Université Abdelmalek Essaadi

Ecole Nationale des Sciences Appliquées Al-Hoceima

Real-Time Spam Detection

Digital Transformation

&

Artificielle Intelligence 2

Supervised by : Pr. ROUTAIB Hayat

Realized by : BADRI Insaf

ALLALI Mohamed Amin

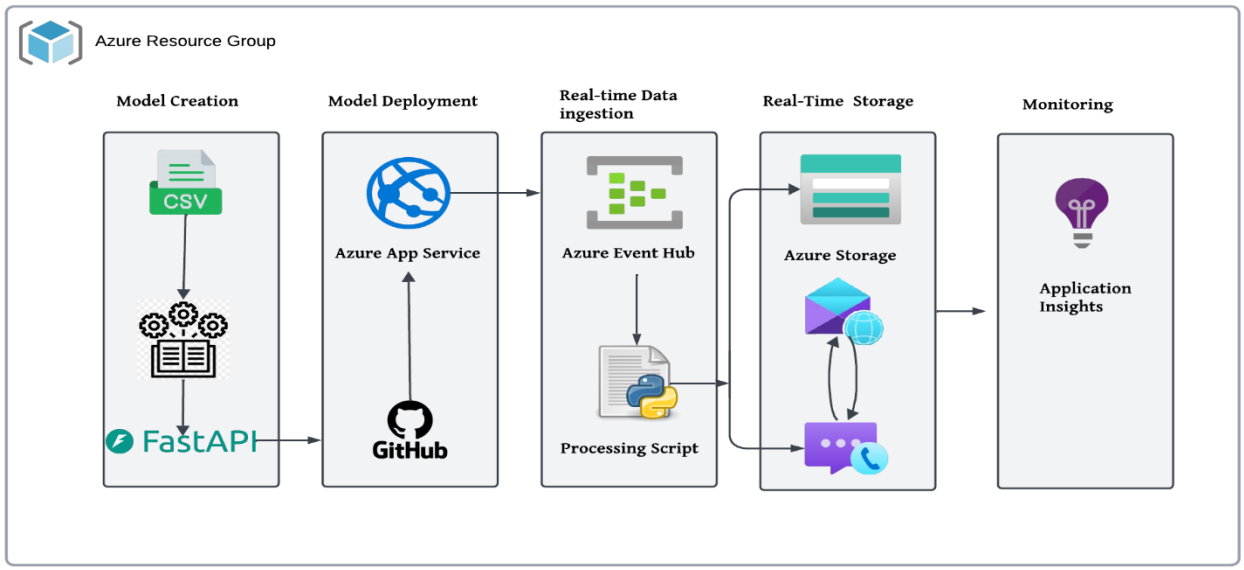
Table of content

I- Problematic:

Spam is one of the major challenges to online communication, with security risks ranging from phishing to malware. Real-time spam detection requires scalable, accurate, and low-latency solutions that can handle huge streams of data. Azure offers powerful tools, such as Cognitive Services, Machine Learning, and Event Hubs, to enable such systems. However, challenges persist: How can we design a solution that balances scalability, cost-efficiency, and integration while ensuring adaptability to evolving spam patterns?

The challenge is to use the Azure ecosystem to develop a robust, real-time spam detection system that can effectively address dynamic threats, minimize false positives and negatives, and be cost-effective for organizations.

II- Project Architecture



III- Model Creation:

1. Data Collection and Processing

1.1. Collection:

The dataset was collected from two different sources. First, the SMS Spam Collection from the UC Irvine Machine Learning Repository, which contains 5,574 instances, including 4,827 messages classified as non-spam and 747 messages classified as spam. Second, a public dataset found on Kaggle, which contains 8,000 instances. From this dataset, 3,900 spam messages were selected to

achieve a balanced class distribution (spam and non-spam).These two datasets were combined to form the final dataset of 9,699 messages labeled as spam or non-spam.

1.2. Processing

The dataset was prepared for creating the model first by removing null values and duplicate entries so that the data is consistent and reliable. The label column was encoded by assigning 0 for non-spam and 1 for spam to enable binary classification. Text preprocessing was done by defining a function to clean the data, which included removing URLs, HTML tags, and non-alphanumeric characters excluding numbers, changing text to lowercase. The text was tokenized, stop words were removed, and lemmatization was performed to normalize the word and reduce them to base format. These pre-processing steps cleaned up the dataset and made it well-structured, leading to optimize it for training an accurate and effective spam detection model.

2. Features selection

For feature selection, both unigrams and bigrams were analyzed in spam and non-spam messages. It was observed that unigrams (single words) appeared more frequently than bigrams (two-word combinations) across the dataset. Based on this analysis, the focus was placed on using unigrams as features. To further optimize the feature set, the Term Frequency-Inverse Document Frequency (TF-IDF) method was applied, with a maximum feature limit of 7000. This approach ensured that only the most relevant and informative words were selected, enhancing the model's ability to effectively distinguish between spam and non-spam messages.

3. Model training and selection

For model training and selection, the dataset was split into training and testing sets, with 80% of the data used for training and 20% reserved for testing. Four different models were evaluated Naive Bayes, Decision Tree, Support Vector Machine (SVM), and Logistic Regression. To optimize hyperparameters, Grid Search with 5-fold Cross-Validation (CV) was used for each model. After training, the models' performance was evaluated on the test set by measuring accuracy. Additionally, confusion matrices were generated to further assess model performance. The model with the highest accuracy was selected as the final model for the task

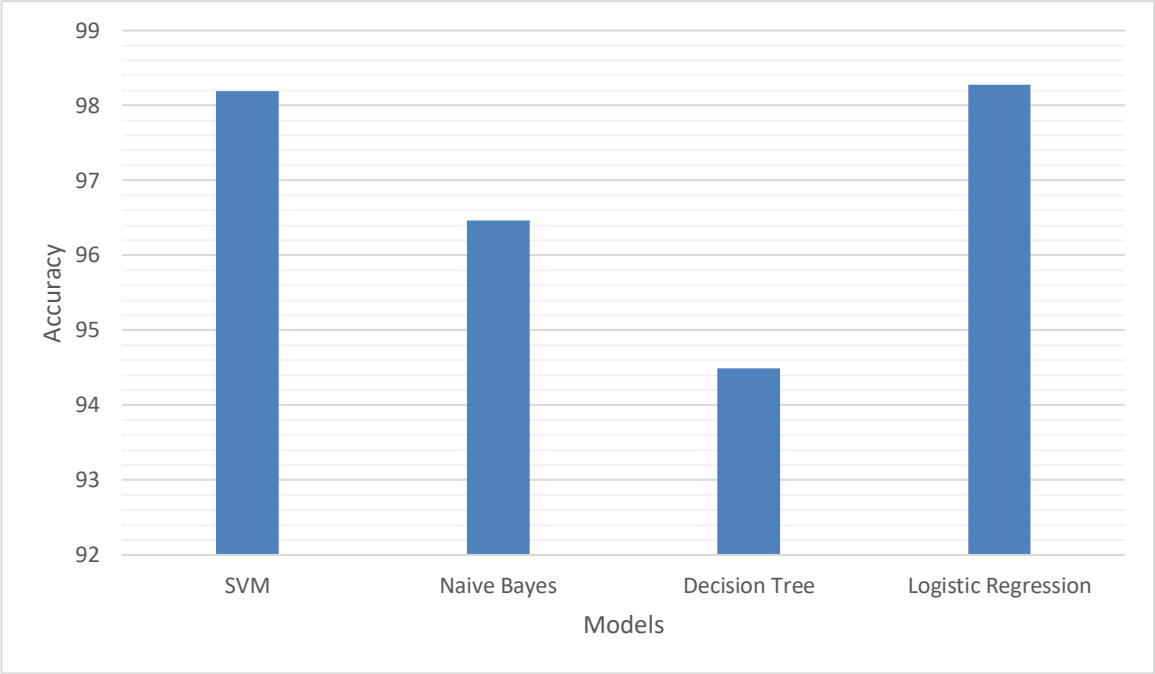


Figure 1: Accuracy of Models on test set

IV- Model Deployment:

1. API creation

For model deployment, a **Fast API** application was developed to enable real-time spam detection. The API accepts a text input from users via a POST request and processes it through the trained model. The prediction, indicating whether the text is spam or not, is returned in JSON format, making it easily consumable by other systems or applications.

2. Deployment

We used Microsoft Azure by creating a resource group and setting up an instance of the Azure Web App service. This provided a scalable and secure environment to host the FastAPI application. To streamline updates, we integrated the deployment pipeline with GitHub for continuous deployment. This ensures that whenever a commit is made to the repository, the FastAPI application is automatically deployed. Additionally, we specified the running command required to start the application, ensuring it runs seamlessly each time it is deployed. The API is available on the route /predict

✓ **Steps to deploy the Fast API App:**

➢ **Create a resource group:**

we should specify the subscription to use, give a name to the resource group and finally chose the resource group region then we click on Review + Create

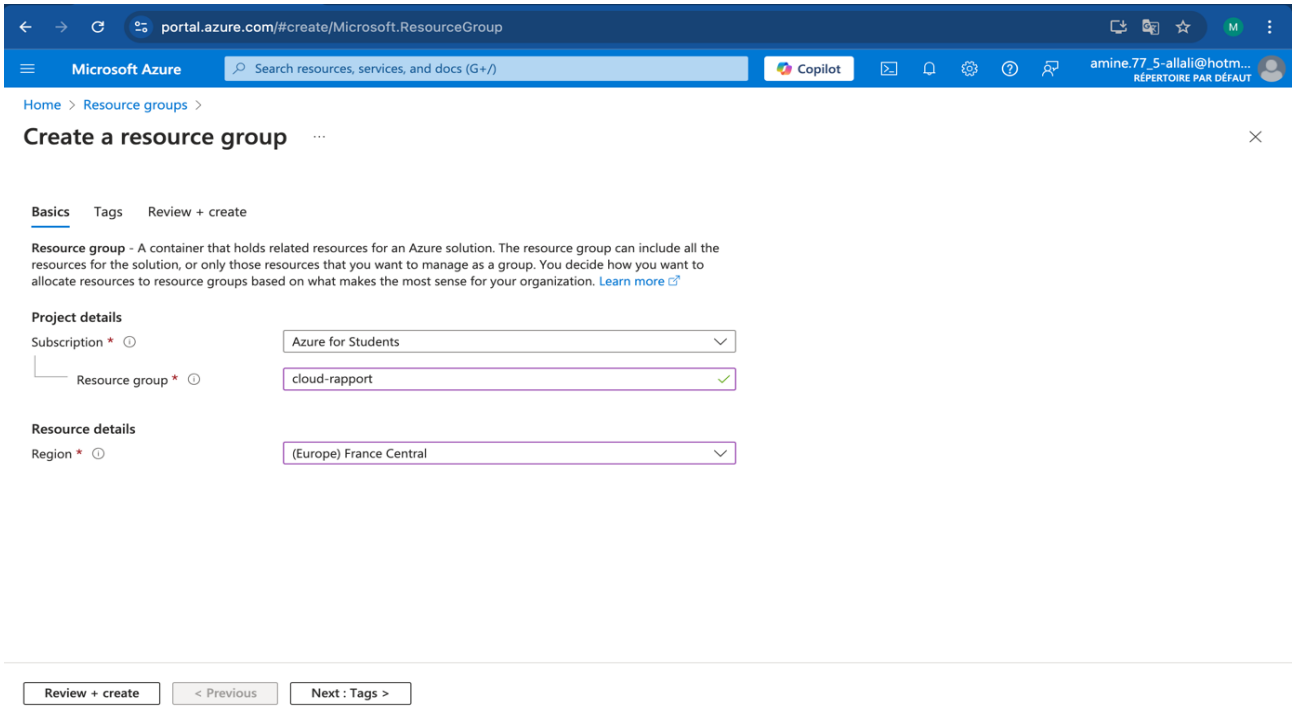


Figure 2: Creating a resource group

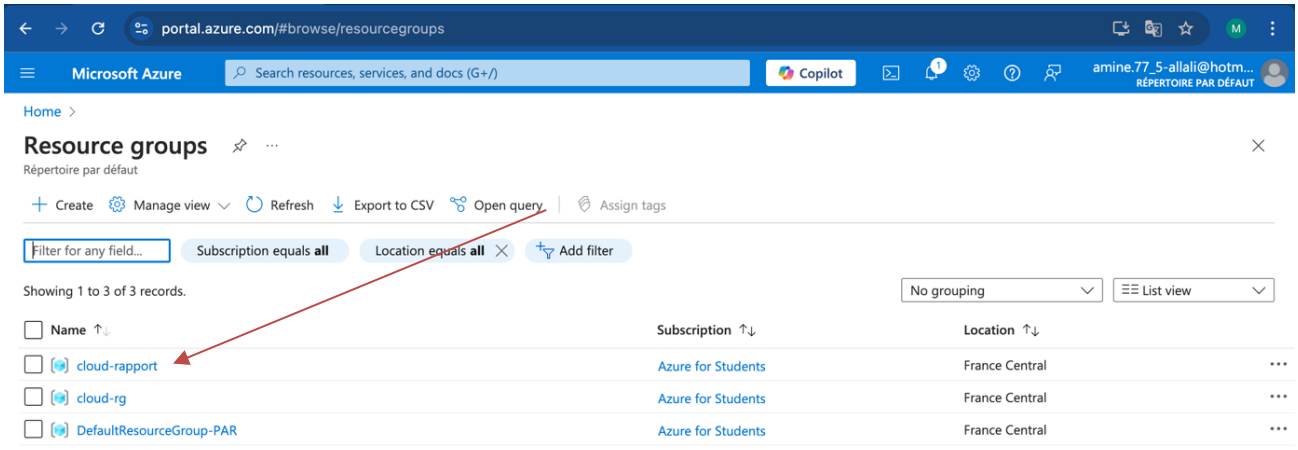


Figure 3: Checking if the resource group has been created

✓ Creating a WebApp

After creating the resource group, we will click on it, then click on create, after that we will look for web App then click create

