Hotel Coupon App: Leveraging Cloud Technologies for a Scalable Solution

Alexander Mamani Yucra
Cloud Platform Programming
Master of Science in Cloud Computing
December 2024

***Abstract*—Cloud computing has changed how applications are built, offering flexible, efficient, and cost-effective solutions for modern software needs. The Hotel Coupon App is a case study that looks at how serverless and cloud-based architectures can be used to create an interactive, and real-time platform.**

The app allows hoteliers to create and manage discount coupons while providing users with the tools to search, redeem, and track offers in real time. Traditional application architectures struggle with the demands of real-time notifications and handling large amounts of data. In this case, we show that using a combination of AWS services like Lambda for serverless execution, DynamoDB for fast data storage, and SNS for scalable notifications, providing a solid foundation for addressing these challenges.

Our results show that AWS SQS works well for processing user interaction data in batches, allowing real-time feedback without slowing down the system. Additionally, integrating WebSocket APIs improves user engagement by providing instant updates, offering a significant advantage over older HTTP-based systems. This approach highlights the potential of combining serverless and event-driven architectures for better scalability and performance.

These findings demonstrate how AWS cloud services can reduce the complexity of operations while meeting the requirements of dynamic applications. This case study also shows the value of serverless architectures in driving innovation across industries, providing useful information for developers and companies looking to improve their cloud solutions.

I. Introduction

In the highly competitive hospitality industry, offering promotional discounts and coupons is a key strategy for attracting customers. With the increasing need for digital solutions, both hoteliers and customers are seeking more efficient ways to manage and access deals in real time. Traditional methods of distributing coupons and tracking their use often fall short because they cannot handle large volumes of data, provide immediate feedback, or engage users effectively. This is where cloud computing technologies provide a powerful solution.

The Hotel Coupon App has been designed to address these challenges by offering a cloud-based platform that allows hoteliers to create, manage, and distribute discount coupons. At the same time, it provides users with an easy and engaging way to discover and redeem these offers. The app uses cloud technologies to ensure scalability, and real-time interaction, ensuring a seamless experience for both hoteliers and users.

The motivation for developing the Hotel Coupon App comes from the need to make coupon management easier for hoteliers while improving the customer experience. Hoteliers require a system that not only makes it easy to distribute coupons but also provides valuable insights into how their offers are performing, including who is redeeming them and when. On the other hand, customers expect to access personalized, real-time deals that are quick and simple to use. Current systems often lack the necessary infrastructure to handle these dynamic interactions at scale, which highlights the importance of using modern cloud technologies to meet the increasing demands of both businesses and users.

The main goals of the project are to cover different stages of the Hotel Coupon App development, starting with the Project Requirements in section II. It then moves on to the Architectural Design Aspects in section III. The project also dives into the advantages and disadvantages of the cloud-based services used for each functional and non-functional requirement in section IV. The implementation of each architectural design aspect is explored through a Custom Python Library, with a focus on cloud-based services, in sections V and VI. Additionally, Continuous Integration, Delivery, and Deployment (CI/CD) is discussed in section VII, and the Conclusion of the project is provided in section VIII.

II. Project Requirements

A. Functional Requirements

- Registered users should be able to find coupons by fields such as title, discount, hotel name, ...etc. They also can view detailed information about a specific coupon.
- Registered users should be able to redeem a coupon, and once it is redeemed, they will subscribe to receive notifications about new coupons from that hotelier.
- Registered users should be able to visualise redeemed coupons on another section, use them through their unique coupon code, and be notified when it is used successfully.
- Registered hoteliers should be able to visualise their general coupon information, and what coupons were used. They also can create a new coupon, and the app must notify all subscribed users about that new coupon.
- Registered hoteliers should be able to modify the status of the redeemed coupon from redeemed to used by entering the unique coupon code. Then, the app must update the dashboard of that user taking that coupon off and notifying that it has been used successfully.

