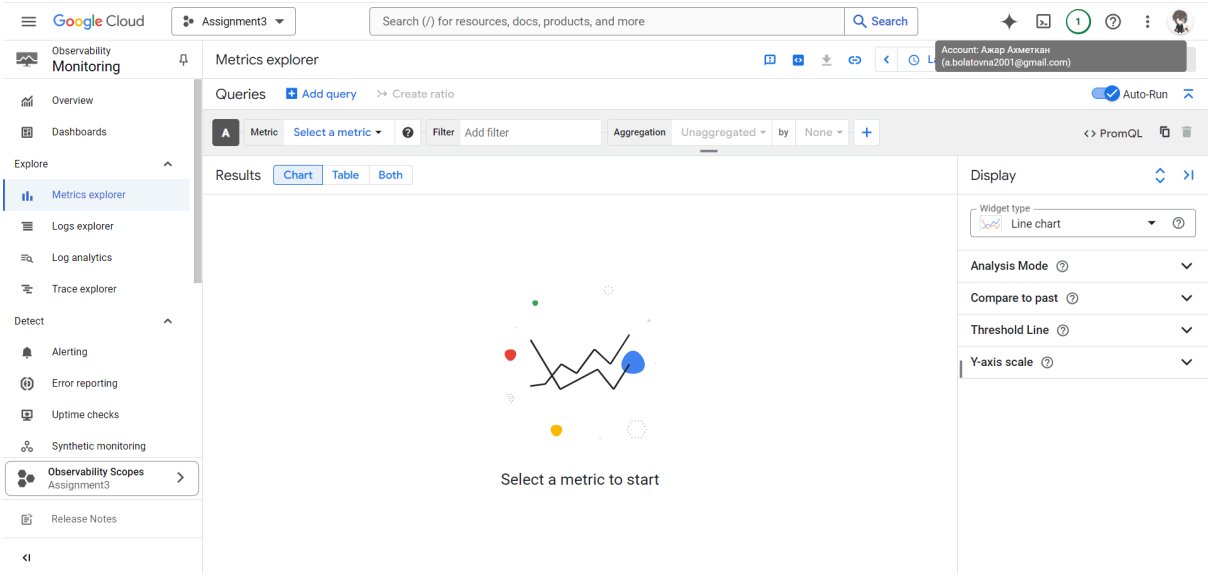


Figure 2.9 Metrics Explorer

Big Data and Machine Learning Integration

Overview of the Pipeline

This project implemented a complete data processing and machine learning pipeline on the Google Cloud platform. The pipeline includes data collection, uploading to cloud storage, processing using BigQuery, training a machine learning model, and deploying the model for use in production. Particular attention is paid to monitoring and logging to ensure stability and performance control. To go to BigQuery you first need to go to your Google Cloud account, where the main page will open as shown in Figure 1.1.

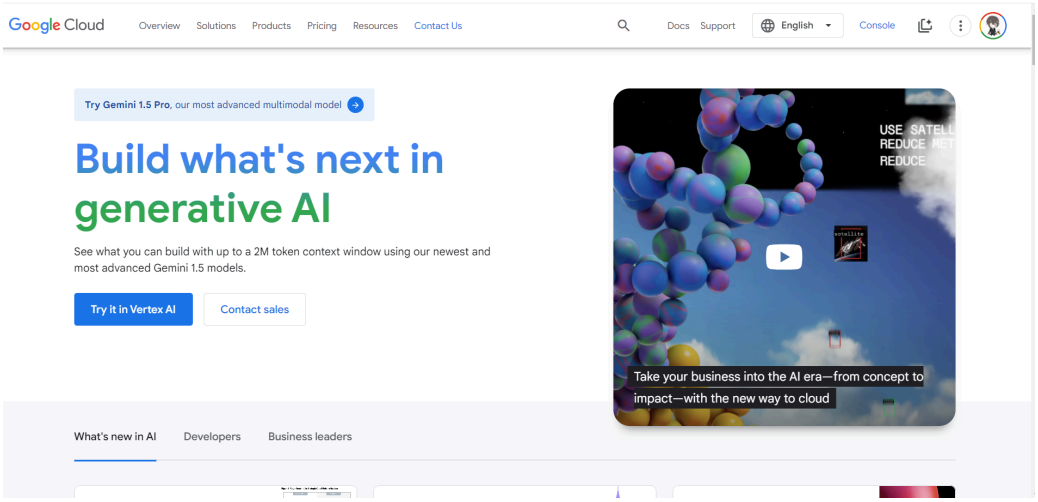


Figure 1.1. Google Cloud's home page

In the upper right corner you will see a button called "Console", which is shown in Figure 1.2, which will move your page to the Google Cloud console.

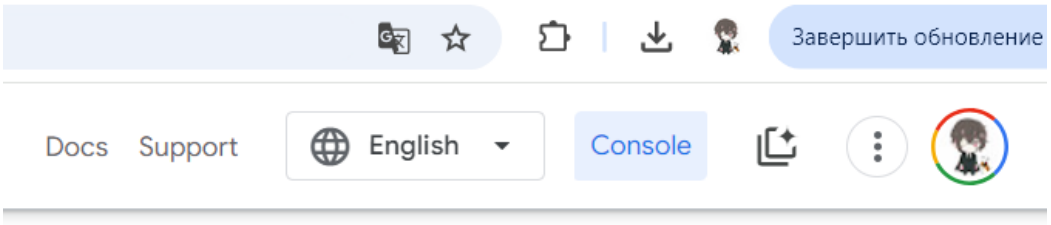


Figure 1.2. "Console" button

The main page will open in front of you with many features of the Google Cloud Console. However, we will need to find the "BigQuery" icon among them to start working with it, which is shown in Figure 1.3.