After that we will specify the name of the app, publish method (code or container), the runtime stack in our case Python 3.12, the region and the linux Plan for the hosting and finally the pricing plan, in our case we will chose F1 because it's free. After all that we will click on Review + Create and navigate to the resource once created

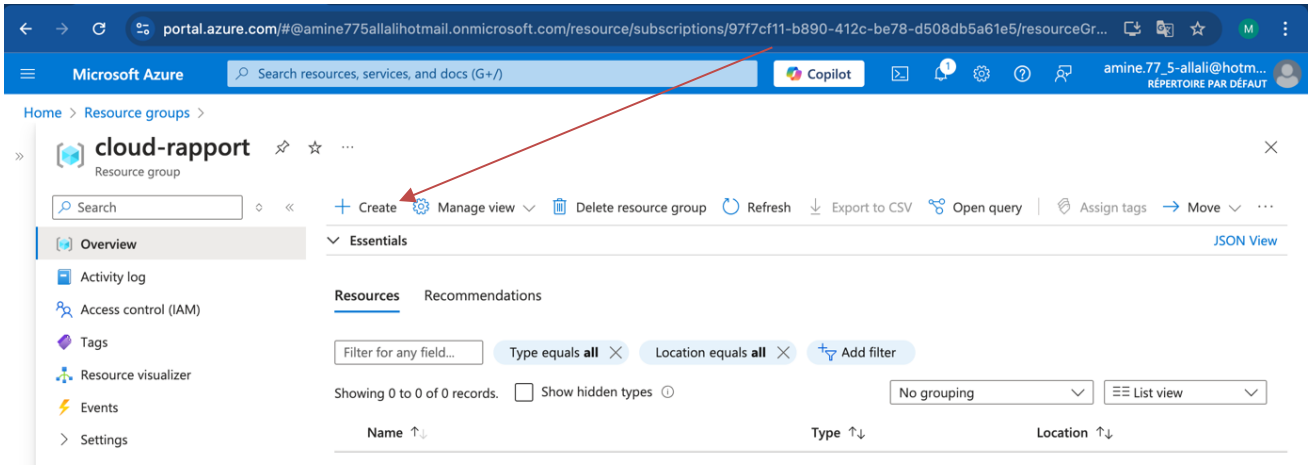


Figure 4: creating a new resource

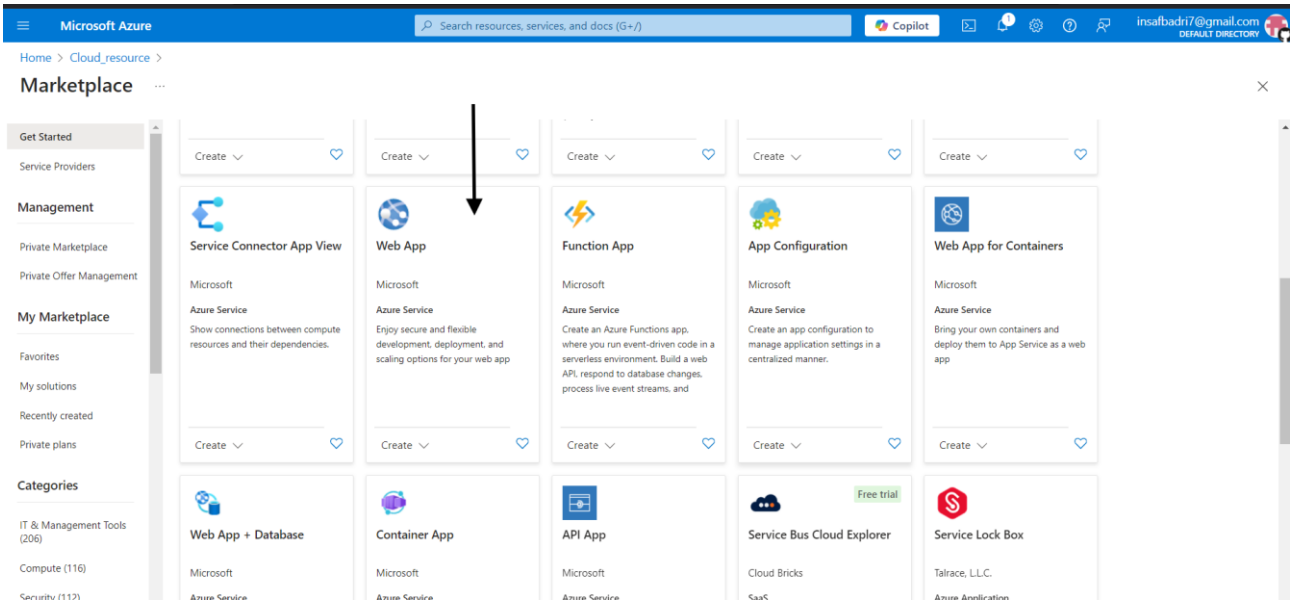


Figure 5: creating a Web App

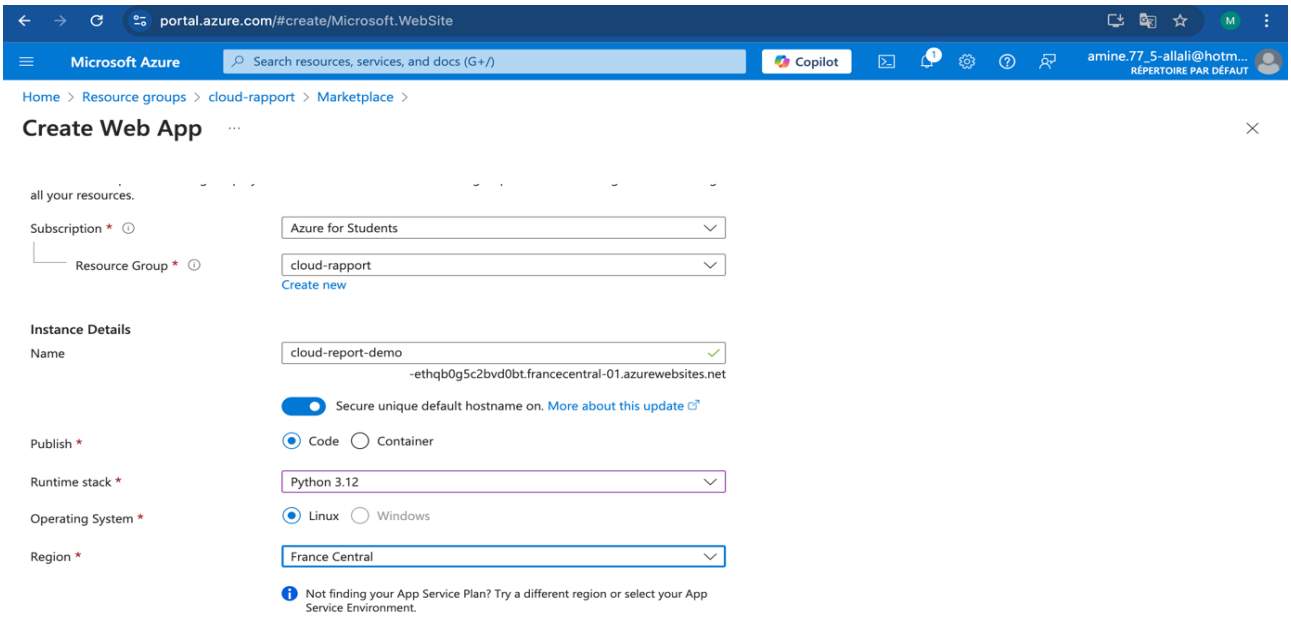


Figure 6: configuring the Web App

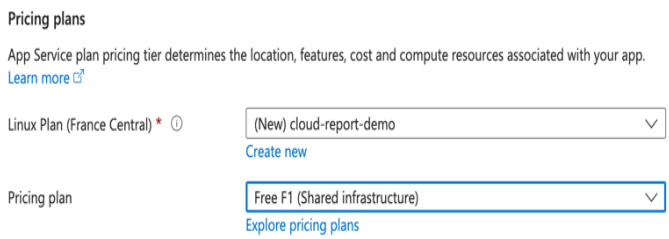


Figure 7: configuring the Web App

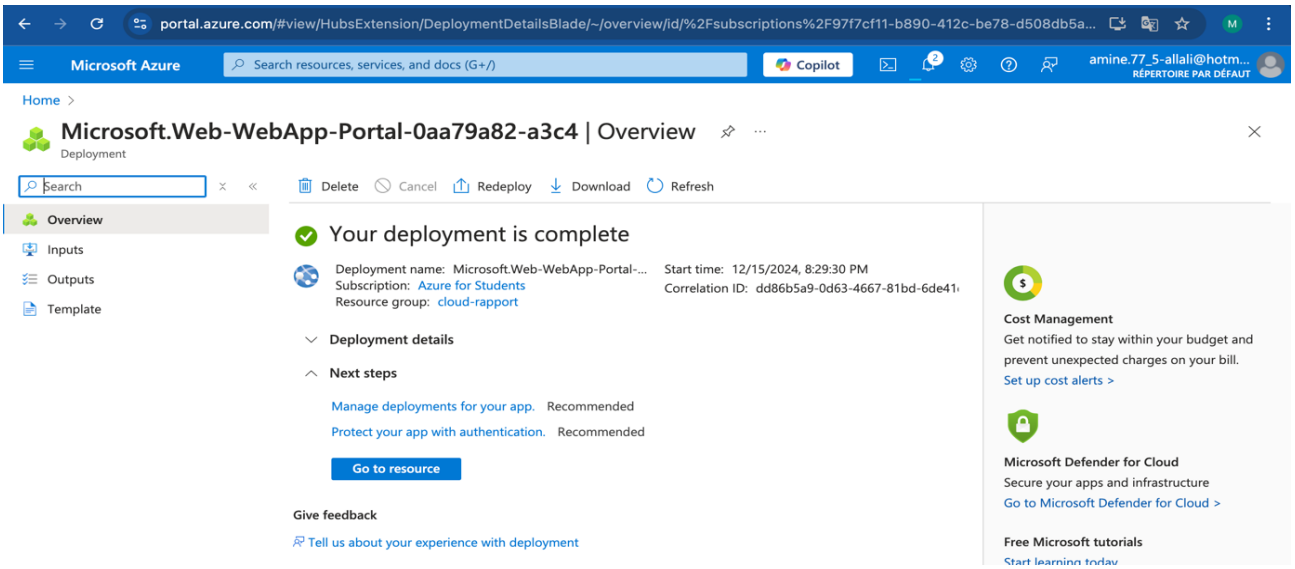


Figure 8: Creation of the Web App succeeded

Once the resource is created (the Web App) we will go to Deployment Center where we are going to choose the source (GitHub) then the repository on our GitHub (After linking our GitHub account to Azure) once we done that, we will save the modifications.