- Registered hoteliers should be able to get a custom PDF report, where they can visualise information such as the user interactions with their coupons from a range of dates, and the general status of all their coupons.

B. Non-functional Requirements

- User Experience: User and hotelier notifications should be in real-time to engage them with the app.
- Graphical User Interface Performance: Collecting user data interaction, such as coupon views and redeemed coupons, must not impact the user experience in terms of speed.
- Batch Processing: User interaction data must be captured and temporarily stored. Once the data is processed, it must be disposed of.
- Build and Deployment Automation: New app features must be deployed in an AWS EC2 instance in an automated way, going through stages such as testing, building and deployment.

III. Architectural Design Aspects of The Hotel Coupon App

To build The Hotel App, various AWS services were considered, including storage, databases, messaging, batch processing, and event-driven components. So, it was concluded that the fittest cloud-based services were Simple Storage Service (S3), DynamoDB, Relational Database Service (RDS), Simple Notification Services (SNS), Simple Queue Service (SQS), Lambdas, and Amazon Websockets API. The use of each service is justified in the following features implemented in the Hotel Coupon App.

A. Capture of User Interactions

Due to the massive data user interaction like views and redeems of coupons that one user could generate in only one session. And the fact that it has to be stored and then disposed of as well. It has opted to use SQS for batch processing, Websocket API to send data constantly, and DynamoDB to save the Websocket connection sessions as the users are logged on the system.

The "Fig. 1" below, shows how one user sends their interactions and it goes through Amazon Websocket API, then to Lambda (Collect-user-interaction), and finally it is stored in the AWS SQS Queue.

B. Create and Use Coupons

The actions of creating and using coupons involve various tasks, such as storing them in a database and file system, and sending notifications about them to all subscribed users to the hotelier who owns the coupon.

In order to complete these tasks, it has opted to use RDS to store coupon data, S3 to store multimedia like images, and SNS to publish on a specific topic once the coupon was already created or used. Once the message is ended, the AWS Lambda function is invoked to connect to RDS to know what users are subscribed to the hotelier who owns the coupon,

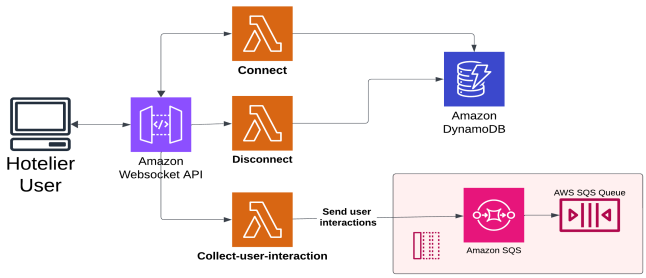

Fig. 1. Architecture diagram for sending notifications and capturing user interactions.

then the Lambda function finds the Websocket connections on AWS DynamoDB and finally notifies all subscribed users. The "Fig. 2" below illustrates the complete flow of the architecture.

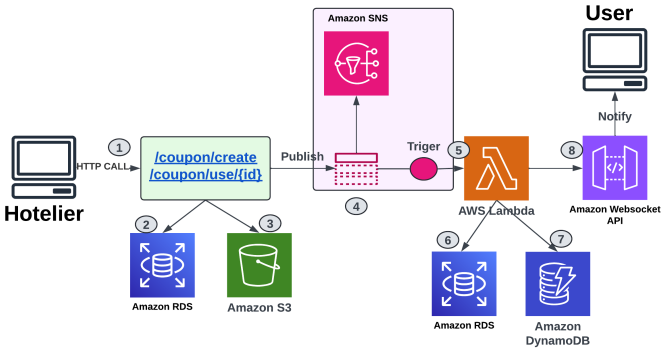

Fig. 2. Architecture diagram for creating and using coupons by hoteliers

C. Generate Reports

The generation of reports is the most complicated feature due to batch processing complexity. To complete this task successfully, it has opted to use SQS. polling the user data interactions and processing them to select which data belongs to a specific hotelier.