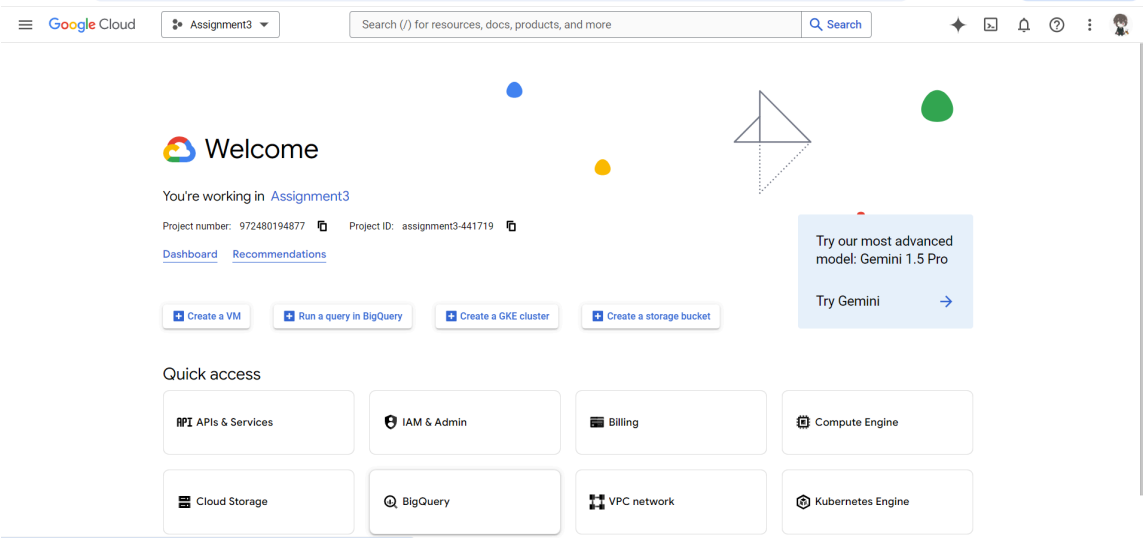


Figure 1.3. Google Cloud Console's home page

In the opened BigQuery's window shown in Figure 1.4, you can start working with the task.

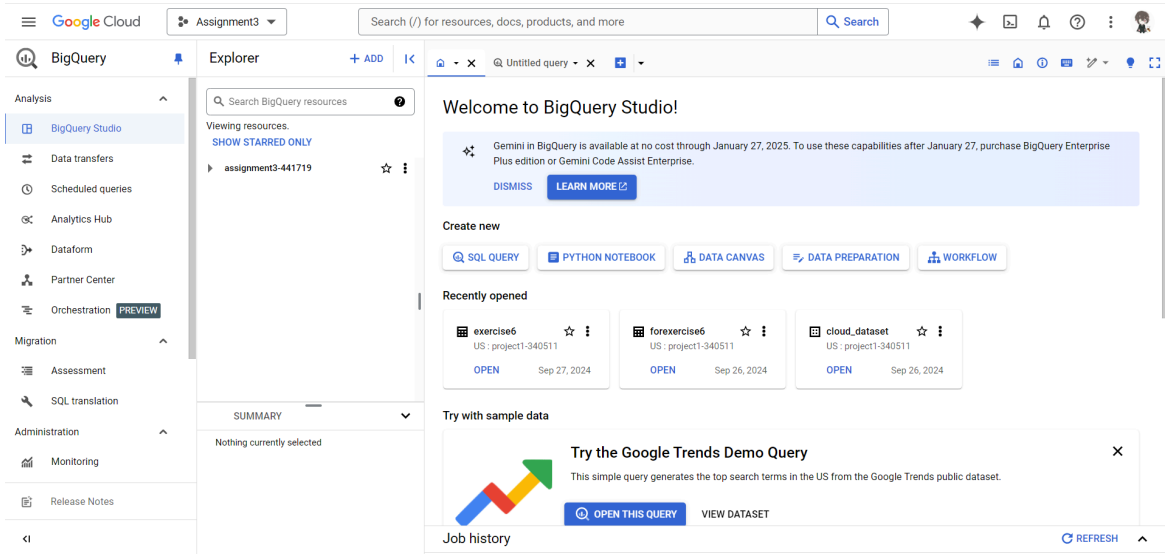


Figure 1.4. BigQuery's home page

Data Ingestion and Processing

The dataset used in the project contains physicochemical characteristics and quality assessments of Portuguese Vinho Verde wines. It consists of 12 variables:

1. Physicochemical characteristics (input variables):
    - fixed acidity
    - volatile acidity
    - citric acid
    - residual sugar
    - chlorides
    - free sulfur dioxide
    - total sulfur dioxide
    - density
    - pH

- sulphates
- alcohol

2. Quality (target variable):
    - quality — wine quality assessment on a scale from 0 to 10.

Purpose of using the dataset is transform the problem into a binary classification: divide wines into "good" (score above 6.5) and "bad".

To build a machine learning model to predict wine quality based on its physicochemical characteristics.