Then we will navigate to configuration to specify the startup command that will run our app

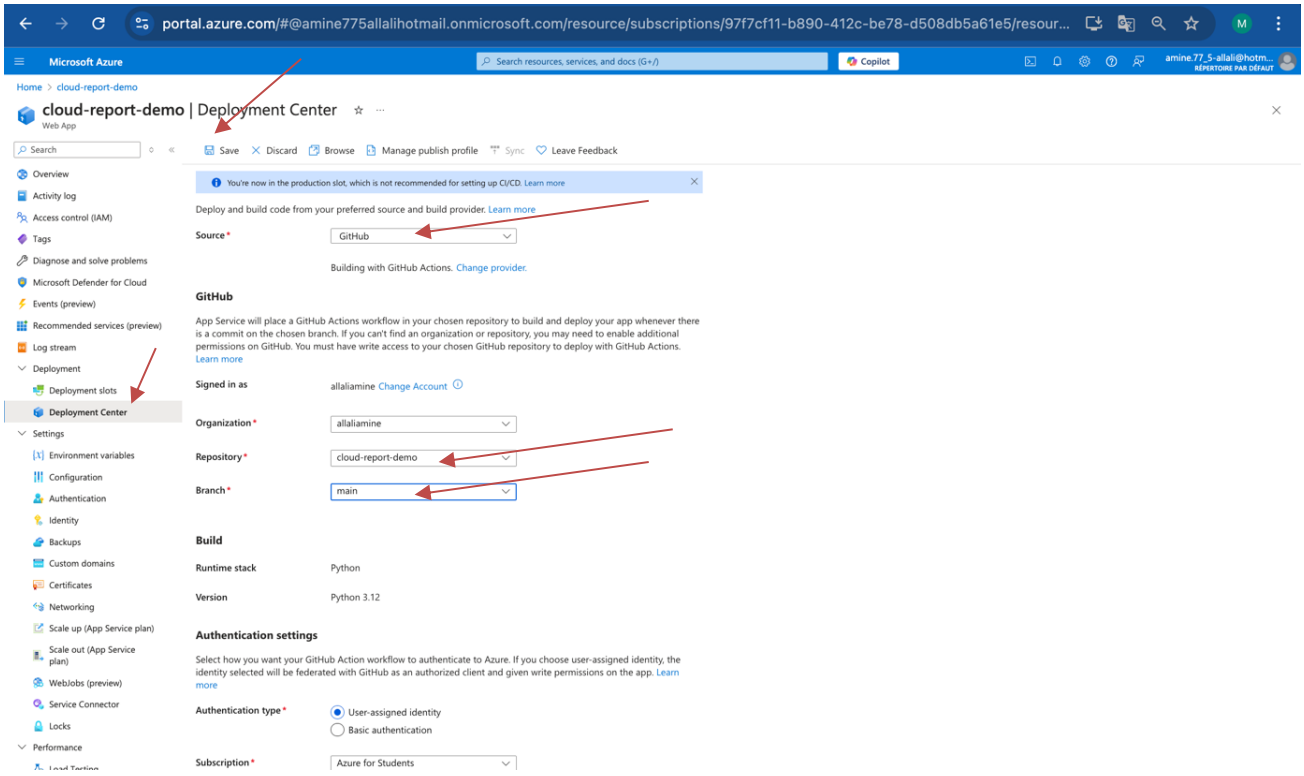


Figure 9: Deployment of the Web App

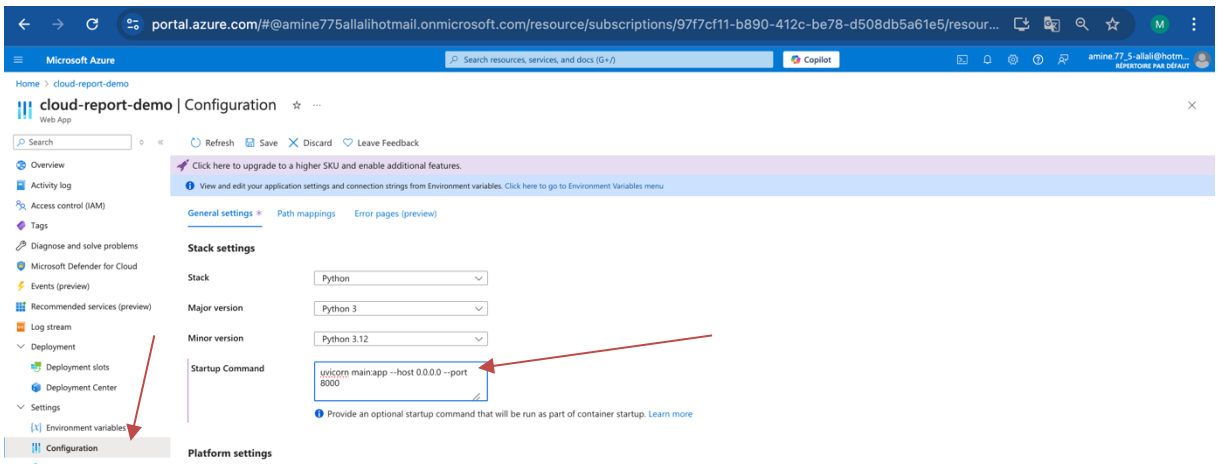


Figure 10: Starting Command of the Web App

Now we can see that the deployment was Succeeded, after that we will browse to our API by clicking Browse

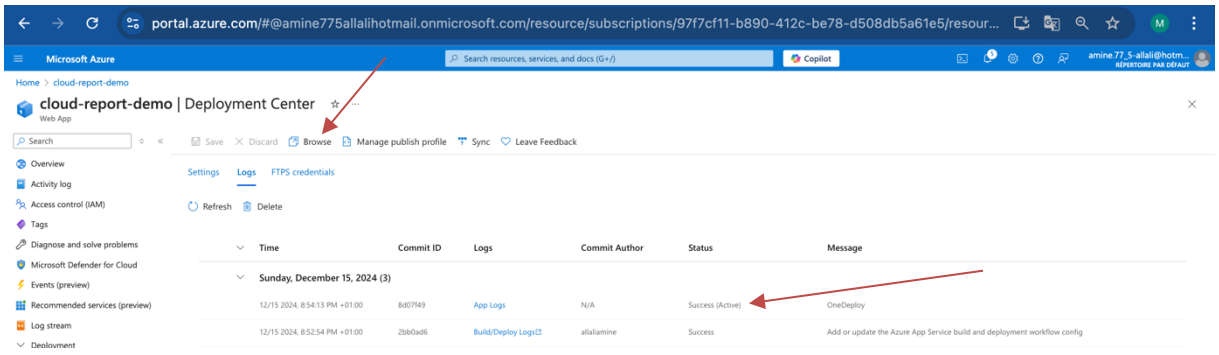


Figure 11: Checking if deployment is succeeded

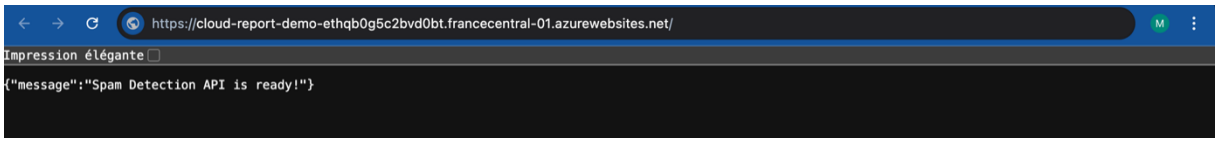


Figure 12: API

V- Real-time Data Ingestion:

Now moving to the next step where we are going to ingest data in real time to get prediction in real-time using our deployed API, in this step we will be using Azure Event Hub. But before configuring azure Event Hub we need a data source from where we are going to fetch data then ingest it in real-time.

1. Creating an API for data

We created a custom API integrated into the same *main.py* file as our previously deployed prediction API. This new API, accessible through the route */send-email*, dynamically fetches and returns data from a CSV dataset. Each time the API is called, it randomly selects a row from the CSV file and returns three fields: the receiver of the message (Email), the content of the message (Message), and the sender of the message (Sender). By ensuring that a random and unique row is selected for each request.

2. Configuring Event Hub and The Sending Script

Before creating the sender script, we will configure an Azure Event Hub following those steps

- ✓ Creating an Event Hub namespace

We will navigate to our resource group click on new, search for event hub and click create

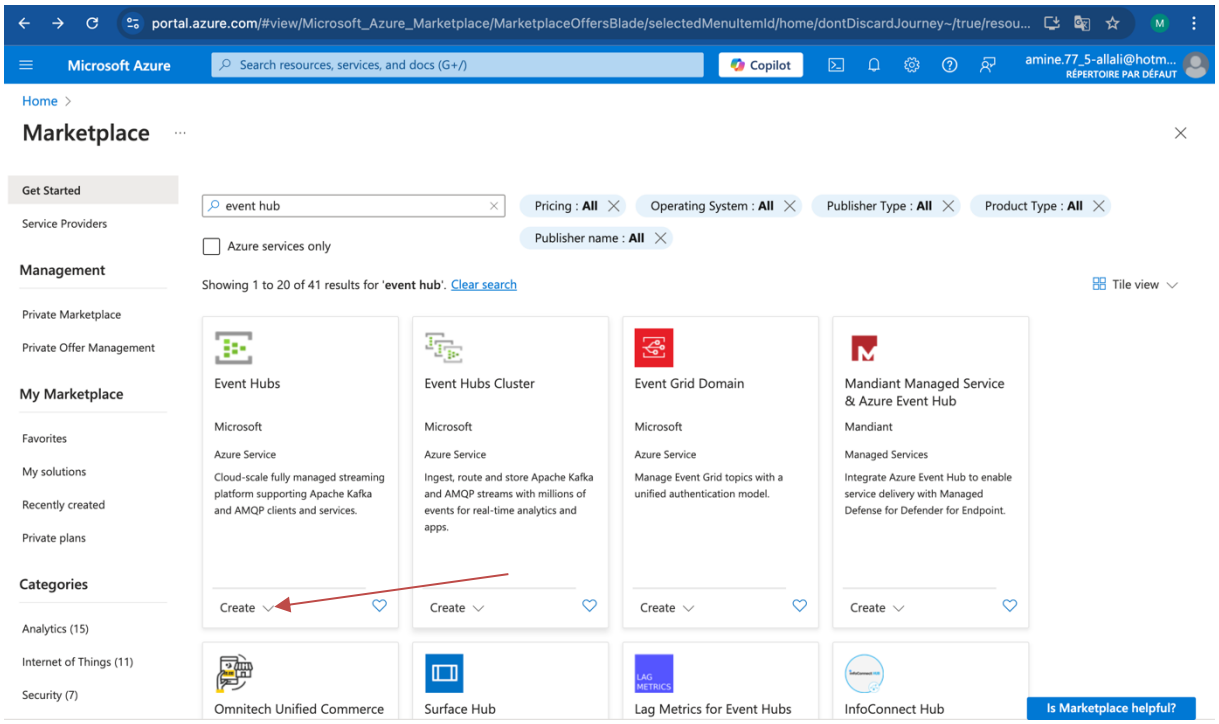
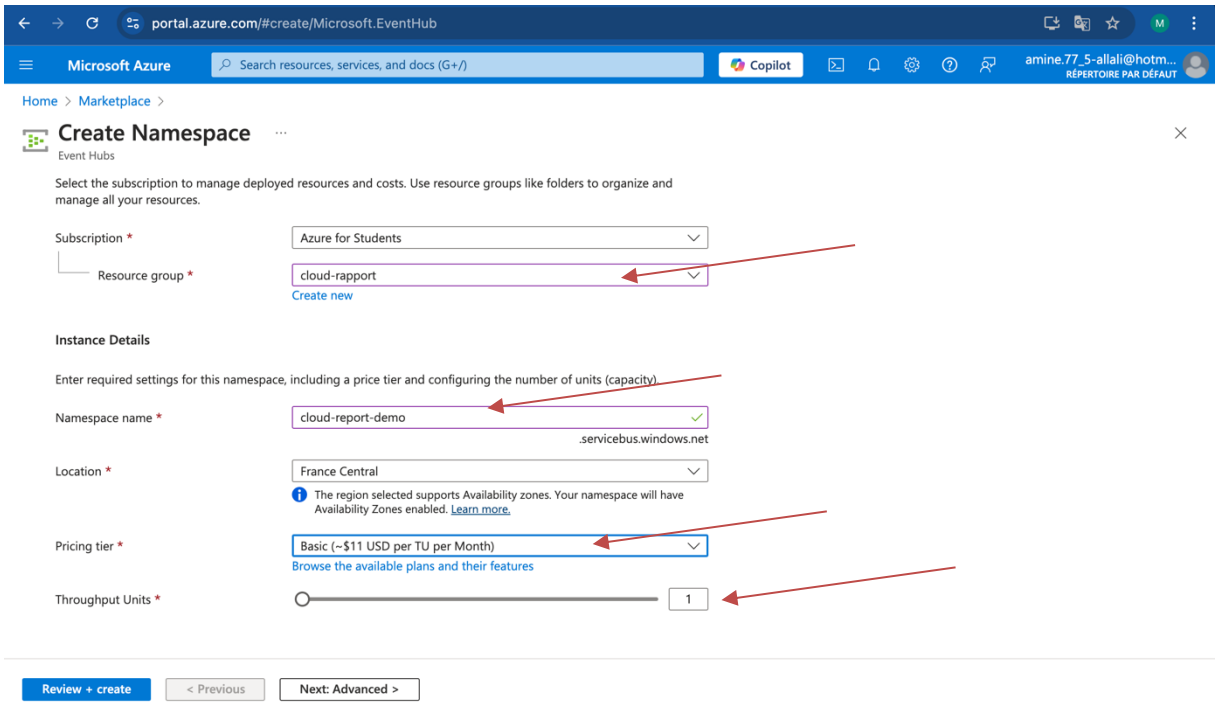


Figure 13: Creating an Event Hub



We should specify the resource group, the name of the event namespace the location, then we should choose a pricing tier and finally we should choose the number of Throughput Units. After all that we click on Review + Create

Throughput Units (TUs) are a measure of the capacity for data ingress (incoming data) and egress (outgoing data). Here's a breakdown of what they mean:

Ingress: Each Throughput Unit allows for up to 1 MB per second or 1,000 events per second, whichever comes first.

Egress: Each Throughput Unit allows for up to 2 MB per second or 4,096 events per second, whichever comes first.