Firstly, an HTTP request is sent to the REST API service (Django) by the hotelier. Secondly, user data interactions are polled and processed. As a result, a PDF report is generated and stored in RDS and S3. Once the report is generated, a message is published to an SNS topic to notify hoteliers that their report is ready. Since the data processing can take time, notifications via SNS and WebSocket are suitable for asynchronous requests. The "Fig. 3" illustrates the sequence of tasks.

IV. Advantages and Disadvantages of Cloud-based Services Used

A. Simple Storage Service

One of the advantages of S3 in the project is the scalability, allowing it to scale automatically to handle the growing number of coupon-related media files such as images, and PDF reports. Another advantage of S3 is that it provides 99.99%

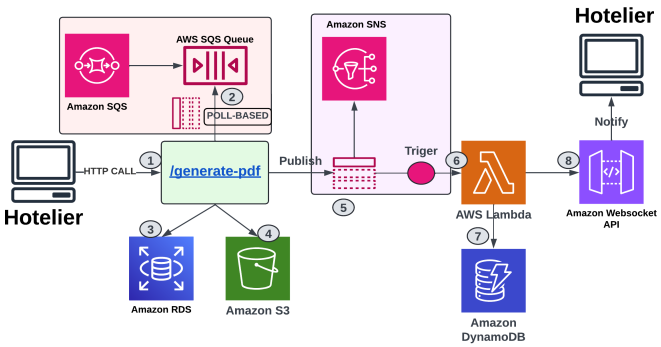

Fig. 3. Architecture diagram for generating PDF reports for hoteliers

availability and 99.999999999% durability, ensuring reliable storage for coupon data [1].

Some disadvantages are complex permissions to access, involving creating IAM roles and bucket policies to access it. It might become a problem to configure it, causing security issues if you are new to AWS Services.

B. Relational Database Service and DynamoDB

Advantages such as familiarity with SQL, data support, and backup and recovery were key factors in choosing RDS as the database for storing business logic, including hotelier and customer profiles, coupon details, used coupon transaction records, and reports. This choice allows for rapid progress with functional requirements while ensuring that crucial data is securely managed and administered by the AWS Support team [2].

Regarding DynamoDB, advantages such as low latency and event-driven integration were taken into account, making it easy to integrate with Lambda services [3]. This allows the separation of business logic from the user experience (non-functional requirements), making it suitable for real-time data such as user interactions.

One disadvantage of RDS is its higher costs, which are reflected in the instance type chosen for the project (db.t4g.medium), exchanging safety and support over performance.

Some disadvantages of DynamoDB include complex data modelling, a steep learning curve, and limited querying. The first two directly impact the start of the project, making it a bit difficult to get started, but they can eventually be managed. The last one requires designing queries around partition and sort keys.

C. Simple Queue Service

Advantages such as scalability, flexibility and being fully managed by AWS were key factors in choosing SQS. Its ease of handling high message loads and processing batches of data made it the perfect choice. making the stack grow at the demand of user interactions, then releasing already processed data [4].

Some disadvantages are limited messages size, allowing up to 256 KB message size, and messages may not be processed in the exact order, leading to possible delays.

D. Simple Notification Service

The most relevant advantages of SNS in the project are scalability and asynchronous messaging. This allows pushing messages to topics in real-time, handling large volumes of notifications. It is also suitable because from a topic, the app can implement various types of notifications, with WebSockets being one option to alert users and hoteliers. Additionally, it can integrate email or SMS notifications using third-party libraries [5].

One disadvantage of SNS is the eventual consistency, where messages may not be delivered instantly or in order, causing potential delays in time-sensitive notifications.

E. Lambdas (Serverless)

The event-driven advantage was a key factor in implementing the Hotel Coupon app, as it facilitated integration with other AWS services like DynamoDB, RDS, SQS and SNS to trigger processes such as sending notifications or processing coupons. Other advantages included cost efficiency and backend simplification. As it helped to reduce costs by only charging for execution time and eliminates the need for server management, reducing operational implementation time [6].