To carry out the wine quality data analysis project, all actions were performed on the Google Cloud platform. First, a project was created in the Google Cloud Console in Figure 1.1. In opened window as in Figure 1.3 to do exercise, the "Select a project" menu was selected at the top of the console, where the "New Project" button was pressed as in Figure 1.5.

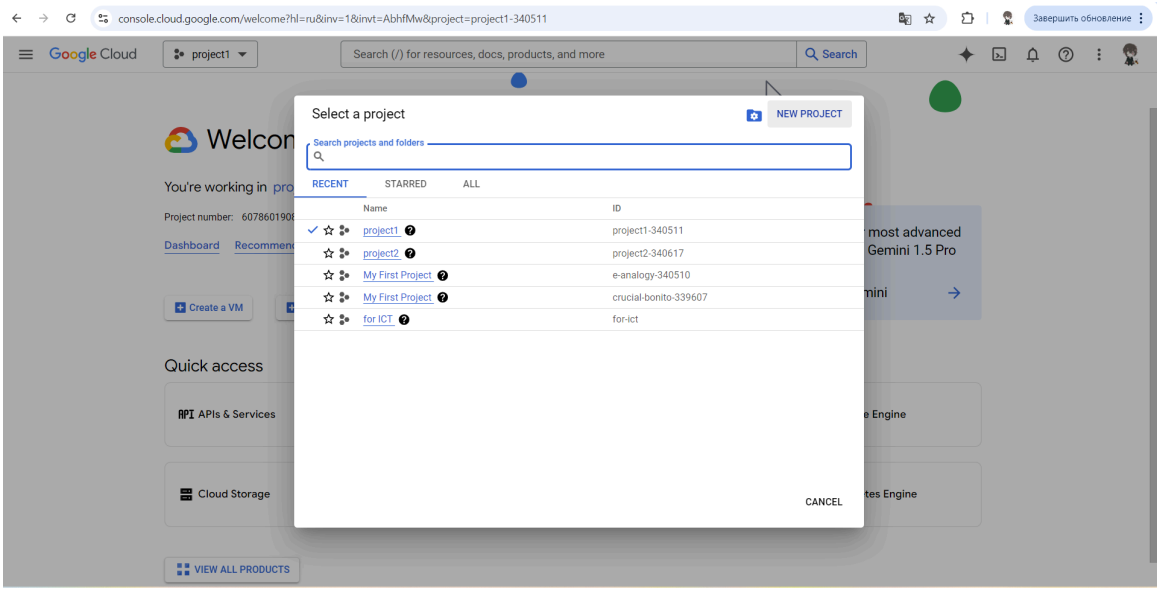


Figure 1.5 List of projects

A window will open where you will need to give a name to the new project and make some settings on how to specify the organization and click “CREATE” button for creation, which figured in Figure 1.7.

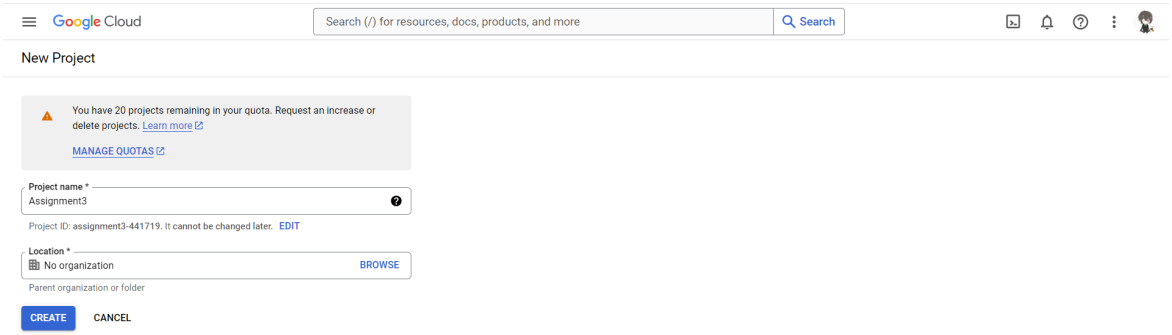


Figure 1.7 Project creation window

After creating the project, as showed in Figure 1.8 the necessary APIs were enabled via "Navigation Menu" → "APIs & Services" → "Library" in Figure 1.9.