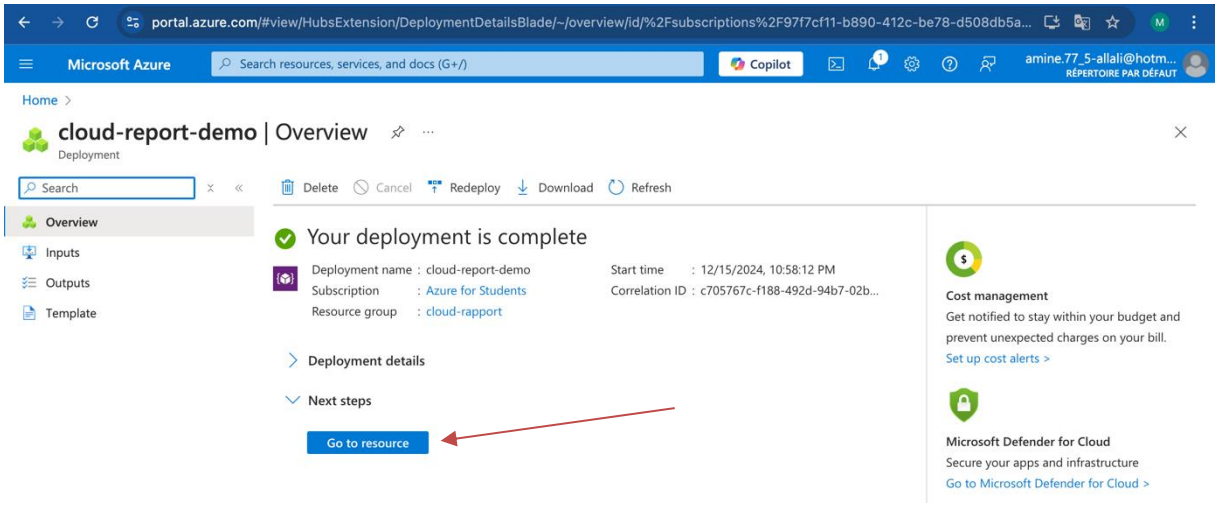


Figure 14: Creation of the Event Hub succeeded

✓ Create an Event Hub:

After creating the Event Hub namespace, we should create an event hub and in order to do that we should click Event Hubs in entities and click on Event Hub.

We should enter the name of the Event Hub, Partition Count, Clean up policy and retention time

Partitions are used to parallelize the processing of events.

Retention time is the time before the clean-up policy is applied because Event Hub isn't a storage tool it doesn't store data coming in it for a long time

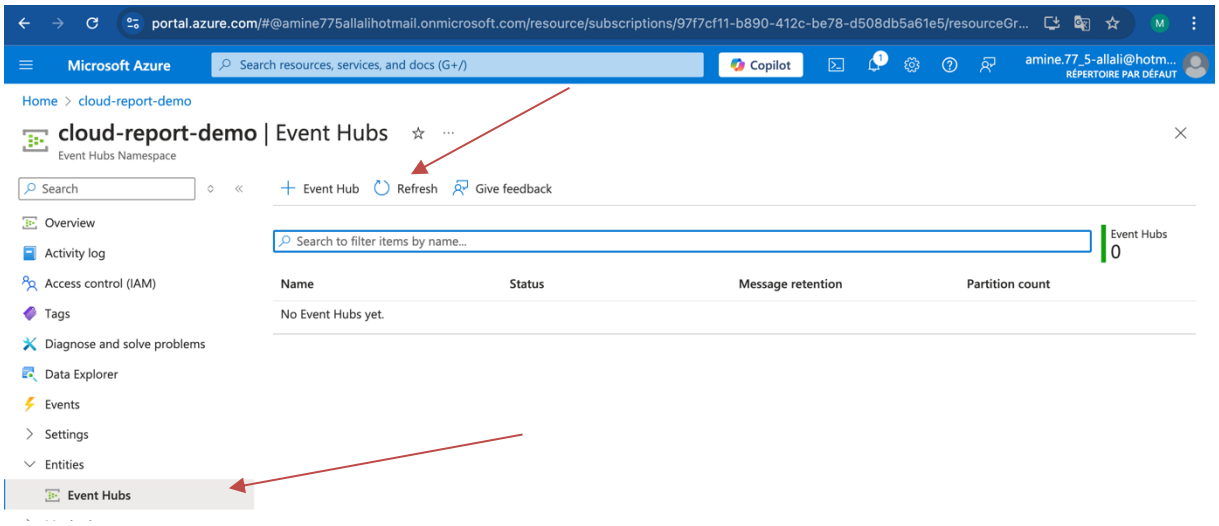


Figure 15: Creation of an Event Hub

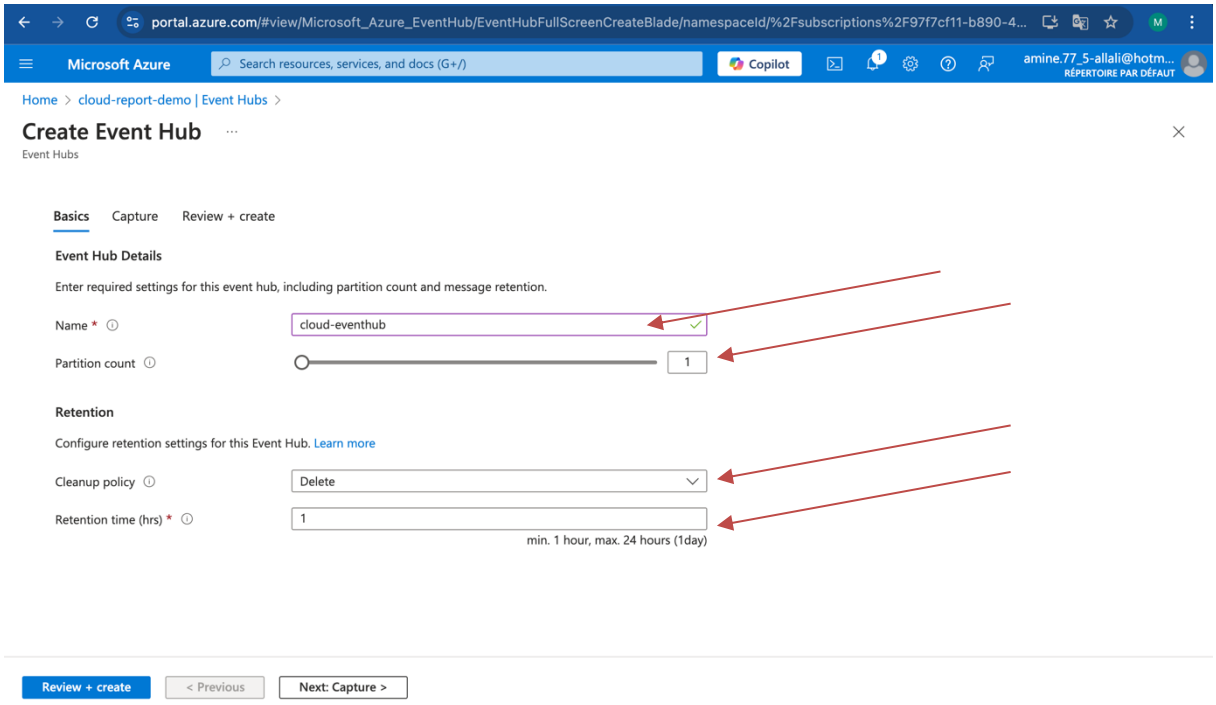


Figure 16: Creation of an Event Hub

After we specify all of that we click on Review + Create and it creates our event hub as shown bellow

Figure 17: Creation of the Event Hub succeeded

✓ Create a shared access policy:

After that to use our event hub we should create a shared access policies and in order to do that we should click on our event hub go to settings and click on shared access policies, click on add, enter a name and choose manage then hit create.