The clearest disadvantage of Lambda is complex debugging, with functions being challenging to debug, especially in a distributed environment. Another disadvantage is the runtime limit, which is limited to 15 minutes per function, which may not be suitable for long-running tasks.

V. Custom Python Library

One of the key factors for an application to be successful is not only that it performs properly its functions, but also that it is able to present its information to users in an attractive and clear way. So, for the Hotel Coupon App was important to make focus in those aspects, processing the user data interactions in a manner properly, generating an attractive report and delivering it on an interactive way.

The main goal of the library is to expose a custom programming interface so that the REST-API service (Django) can interact with AWS services (SNS and SQS) and generate reports transparently. Additionally, AWS Lambda functions can also interact without problem with AWS services through this library. The library is available in http://pypi.org: here.

A. SNS Library Functionalities

The `SNSService` Class allows the backend to connect with AWS SNS Service. Its constructor receives parameters such as `AWS_ACCESS_KEY`, `AWS_SECRET_ACCESS_KEY`, and `AWS_REGION`. This class also have the method publish_message, which allows to send publish a message on the topic, it receives `SNS_ARN` as a parameter. The

library also exposes a custom Exception Class called SNSPublishMessageError to handle errors when a message is being published.

The “Fig. 4” illustrates the implementation of the SNS library when one coupon is used by a hotelier role and a message has to be published so the user knows about that.

```python
def retrieve(self, request, *args, **kwargs):  # Alexander Mamani Yucra *
    try:
        queryset = self.get_object()
        serializer = self.get_serializer(*args queryset, context={'request': request})
        user_profile_id = str(queryset.user_profile_id.id)
        coupon_code = str(queryset.id)
        message = { "user_profile_id": user_profile_id, "coupon_code": coupon_code}

        env = environ.Env()
        AWS_ACCESS_KEY_ID = env('AWS_ACCESS_KEY_ID')
        AWS_SECRET_ACCESS_KEY = env('AWS_SECRET_ACCESS_KEY')
        AWS_REGION = env('AWS_REGION')
        AWS_SNS_USED_COUPON_NOTIFICATION_ARN = env('AWS_SNS_USED_COUPON_NOTIFICATION_ARN')

        sns_service = SNSService(aws_access_key=AWS_ACCESS_KEY_ID, aws_secret_key=AWS_SECRET_ACCESS_KEY, region_name=AWS_REGION)
        try:
            sns_service.publish_message(AWS_SNS_USED_COUPON_NOTIFICATION_ARN, message, subject="Report Used Coupon Notification")
        except SNSPublishMessageError as e:
            raise SNSPublishMessageError(f"Error when it was publishing messages: {e}")
        return Response(serializer.data)
    except Exception as e:
        return Response(str(e), status=400)
```

Fig. 4. Implementation of SNS library to alert one user that their coupon was used

B. SQS Library Functionalities

The `SQSService` Class allows the backend and AWS lambda to send messages to AWS Queue and poll data from it. Its constructor receives the same parameters that the `SNSService` constructor plus `AWS_SQS_QUEUE_URL` parameter.