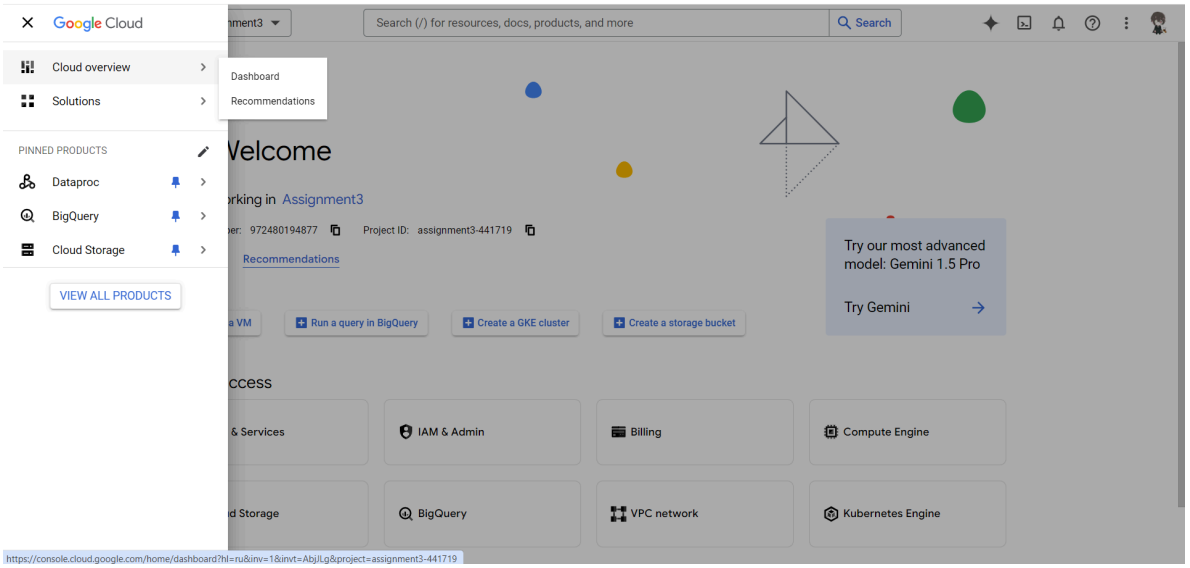


Figure 1.8 Navigation Menu

Here, the BigQuery API, Cloud Storage API, and AI Platform API are activated, which are necessary for working with data and training the model.

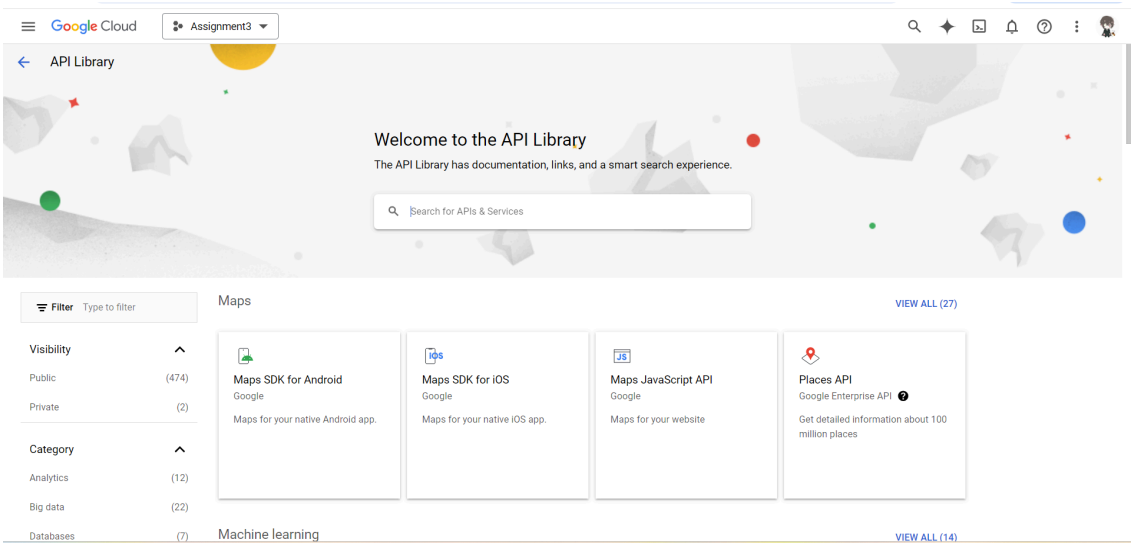


Figure 1.9 Library

Next, Google Cloud Storage was configured to store data. In the "Navigation Menu" menu, "Storage" → "Browser" was selected, where a new bucket was created as in Figure 2.1.

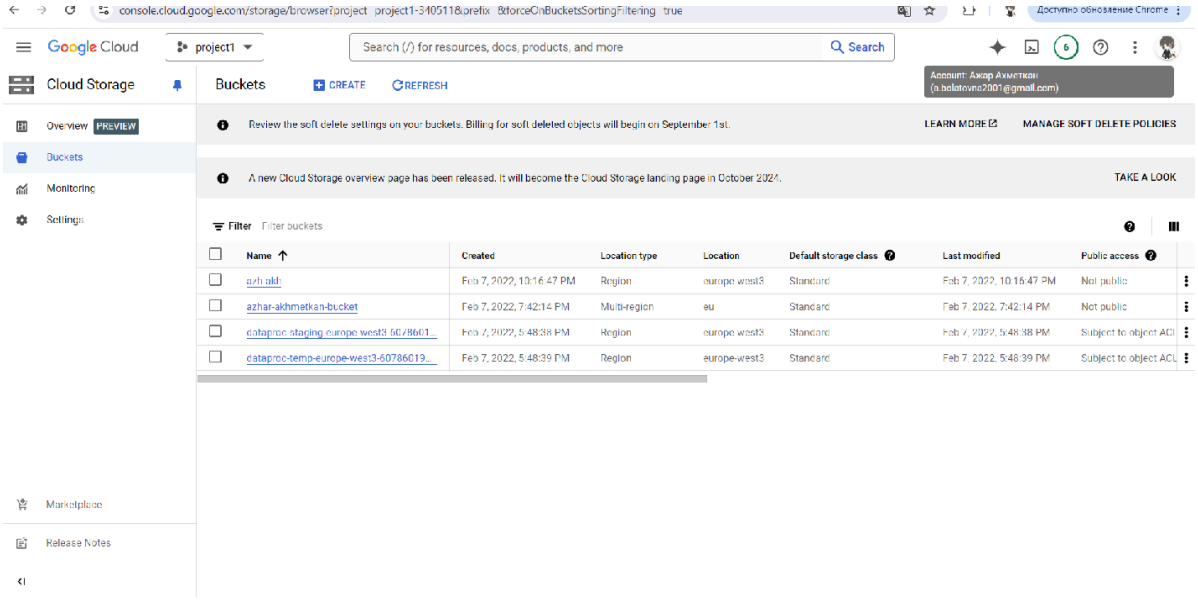


Figure 2.1 Bucket creation

The bucket settings specify the region "US-central" and the storage level "Standard". After the bucket was created, the original file winequality-red.csv was uploaded from the local computer via the "Upload Files" button inside the bucket as in Figure 2.2.