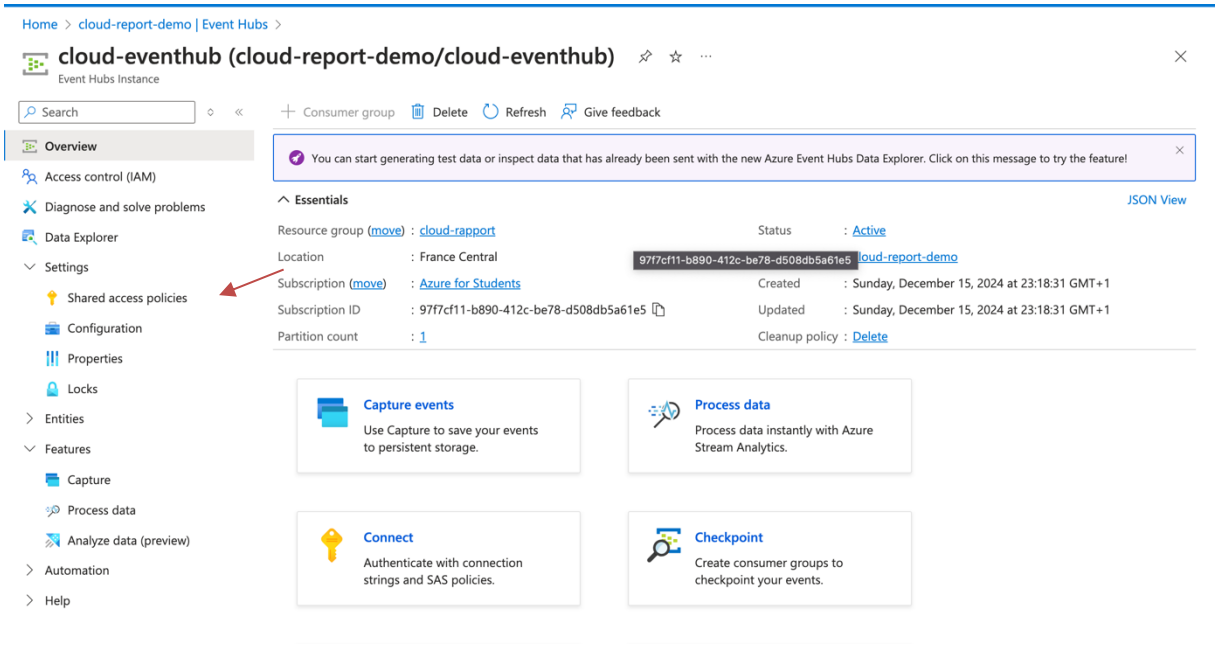


Figure 18: Event Hub Overview

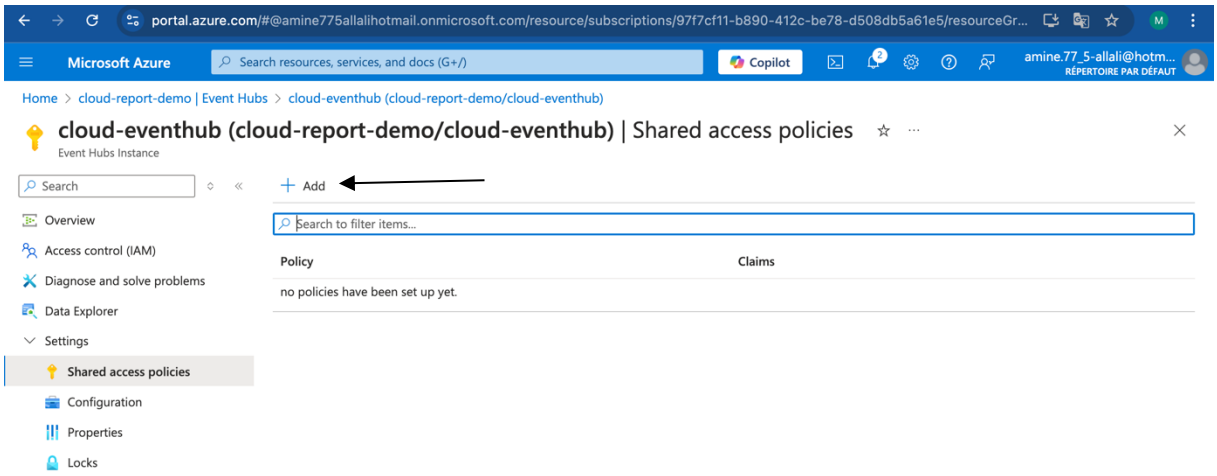


Figure 19: Event Hub Shared access policies

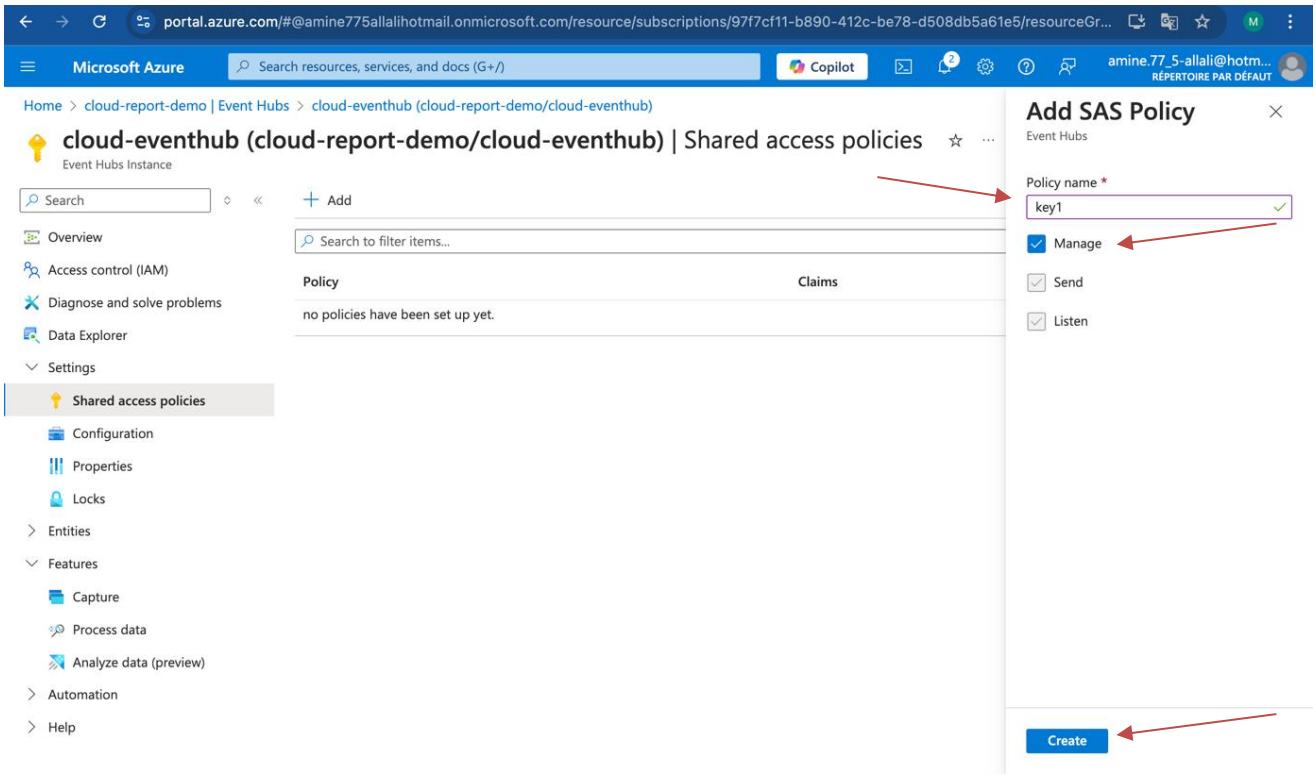


Figure 20: Creating an Event Hub Shared access policies

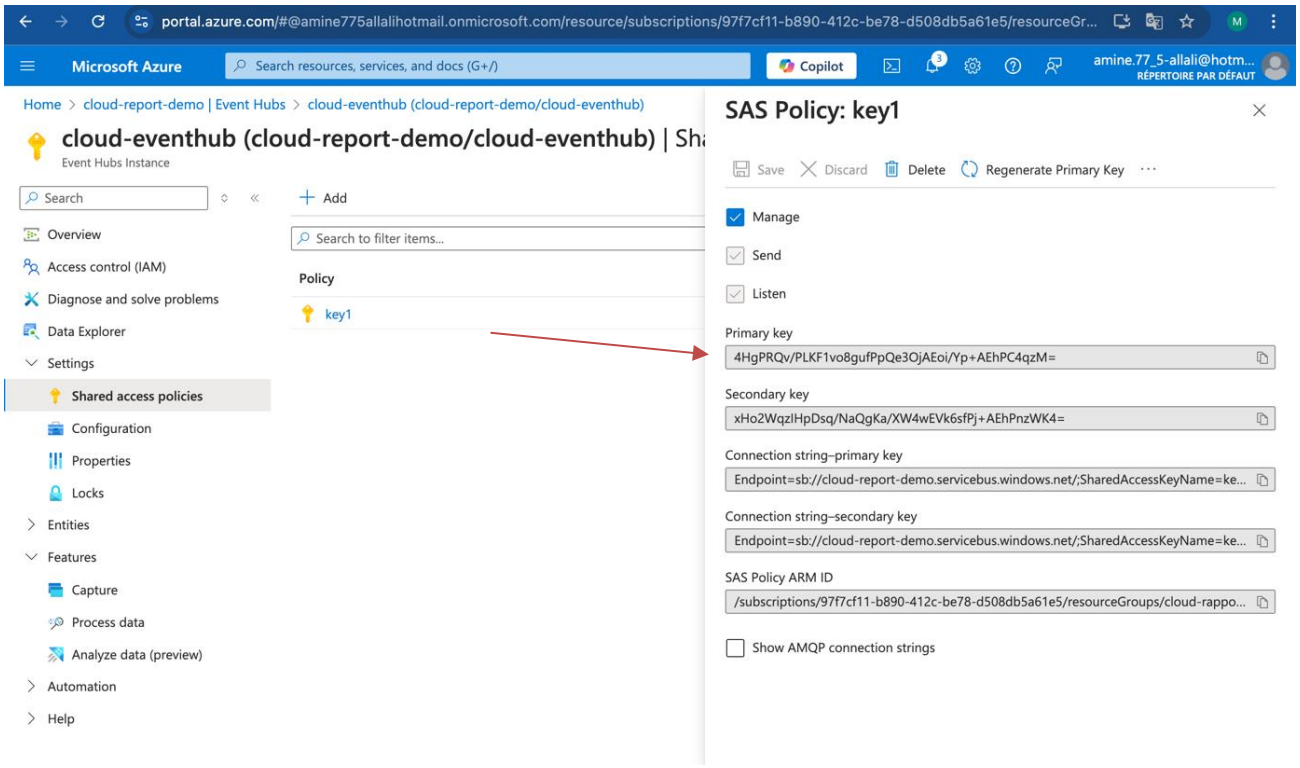
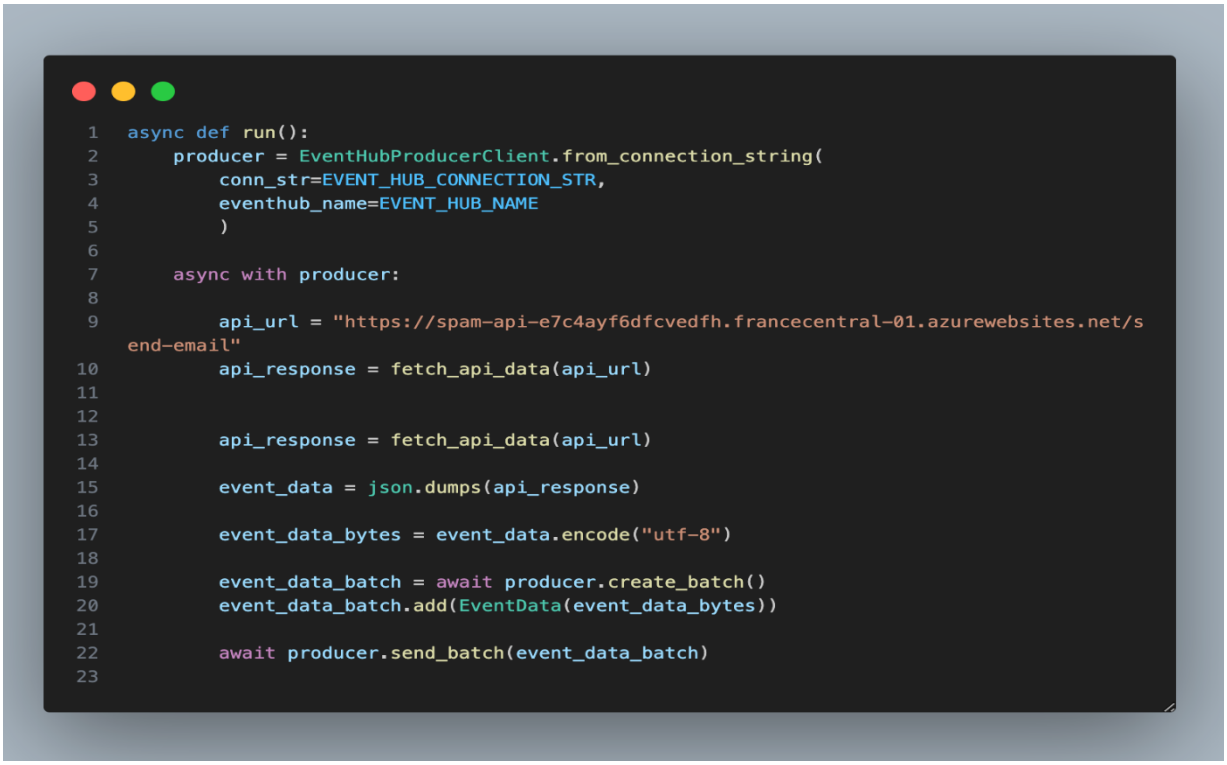


Figure 21: View the Event Hub Shared access policies created

Now we can use our Event Hub to ingest data in real time

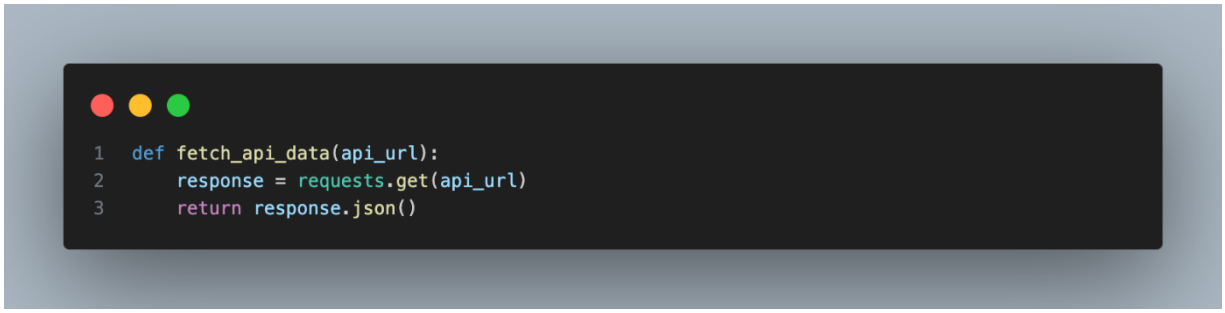
To consume and ingest data into our Event Hub, we will use the Python script below, leveraging the Azure Event Hub library. The script defines three functions

- ✓ **fetch_api_data()**: This function fetches data from our customized API.
- ✓ **run()**: This function sends the fetched data to the Event Hub.
- ✓ **main():** This function runs a loop that sends data by calling the run() function.



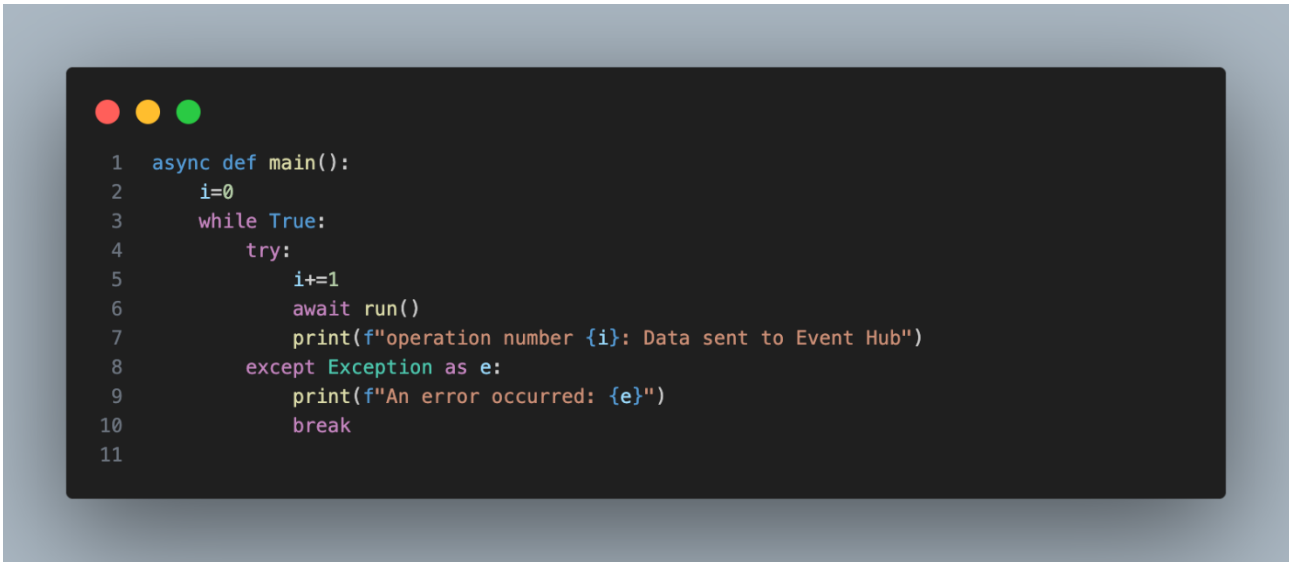
```python
async def run():
    producer = EventHubProducerClient.from_connection_string(
        conn_str=EVENT_HUB_CONNECTION_STR,
        eventhub_name=EVENT_HUB_NAME
        )

    async with producer:

        api_url = "https://spam-api-e7c4ayf6dfcvedfh.francecentral-01.azurewebsites.net/send-email"
        api_response = fetch_api_data(api_url)


        api_response = fetch_api_data(api_url)

        event_data = json.dumps(api_response)

        event_data_bytes = event_data.encode("utf-8")

        event_data_batch = await producer.create_batch()
        event_data_batch.add(EventData(event_data_bytes))

        await producer.send_batch(event_data_batch)

```

Figure 22: run function that send data



```python
def fetch_api_data(api_url):
    response = requests.get(api_url)
    return response.json()
```

Figure 23: run function that send data



```python
async def main():
    i=0
    while True:
        try:
            i+=1
            await run()
            print(f"operation number {i}: Data sent to Event Hub")
        except Exception as e:
            print(f"An error occurred: {e}")
            break