In order to poll messages from the AWS SQS Queue, A handler function must be implemented and passed to method `poll_messages`. This handler function is in charge of processing the data from the AWS Queue and, it is invoked for each message polled from the AWS SQS Queue. This handler function must return true if the message was processed properly and it needs to be deleted from the AWS SQS Queue, and False if that message does not need to be deleted. The “Fig. 5” illustrates the implementation of polling messages.

```python
class GenerateReportPDFView(APIView):  2 usages  Alexander Mamani Yucra *
    def get(self, request, *args, **kwargs):  Alexander Mamani Yucra *
        hotelier_authenticated = HotelierProfile.objects.get(user=self.request.user)
        hotelier_coupons = Coupon.objects.filter(hotelier_profile=hotelier_authenticated)
        hotelier_coupon_ids = [str(coupon.id) for coupon in hotelier_coupons]
        buffer_processed_data = []

        env = environ.Env()
        AWS_ACCESS_KEY_ID = env('AWS_ACCESS_KEY_ID')
        AWS_SECRET_ACCESS_KEY = env('AWS_SECRET_ACCESS_KEY')
        AWS_REGION = env('AWS_REGION')
        AWS_SQS_QUEUE_URL = env('AWS_SQS_QUEUE_URL')
        from_date_report = datetime.datetime.now().date()

        sqs_queue_instance = SQSService(aws_access_key_id=AWS_ACCESS_KEY_ID,
                                        aws_secret_access_key=AWS_SECRET_ACCESS_KEY, aws_region=AWS_REGION, aws_sqs_queue_url=AWS_SQS_QUEUE_URL)

        handler_with_buffer = partial(handler_to_get_data_for_a_specific_hotelier_id, buffer_data=buffer_processed_data,
                                      hotelier_coupon_ids=hotelier_coupon_ids, from_date_report=from_date_report)
        try:
            sqs_queue_instance.poll_messages(handler_with_buffer, target_message_count=30)
            sqs_queue_instance.close()
        except (SQSPollingMessagesError, SQSClosingConnectionError) as e:
            raise SQSPollingMessagesError(f"Error when it was polling messages: {e}")
```

Fig. 5. Implementation of SQS library to poll user data interactions on Django Service

The action of sending messages to the AWS SQS Queue is carried out by a method called `send_message` on the AWS lambda function. This method receives a dictionary object as a message, and is it parsed to a string to send the AWS SQS Queue into the `send_message` method. The “Fig. 6” illustrates the implementation of sending messages to the AWS SQS Queue. As the SNS Library, the SQS Library exposes Custom Exception Classes to handle the errors when the messages are polling and sending.

```python
import json
import os
from hotel_coupon_app_package_alexandermamani.aws_services import SQSService,SQSSendMessageError,
                                                                  SQSClosingConnectionError

def lambda_handler(event, context):
    sqs_queue_instance = SQSService(aws_sqs_queue_url=os.environ['AWS_SQS_QUEUE_URL'])
    message = json.loads(event['body'])['message']

    data = {}
    data['coupon_id'] = message['coupon_id']
    data['action'] = message['action']
    data['user_profile_id'] = message['user_profile_id']
    data['country'] = message['country']
    data['date'] = message['date']

    try:
        sqs_queue_instance.send_message(data)
        sqs_queue_instance.close()
    except (SQSSendMessageError, SQSClosingConnectionError) as e:
        raise ("Error SQS", e)

    return {}
```

Fig. 6. Implementation of SQS library to send user data interactions on a lambda function

C. Report Library Functionalities

The `ReportPDF` class generates a custom PDF report, highly crucial information for hoteliers such as the general status of their coupons, and the user interactions, including views and redemptions. This information is vital for hotel decision-making, allowing them to identify their target market and determine which coupons have the greatest impact.

To use this class, data such as `user_interaction_data` and `general_coupon_information` must be passed to the ReportPDF constructor. Once this is done, the generate method should be invoked to obtain a PDF buffer, which can then be saved to a database. The “Fig. 7” illustrates the implementation of generating a custom report.

```python
for data in buffer_processed_data:
    if not data['coupon_id'] in user_interaction_data:
        user_interaction_data[data['coupon_id']] = {}
        user_interaction_data[data['coupon_id']]['view'] = 0
        user_interaction_data[data['coupon_id']]['redeem'] = 0
        coupon_target = Coupon.objects.get(id=str(data['coupon_id']))
        user_interaction_data[data['coupon_id']]['coupon_title'] =  str(coupon_target.title)

    user_interaction_data[data['coupon_id']][data['action']] += 1

hotelier_name = str(hotelier_authenticated.name)
report = ReportPDF(user_interaction_data, coupon_gral_information, datetime.datetime.now().date(), hotelier_name)
pdf_buffer = report.generate()

pdf_filename = f"{datetime.datetime.now().isoformat()}.pdf"

# Create a new ReportTime instance
report_name = current_day.isoformat() + "-" + hotelier_name + "-Report"
report_instance = Report(hotelier_profile=hotelier_authenticated, title=report_name)
report_instance.media_url.save(pdf_filename, ContentFile(pdf_buffer.read()), save=True)

# PUSH to SNS
```