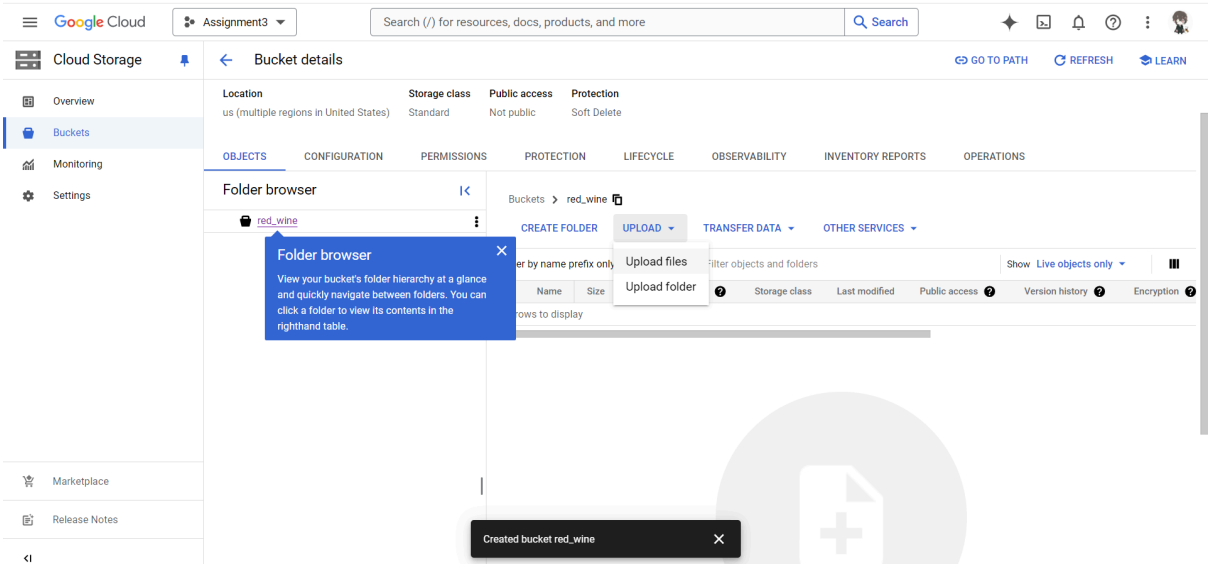


Figure 2.2 Upload File

For further analysis, the data from the bucket was imported into BigQuery. First, a new wine_quality dataset was created in BigQuery by selecting "Create Dataset" in the left panel of BigQuery as in Figure 2.3.

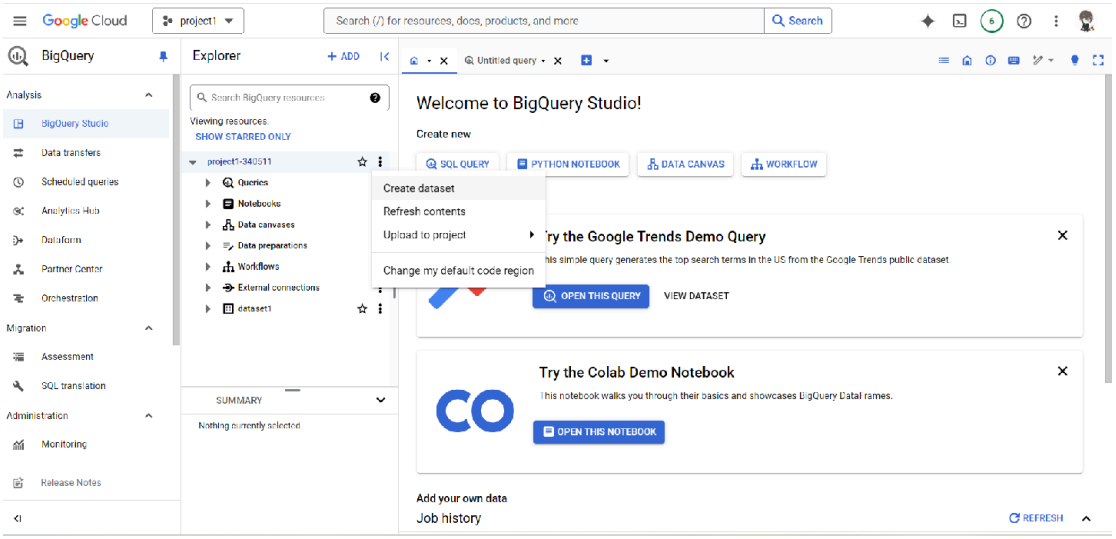


Figure 2.3 Create Dataset

After that, a table was created via "Create Table", where "Google Cloud Storage" was selected as the data source in Figure 2.4. The path to the file was specified in the format gs://wine-quality-dataset/winequality-red.csv, and for ease of use, the "Auto detect" option was enabled to automatically determine the data scheme.