```

Figure 24: main function

To run the script, we type the command python3 send-to-event-hub.py

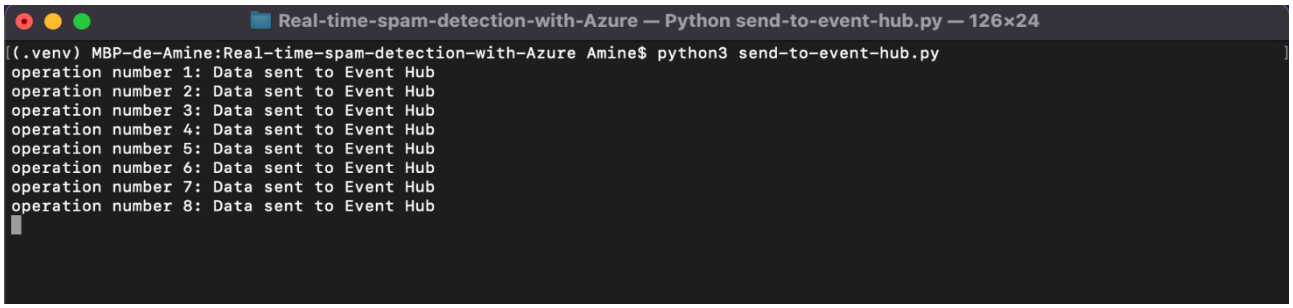


Figure 24: sending data to Event Hub

Now let’s check if the data has arrived at Event Hub, in order to do that we click on our event hub name then go to data explorer and hit view events:

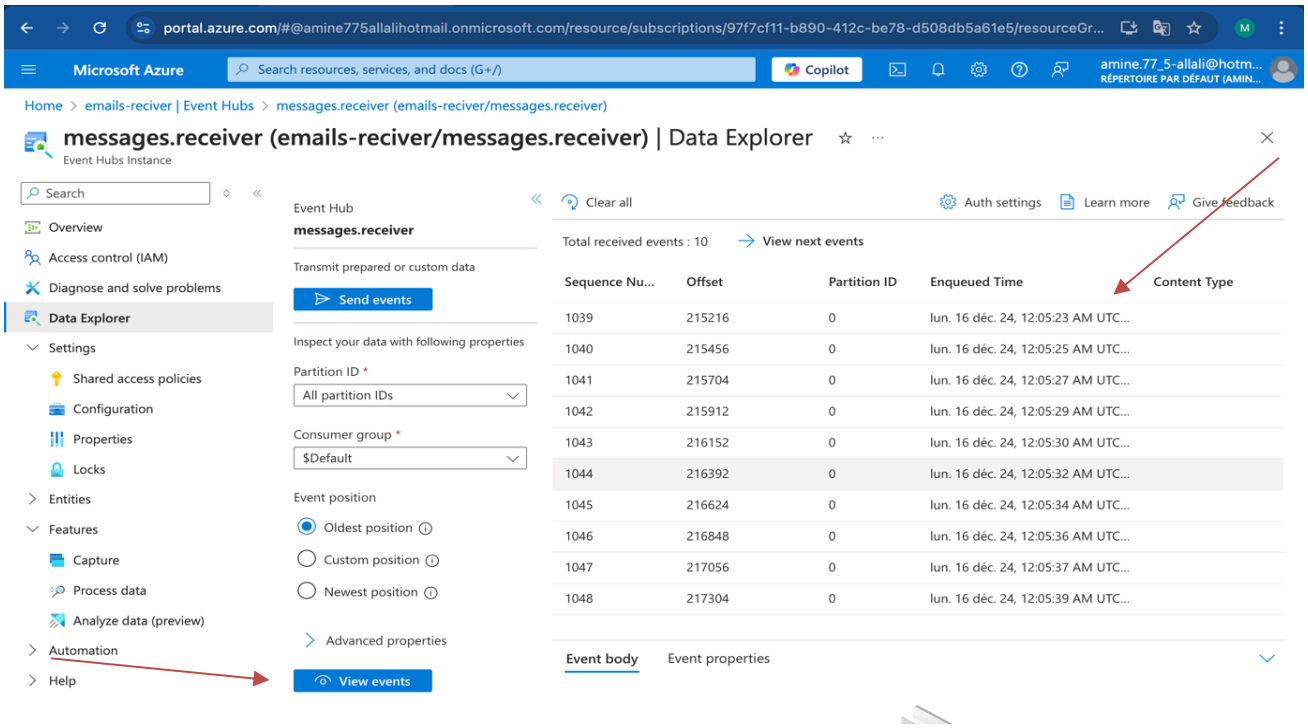


Figure 25: Received data in Event Hub

VI- Classify Streaming Data in Real-Time:

In this step, we focus on integrating the model with real-time data streams to classify incoming data in real time then store it in a Blob storage **as Json files**.

1. Creating a Blob storage to save data

First, we are going to create a blob storage and a container to stock our data and it's predictions in order to do that we will follow those steps:

- ✓ Create a Storage account in our resource group

We will navigate to our resource group click on Create and look for Storage account then click on create, fill up the necessary fields and click on Review + Create

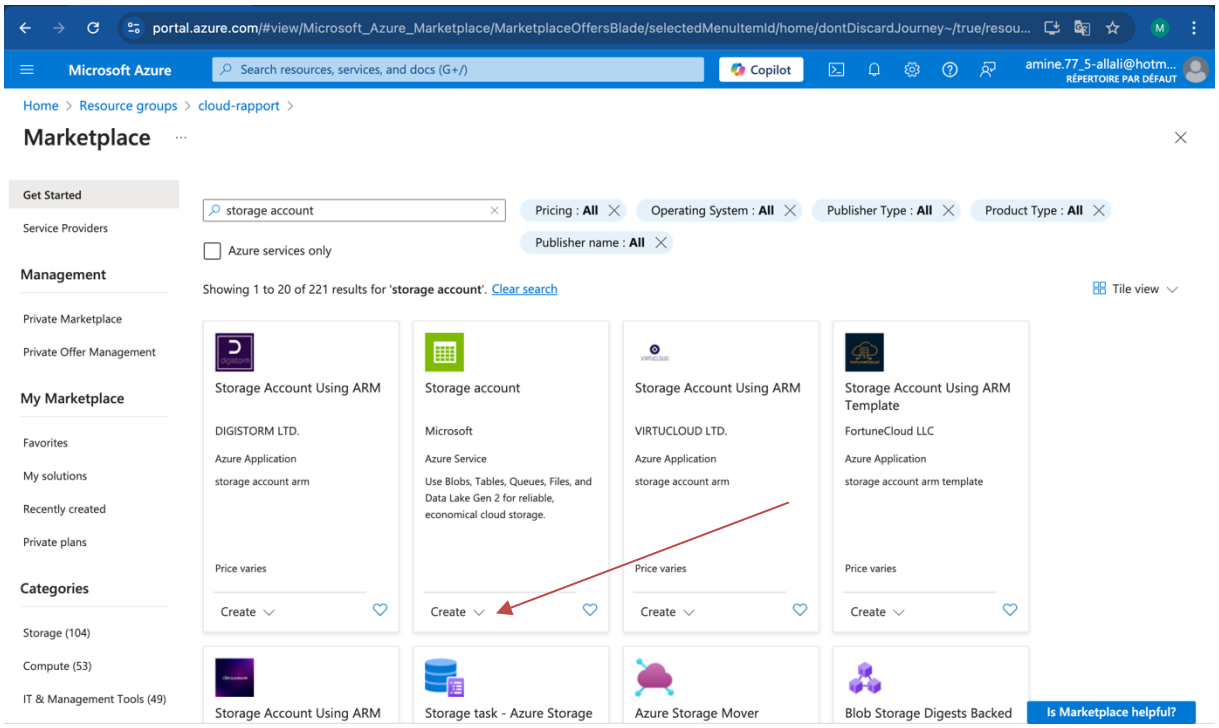


Figure 26: Creating a storage account

In the Primary service we will choose Azure Blob Storage or Azure Data Lake Storage Gen2, standard performance and redundancy we will choose LRS, after that we will create a container to store our json files.

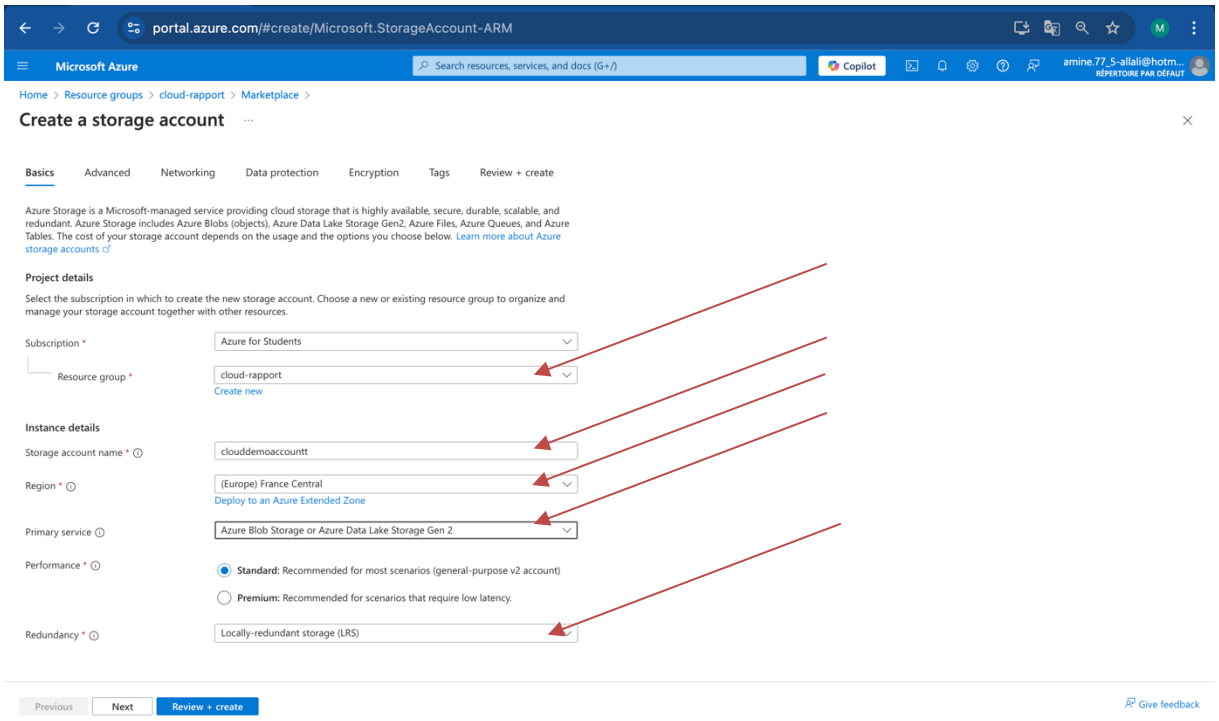


Figure 27: Creating a storage account (Blob storage)

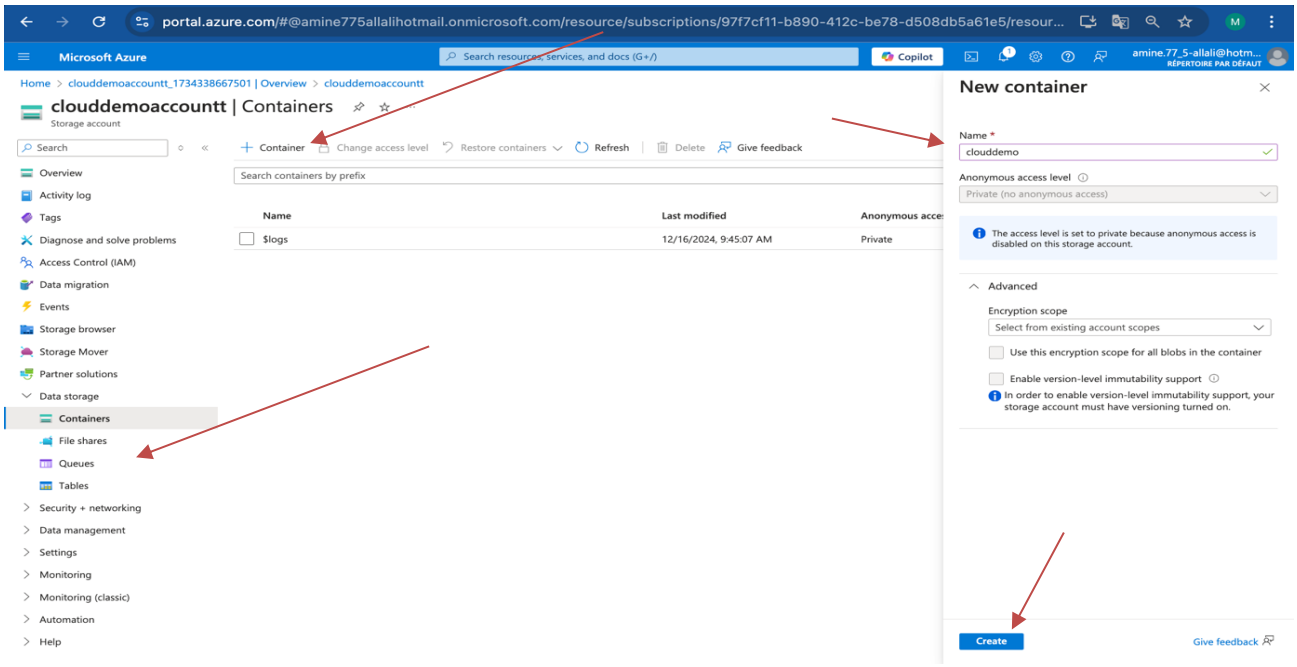


Figure 28: Creating a container to store our Json files

2. Creating the Script

Once our container is created, we can define our python script to receive data from event hub predict it using our API and store it into Blob storage. The script contains:

- ✓ Save_to_blob_storage(): function that save the data to blob storage
- ✓ On_event(): function that process the data received(call the API, store it, and send alerts using send_email)
- ✓ Main(): initializes the Event Hub consumer client, which then continuously receives events and processes them using the *on_event* function

```python

def save_to_blob_storage(data, file_name):

    try:
        blob_service_client = BlobServiceClient.from_connection_string(CONN_STRING_BLOB)

        blob_client = blob_service_client.get_blob_client(container=CONTAINER_NAME,
                                                          blob=file_name)

        blob_client.upload_blob(data, overwrite=True)
        print(f"Data saved to blob storage as {file_name}")

    except Exception as ex:
        print(f"Failed to save data to blob storage: {ex}")

```

Figure 29: save_to_blob_storage function

```python

def main():
    client = EventHubConsumerClient.from_connection_string(
        conn_str=CONNECTION_STR,
        consumer_group=CONSUMER_GROUP,
        eventhub_name=EVENTHUB_NAME
        )

    with client:
        client.receive(
            on_event=on_event,
            starting_position="-1",
        )

```

Figure 30: Main function

```python

def on_event(partition_context, event):

    data = json.loads(event.body_as_str())
    email = data.get("email")
    message = data.get("message")
    sender = data.get("sender")

    response = requests.post(PREDICT_API_URL, json={"email": email, "message": message, "sender": sender})
    prediction = response.json().get("prediction")

    logging.info(f"Predicted result for {message}: {prediction}")

    print(f"Predicted result for {message}: {prediction}")
    print()

    if prediction == "Spam":
        send_email(email, message, sender)

    data_with_prediction = {
        "email": email,
        "message": message,
        "sender": sender,
        "prediction": prediction
    }

    save_to_blob_storage(json.dumps(data_with_prediction), f"{str(time.time()).replace(".", "_")}_data.json")
    time.sleep(3)
```

Figure 31: on_event function

VII- Notification systems to send Real-time alerts:

For the send email function, we will use two services:

- ✓ Azure Communication Service
- ✓ Azure Email Communication Service

Those two services give us the possibility to send email to users whenever a spam is detected, we will be configuring these two services in the next part.

1. Creating Azure Email Communication Service

First, we will create the Azure Email Communication service

We will navigate to our resource group click on Create and look for Email Communication Service then click on create, fill up the necessary fields and click on Review + Create

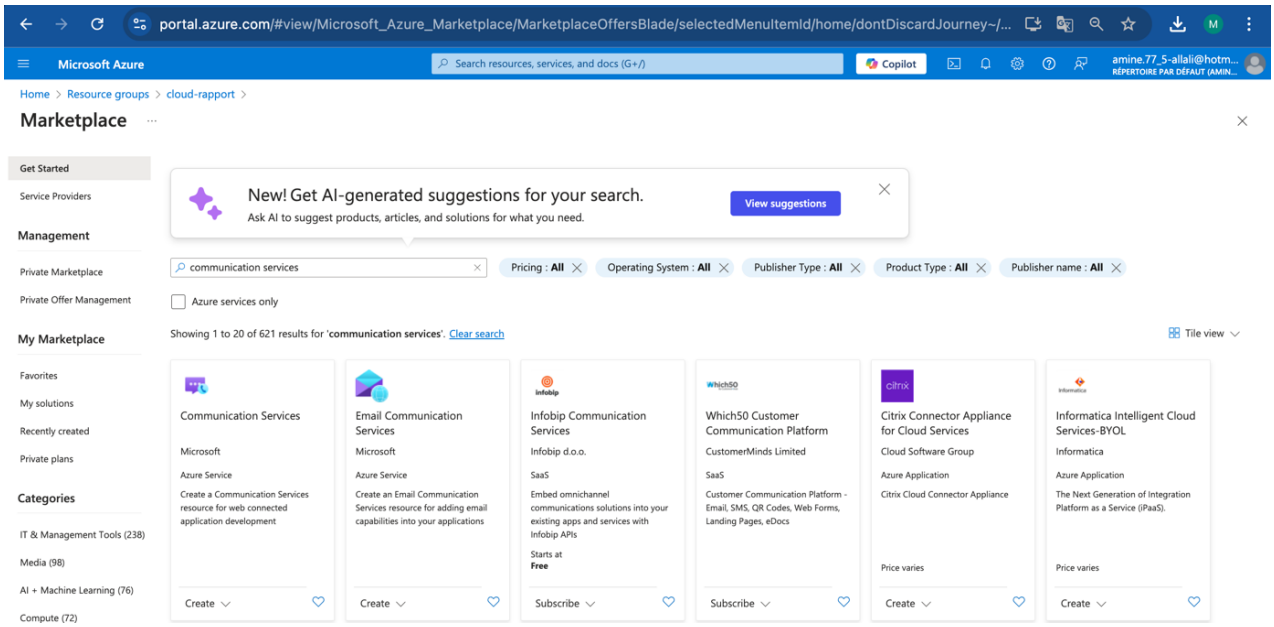


Figure 32: Creating Email Communication Service

We will fill the necessary fields then create the service

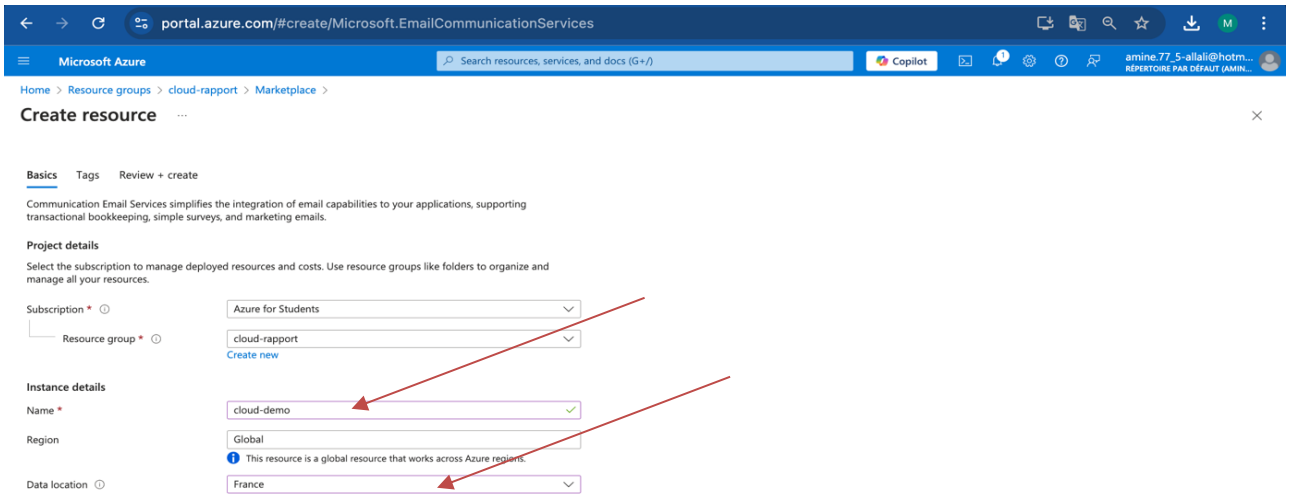


Figure 33: Creating Email Communication Service

We will navigate to provision domains and click on *click add* to add a free azure sub-domain

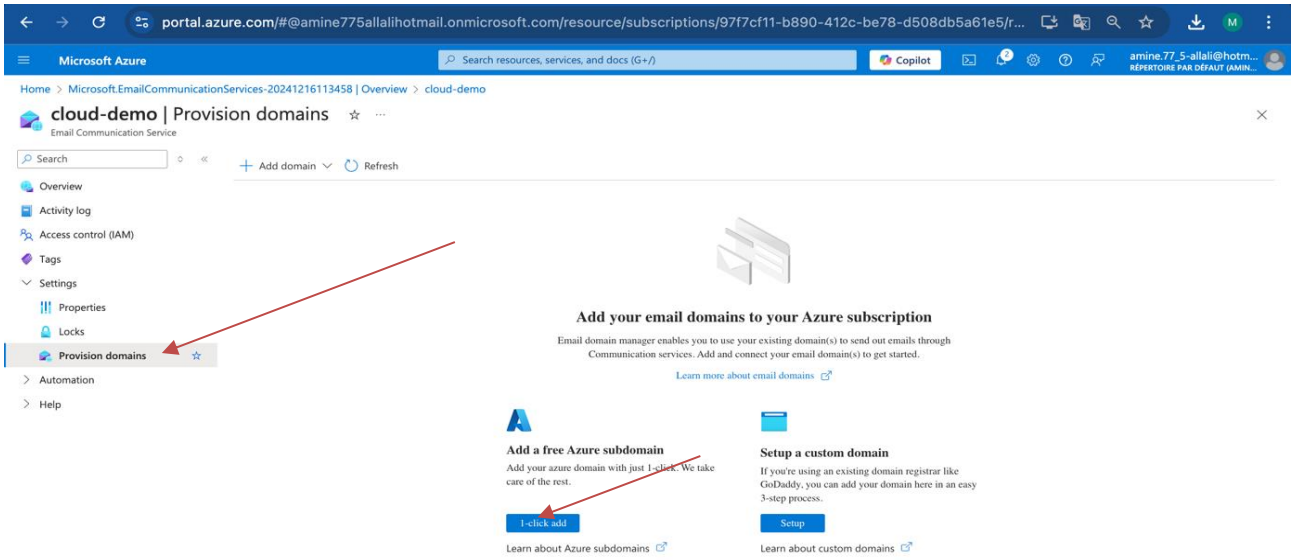


Figure 34: Creating Email Communication Service domain

After that we can see that the domain has been added

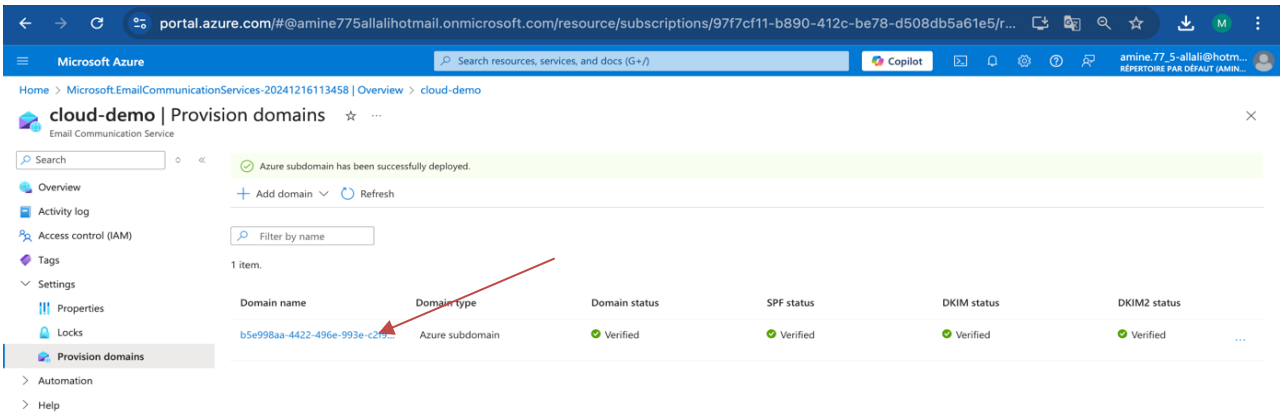


Figure 35: Domain created successfully

After creating the email communication service, we will create a communication service it will be the service that will send email to user whenever a spam is detected in real time.