Fig. 7. Implementation of Report library to generate PDF report on Django service

VI. Implementation

This section will resume the Project Requirements tackled in Section II. Diving into each Functional and Non-functional requirement and how it relates to Cloud-based services.

A. Storing - S3

For functional requirements such as creating and visualising Coupon details, downloading PDF Reports to get coupon information, and storing user and hotelier profile pictures; it is important to have a good file system, always avoiding delegating those tasks to the app server production. For this scenario, AWS S3 is suitable. As User Cloud we do not have to care about that infrastructure because we assume that always be available and supported by the AWS Team. Guarantee good performance when users and hoteliers navigate around the Hotel Coupon App.

In the architectures “fig. 2” and “fig. 3” we can look at how actions such as creating and generating reports use AWS S3 to store multimedia files.

B. Database - RDS and DynamoDB

Functional requirements such as storing user, hotelier and coupon information, and using it to filter are covered by the RDS. This allows us to keep a strong relationship between tables and do complex queries.

On the other hand, Non-functional requirements like getting a real-time notification when a coupon is released, used or a report is ready to involve storing websocket connections. DynamoDB helps us to store these connections and access them quickly, which is important for User Experience Non-functional requirements. The “fig. 8” illustrates how Socket Connections are retrieved from DynamoDB to send a notification to the right user when their coupon is used.

```
paginator = dynamodb.get_paginator('scan')
connectionIds = []
for page in paginator.paginate(TableName=os.environ['USER_TABLE']):
    users_connected = page['Items']
    for user_connected in users_connected:
        if user_profile_id == user_connected['user_profile_id']['S']:
            connectionIds.append(user_connected)
for connectionId in connectionIds:
    apigatewaymanagementapi.post_to_connection(
        Data=json_data,
        ConnectionId=connectionId['connection_id']['S']
    )
return {}
```

Fig. 8. Implementation of DynamoDB to get WebSocket connections on AWS Lambda

C. Serverless Computing - AWS Lambda

AWS Lambda was the key for executing tasks in isolation, its big support to integrate multiple AWS services in one function helped a lot to carry out complex tasks.

Functional requirements such as collecting user interactions and notifying users and hoteliers were implemented in it. It was also used to administer WebSocket connection, storing and removing from DynamoDB for the user-experience non-functional requirement. The “fig. 9” lists the different tasks that Lambda carry out.

Function name	Description
notify-hotelier-report	An Amazon SNS trigger that notifies the hoteliers when their report is ready
new-hotel-coupon-connect	Lambda that store the websocket connection on DynamoDB
notify-user-about-used-coupon	An Amazon SNS trigger that notifies the user when their coupon has already been used
new-hotel-coupon-disconnect	Lambda that remove the websocket connection from DynamoDB
new-hotel-collect-data-user-interaction	Lambda that send user interactions to AWS SQS
notify-about-new-coupon-to-users	An Amazon SNS trigger that notifies the user when a new coupon has been released

Fig. 9. List of AWS Lambda functions

D. Notificacion Service - SNS

Simple Notification Service was a key factor in implementing the notification functional requirement between users and hoteliers because it helped to share messages between Django Service and AWS Lambdas. Having a communication App to App (A2A) based on publish/subscribe messaging. It also makes the notification module scalable because we can implement multiple Lambda functions subscribed to SNS topics, where each one could implement different ways to notify users and hoteliers like SMS, Email, and others.

Steps 4 and 5 of the “fig. 2” and steps 5 and 6 of the “fig. 3” illustrate how SNS is used as A2A messaging, where Django service publishes a message and it triggers a subscribed function lambda.