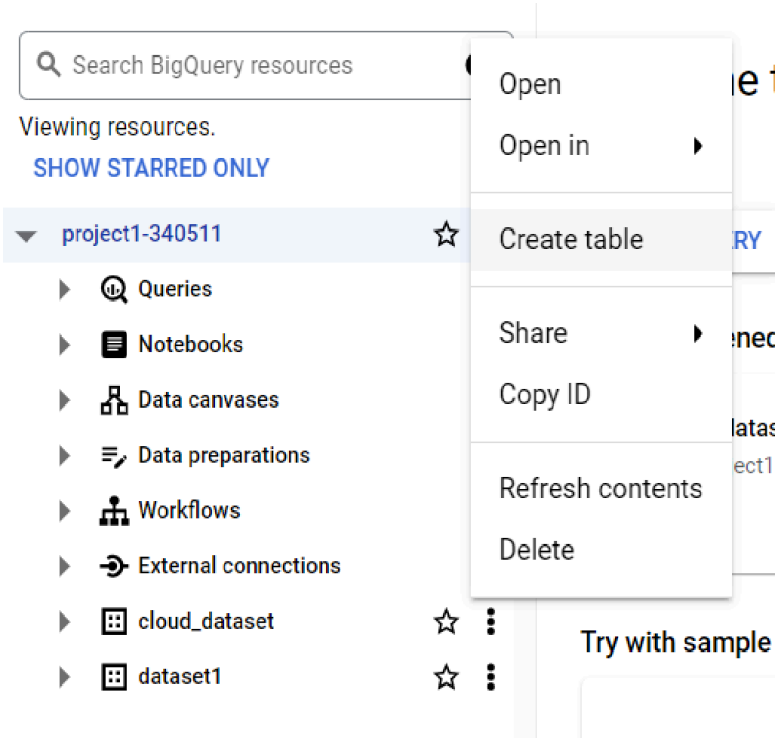


Figure 2.4 Create Table

After loading the data, processing began in BigQuery. SQL queries were executed via the "Query Table" available in the table interface. To clean the data, rows with missing values were removed, and the target variable quality was converted into a category. Values above 6.5 were classified as "good", the rest as "not good", using the SQL query:

```
SELECT *,
              CASE WHEN quality > 6.5 THEN 'good'
               ELSE 'not good' END AS quality_label
FROM `wine_quality.table`
```

Machine Learning Model Training

After preparing the data in BigQuery and exporting it, a machine learning model was built using Python. For this task, the Scikit-learn library and the Random Forest algorithm, which is suitable for classification, were used.

The data was split into training and testing sets (e.g. 75% for training and 25% for testing).

```
from sklearn.model_selection import train_test_split

# Convert the 'quality' column into a binary classification problem
# Wines with quality > 6.5 are classified as 'good' (1), else 'not good'
(0)
updated_wine_data['quality_label'] =
updated_wine_data['quality'].apply(lambda x: 1 if x > 6.5 else 0)

# Separate features (X) and target (y)
X = updated_wine_data.drop(columns=['quality', 'quality_label'])
y = updated_wine_data['quality_label']

# Split data into training and test sets (75% train, 25% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
random_state=42)
```

The Random Forest algorithm was used to build the model. It can effectively work with a large number of features and does not require data normalization.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report
# Create and train the Random Forest model
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)
```

The model performance was evaluated using the following metrics:

- Accuracy: Total number of correct predictions.
- Recall: The proportion of correctly classified objects among all objects of a given class.
- F1 Score: Harmonic mean between accuracy and recall.

```
# Evaluate the model on the test set
y_pred = model.predict(X_test)
evaluation_report = classification_report(y_test, y_pred, output_dict=True)
# Convert evaluation metrics to a DataFrame for better readability
evaluation_metrics = pd.DataFrame(evaluation_report).transpose()
# Display the evaluation metrics for reporting purposes
import ace_tools as tools; tools.display_dataframe_to_user(name="Evaluation
Metrics for the Model", dataframe=evaluation_metrics)
```

Model Deployment

After successful training, the model was deployed to Google AI Platform to provide real-time predictions via a REST API.
The model was saved in joblib format for later use.