E. Queue Service - SQS

The functional requirement of generating a PDF report and the non-functional requirement of Batch Processing involve sub-tasks such as capturing, polling, processing, and deleting user interaction data. One of the biggest challenges in terms of saving storage is that not all of this data is necessarily relevant to store in a database, as it only needs to be processed once and then discarded.

AWS Simple Queue Service (SQS) played a key role in implementing these tasks due to its producer/consumer architecture, making it well-suited for collecting and polling user interaction data. The responsibility of collecting user interaction data was delegated to AWS Lambda functions, while polling was managed by a Django REST service. Both subsystems are integrated and managed through a custom Python library for SQS, as described in the section Custom Python Library (V-B).

F. API Gateway WebSocket APIs

A non-functional requirement, such as Graphical User Interface Performance, was supported by WebSockets, as they improved request performance in the graphical user interface. Instead of sending a separate HTTP request for each interaction, such as coupon views or redeemed coupons, WebSockets establish a single real-time channel connection to handle these interactions efficiently.

Another non-functional requirement that benefited was User Experience, as WebSockets facilitated asynchronous tasks. For example, they enabled real-time updates to graphical components, such as removing redeemed coupons and displaying new reports and coupons on both the user and hotelier interfaces.

VII. Continuous Integration, Delivery, and Deployment

According to Redhat [7], Continuous Integration (CI) refers to an automation process that builds, tests, and merges code changes. Continuous delivery (CD) refers to the automated release of a software version already built, tested and merged into a repository, and finally, Continuous Deployment (CI) refers to the automated deployment to production.

For the CI/CD process to succeed, three key factors were taken into account. The first was how to wrap and isolate the dependencies and execute the tests for each repository. The second was how to organize the release version for each repository. Finally, the third was how to identify when a specific release of the project is ready to be deployed to production servers.

To carry out these tasks successfully, Docker was chosen as the containerisation platform, and Docker Compose was used as the container orchestration system. Additionally, branches such as `staging/***` and `main` were defined for releasing and deploying each repository.

A. Docker and Docker Compose

Docker was used to install all dependencies and to separate the production and development environments using environment variables for each repository. A custom Docker image was created for each repository, containing its own code and dependencies in isolation. This setup allowed all repository Docker containers to be run via Docker Compose, enabling orchestration of the execution order and management of dependencies between them

B. Repositories and Branches

The project is divided into four repositories: a Django REST API service, an Angular Single Page Application, a Python library, and a deployment repository hosted on a Distributed Version Control System (VCS) such as GitHub. Each repository includes two key branches: `staging/***` and `main`.

The staging/*** branch is used to push new features of the Hotel Coupon App and to manage versioning for app releases, such as `staging/1.2.0`, `staging/1.2.1`, `staging/1.2.2`, ...etc.

On the other hand, the `main` branch is used to push new tested versions of the repository.

C. Pipelines

GitHub Actions execute the pipelines, which are divided into four stages: three are handled by the `staging/***` branch, and the last is managed by the `main` branch.

The `staging/***` branch builds, runs tests, and releases a specific docker image to the Docker registry. The "fig. 10" illustrates the CI/CD pipeline stages for the `staging/***` branch

On the other hand, the `Main` branch is used to deploy the new repository version. It connects to an EC2 instance, pulls the new Docker image of the repository, and orchestrates it using Docker Compose. Figure 11 illustrates the deployment stage of the main branch.