```
import joblib
# Сохранение модели
joblib.dump(model, 'model.joblib')
```

The saved model.joblib file has been uploaded back to Google Cloud Storage:

1. Go to "Cloud Storage".
2. Select a bucket, for example, wine-quality-dataset.
3. Click "Upload Files" and select the model.joblib file.

Deploying a model to AI Platform:

- In the Google Cloud console, select "AI Platform" → "Models" like in Figure 2.5

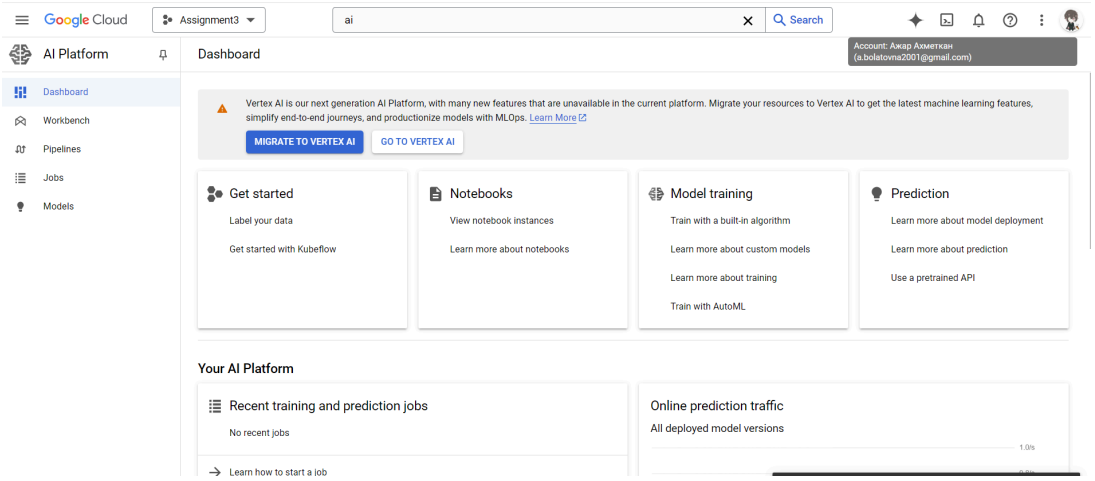


Figure 2.5 AI Platform

- Click "Create Model" and enter a model name, for example, wine_quality_model in Figure 2.6.

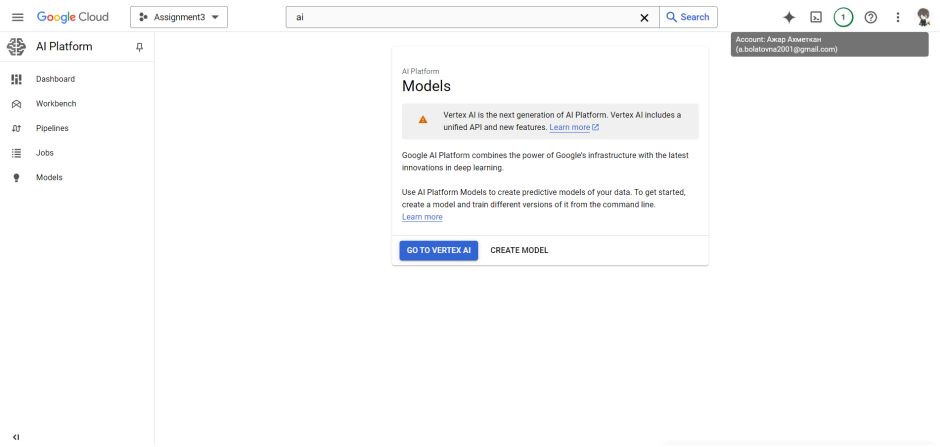


Figure 2.6 Models

- After the model is created, click "Deploy Version" like in Figure 2.7.

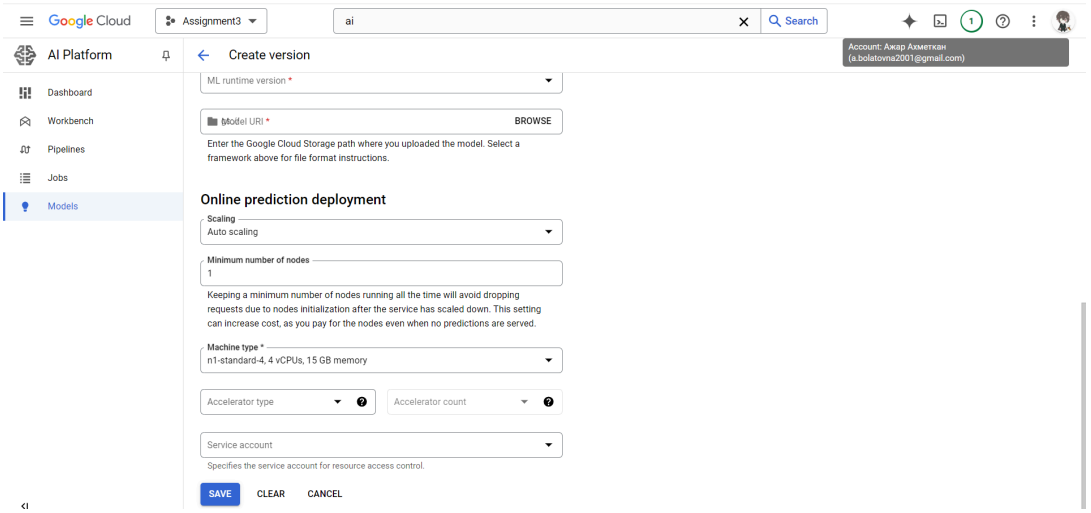


Figure 2.7 Deploy

- Specify the path to the model file in the format gs://wine-quality-dataset/model.joblib.
- Select the required resources (for example, 1 vCPU, 1 GB of memory).
- Click "Deploy".

The endpoint API was used to test the predictions:

```
curl -X POST -H "Content-Type: application/json" \
    -d '{
          "instances": [
              {"fixed_acidity": 7.4, "volatile_acidity": 0.7,
"citric_acid": 0.0, "residual_sugar": 1.9,
               "chlorides": 0.076, "free_sulfur_dioxide": 11.0,
"total_sulfur_dioxide": 34.0,
               "density": 0.9978, "pH": 3.51, "sulphates": 0.56, "alcohol":
9.4}
          ]
        }' \
    https://<YOUR_ENDPOINT_URL>/predict
```

Challenges and Solutions

Challenge 1: Setting Up Google Cloud Authentication

- **Problem:** During authentication using gcloud auth login, network restrictions or browser settings occasionally caused issues in opening the login window or completing the authentication process.
- **Solution:** Used the --no-launch-browser flag with the gcloud auth login command to manually copy the authentication link into a browser. This ensured successful login even when the default browser failed to launch.

Challenge 2: Docker Image Configuration

- **Problem:** While creating the Dockerfile for the application, missing dependencies or incorrect configurations caused the container to fail during runtime.
- **Solution:** Carefully reviewed error logs and updated the Dockerfile to include all necessary dependencies and configurations. Added health checks to ensure the container was running correctly.

Challenge 3: Pushing Images to Google Container Registry (GCR)

- **Problem:** Permissions were not correctly set, which resulted in failures when pushing the Docker image to GCR.
- **Solution:** Configured IAM roles to ensure the account used had the required roles/storage.admin permission. Additionally, authenticated Docker with GCR using the gcloud auth configure-docker command.

Challenge 4: Kubernetes Configuration Issues

- **Problem:** During deployment, Kubernetes pods failed to connect to the backend due to incorrect environment variable configuration or missing network settings.
- **Solution:** Used kubectl logs and kubectl describe pod to debug errors. Updated deployment YAML files to include proper environment variables and ensured that the container ports matched the service configuration.

Challenge 5: Exposing the Application

- **Problem:** The LoadBalancer service in GKE did not assign an external IP address, making the application inaccessible.
- **Solution:** Ensured that the GCP project had sufficient quota for LoadBalancer resources. Also, verified that the firewall rules allowed incoming traffic on the necessary ports (e.g., 80/443 for HTTP/HTTPS).

These challenges highlighted the importance of careful configuration, debugging, and ensuring proper permissions throughout the project. The implemented solutions ensured the successful deployment and operation of the application.

Conclusion

The project successfully demonstrated the deployment and management of a containerized application using Google Kubernetes Engine. Key objectives, such as creating Docker images, pushing them to Google Container Registry, and deploying them on a Kubernetes cluster, were achieved.

Despite challenges encountered during authentication, Docker configuration, GCR integration, and Kubernetes deployment, effective troubleshooting and systematic solutions ensured the successful implementation of the application. The use of Google Cloud services, including Cloud Storage, GCR, and GKE, provided a reliable and scalable platform for the project.

This project underscored the importance of leveraging modern cloud infrastructure for efficient application deployment and management, highlighting Google Cloud's robust capabilities in simplifying complex tasks. These learnings and solutions lay a strong foundation for future enhancements and more advanced deployments.

References

- Docker. (n.d.). *Get started with Docker*. Retrieved [Insert Date], from https://docs.docker.com/get-started/
- Flask. (n.d.). *Flask quickstart*. Retrieved [Insert Date], from https://flask.palletsprojects.com/en/latest/quickstart/
- Google Cloud. (n.d.). *Cloud storage overview*. Retrieved [Insert Date], from https://cloud.google.com/storage/docs/overview
- Google Cloud. (n.d.). *Google container registry*. Retrieved [Insert Date], from https://cloud.google.com/container-registry/docs
- Google Cloud. (n.d.). *Google Kubernetes engine overview*. Retrieved [Insert Date], from https://cloud.google.com/kubernetes-engine/docs
- Kubernetes. (n.d.). *Kubernetes API reference*. Retrieved [Insert Date], from https://kubernetes.io/docs/reference/kubernetes-api/
- YAML. (n.d.). *YAML language reference*. Retrieved [Insert Date], from https://yaml.org/spec/1.2/spec.html

Appendices

Appendix A: Dockerfile Example

Below is an example of the Dockerfile used to create the containerized web application:

```
FROM python:3.9-slim
WORKDIR /app
COPY . /app
RUN pip install flask
EXPOSE 8080
CMD ["python", "app.py"]
```

Appendix B: Sample Deployment YAML

The following YAML file demonstrates the configuration for deploying the application to a Kubernetes cluster:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web-app
  template:
    metadata:
      labels:
        app: web-app
    spec:
      containers:
      - name: web-app
        image: gcr.io/<project-id>/web-app:v1
        ports:
        - containerPort: 8080
```

Appendix C: Sample Service YAML

This YAML file exposes the application to the internet using a LoadBalancer:

```
apiVersion: v1
kind: Service
metadata:
  name: web-app-service
spec:
  type: LoadBalancer
```

```
  selector:
    app: web-app
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```

Appendix D: Commands Used in Google Cloud Shell
Below are key commands executed during the project:
Build Docker Image:

```
docker build -t gcr.io/<project-id>/web-app:v1 .
```

Push Docker Image to GCR:

```
docker build -t gcr.io/<project-id>/web-app:v1 .
```

Authenticate GKE and Configure kubectl:

```
gcloud container clusters get-credentials web-app-cluster
--region <region>
```

Deploy the Application:

```
kubectl apply -f deployment.yaml
```

Expose the Service:

```
kubectl apply -f service.yaml
```

View External IP:

```
kubectl get services
```

Appendix E: Troubleshooting Steps

- **Authentication Issues:**
    - Used --no-launch-browser flag with gcloud auth login for manual browser-based login.
- **Debugging Kubernetes Pods:**

Commands:

```
 kubectl logs <pod-name>
kubectl describe pod <pod-name>
```

- **Fixing LoadBalancer Issues:**
    - Ensured sufficient resource quotas and verified firewall rules in Google Cloud Console.

These appendices provide supplementary details to support the main report and serve as a reference for recreating the deployment process.