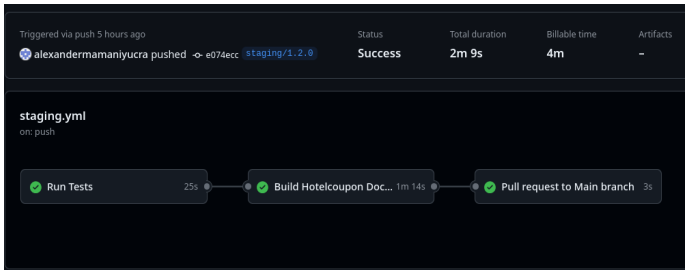

Fig. 10. Continuous Integration and Delivery of Django REST-API Service

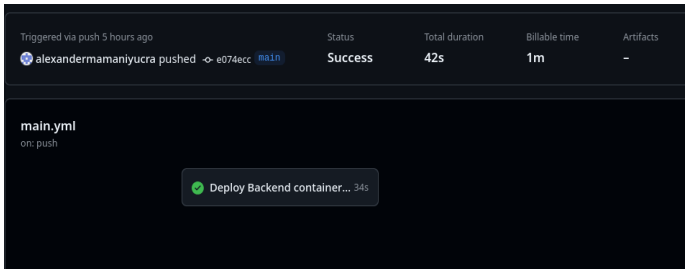

Fig. 11. Continuous Deployment of Django REST-API Service to AWS EC2

The Hotel Coupon App is available in a public IP Here and the credentials are listed in the "Table. I".

TABLE I
Hotel Coupon App Credentials Table

Hotel Coupon Credentials		
Role	***Email***	***Password***
USER	alex@user.com	user
USER	beatriz@user.com	user
USER	luis@user.com	user
HOTELIER	dublinhotel@hotelier.com	hotelier
HOTELIER	barcelonahotel@hotelier.com	hotelier
HOTELIER	greenhotel@hotelier.com	hotelier

VIII. Conclusions

This project helped me understand multiple aspects of cloud-based software development. For instance, I explored the impact of using technologies such as AWS SQS and SNS compared to Apache Kafka. Both implement messaging models like queuing and publish-subscribe, but the main difference lies in the time and resources required. With Kafka, you may spend more time and effort implementing and managing the infrastructure from scratch, whereas, with AWS SQS and SNS, you only pay for the resources you use. This pay-as-you-go model makes AWS services more scalable for unpredictable projects, as you do not need to manage the entire infrastructure yourself.

This project also made me reflect on the importance of being a full-stack developer. For instance, to implement the entire project, it is necessary to have sufficient knowledge

in various software fields, such as Requirements Engineering, DevOps, and Frontend and Backend Development. Focusing on only one field limits our ability to get a comprehensive understanding of a project's full scope. It prevents us from developing the capacity to abstract problems, as a software architect would do.

One aspect I would change in the project is implementing DevOps from the beginning. This approach would have significantly improved the processes of testing, building, and deploying features, especially considering the project's multiple components. Additionally, I would opt for Elastic Container Service (ECS) instead of EC2 instances, as ECS allows for scaling container clusters on demand.

REFERENCES

[1] AWS, "Object Storage Classes – Amazon S3," Amazon Web Services, Inc., 2023 [Online]. Available: https://aws.amazon.com/s3/storage-classes/

[2] AWS, "Amazon RDS Backup & Restore — Cloud Relational Database — Amazon Web Services," Amazon Web Services, Inc. [Online]. Available: https://aws.amazon.com/rds/features/backup/

[3] AWS, "Amazon DynamoDB Features — NoSQL Key-Value Database — Amazon Web Services," Amazon Web Services, Inc. [Online]. Available: https://aws.amazon.com/dynamodb/features/

[4] AWS, "Welcome - Amazon Simple Queue Service," Amazon.com, 2024. [Online]. Available: https://docs.aws.amazon.com/ AWSSimpleQueue-Service/latest/APIReference/Welcome.html

[5] AWS, "What is Amazon SNS? - Amazon Simple Notification Service," docs.aws.amazon.com. [Online]. Available: https://docs.aws.amazon.com/sns/latest/dg/welcome.html

[6] AWS, "What Is AWS Lambda? - AWS Lambda," Amazon.com, 2024. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html

[7] Redhat, "What is CI/CD?," Redhat.com, Dec. 12, 2023. [Online]. Available: https://www.redhat.com/en/topics/devops/what-is-ci-cd