2. Creating Azure Communication Service

We will navigate to our resource group click on Create and look for Communication Service then click on create, fill up the necessary fields and click on Review + Create

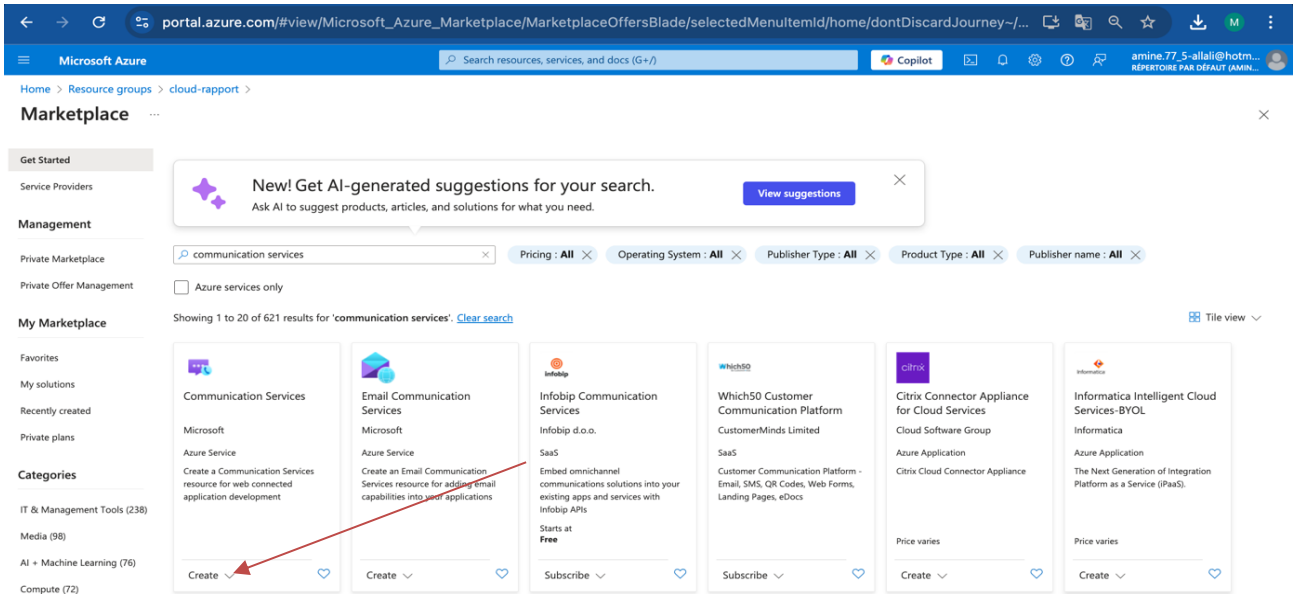


Figure 36: Creating a communication service

We will fill up the name and the data location which means were our data will be stored, then we click on Review + Create

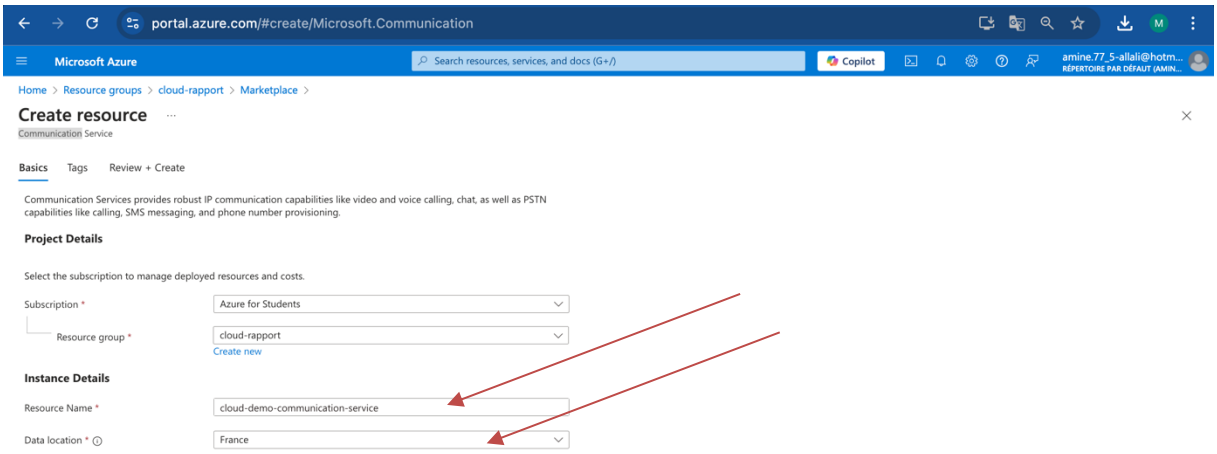


Figure 37: Creating a communication service

After creating the Communication service, we will navigate to email and click on domain then click on connect domain to connect our previous created domain to the communication service.

We will fill up the fields and select our previous created domain and click connect

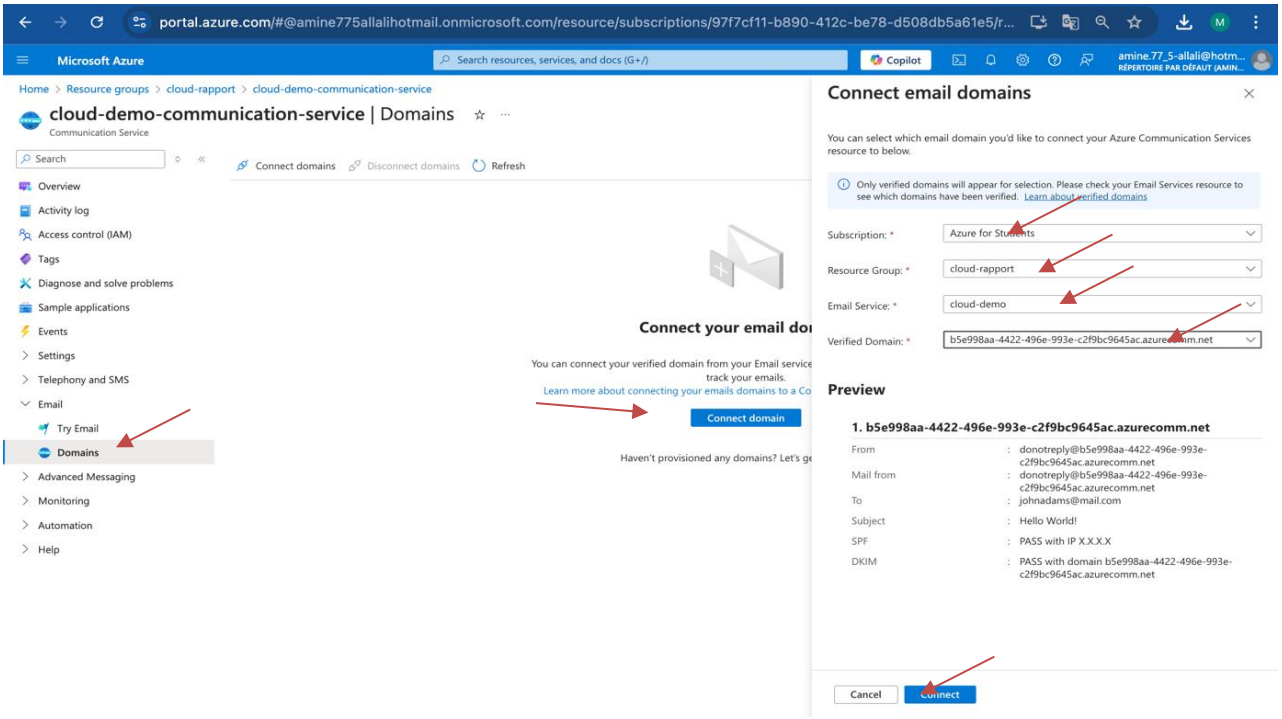


Figure 38: Connect our domain to communication service

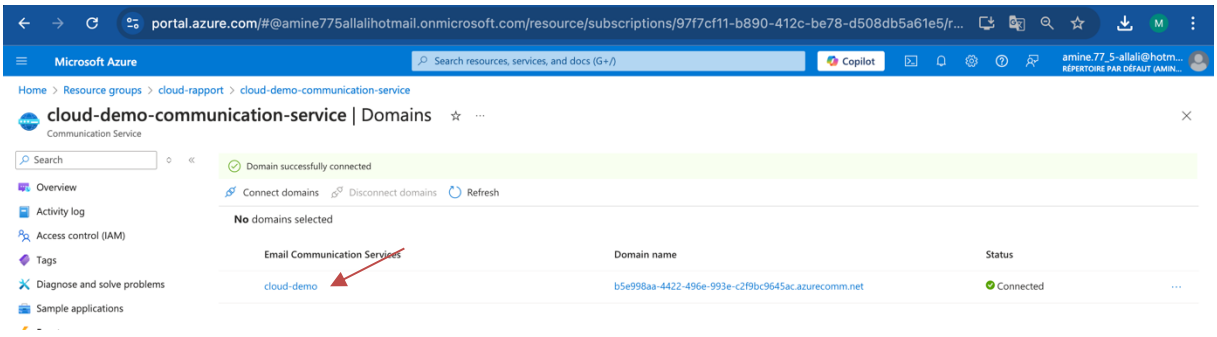


Figure 39: Connection succeeded to our domain

Now we will try sending an email to verify if it works and to do that, we will navigate to try email and fill the fields then we will check our email if anything was sent.

We will specify the email whom we want to send the email to, the subject and the body of the email (message). Finally, we will click on send and a message of success is shown beside the send button.

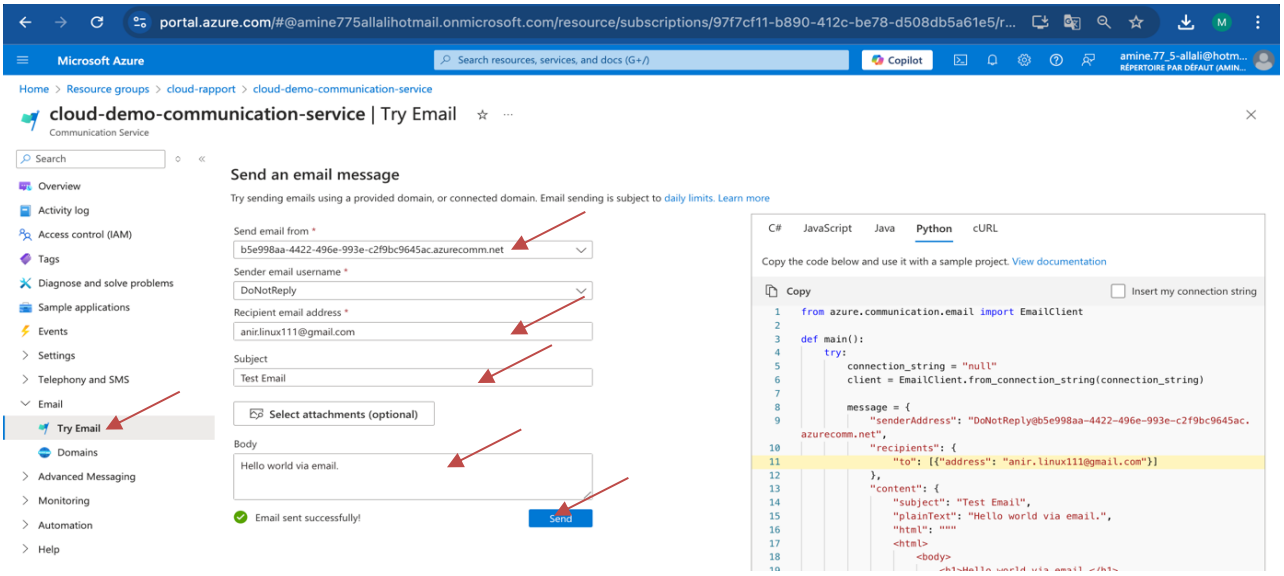
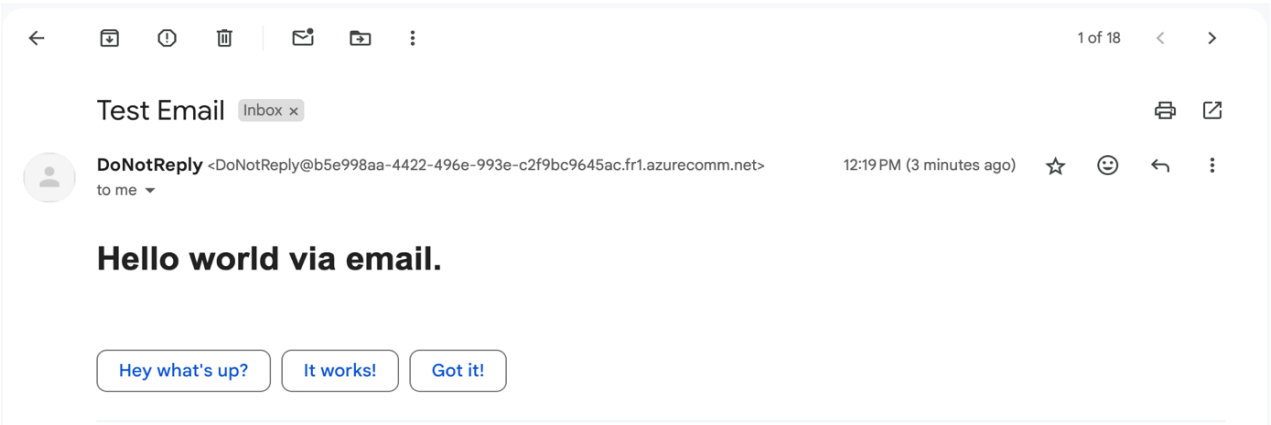


Figure 40: testing email sending

In order to check if it have been received, we will open Gmail and see.



As we can see the email is received successfully

3. Creating The Sending -Email Function

The **send_email** function gives us the possibility to send email alert to user either as a plain text or a html format to decorate the email, to make the function work we will need the connection string and the senderADD (our configured domain)

```python

def send_email(email, message, sender):
    try:
        connection_string = conn_string_email_sender
        client = EmailClient.from_connection_string(connection_string)

        message = {
            "senderAddress": f"{senderADD}",
            "recipients": {
                "to": [{"address": f"{email}"}]
            },
            "content": {
                "subject": "SPAM NOTIFICATION",
                "plainText": "Hello world via email.",
                "html": f"""
                <html>
                    <body>
                        <h3>Hello a <a style="color:red;">SPAM</a> was detected !!</h3>
                        <h4>It was sent by :{sender}</h4>
                        <h4>The Message :</h4>
                        <p>{message}</p>
                    </body>
                </html>"""
            },

        }

        poller = client.begin_send(message)
        result = poller.result()
        print("Message sent")

    except Exception as ex:
        print(ex)

```

Figure 41: send_email function

VIII- Monitoring the App:

For monitoring section of the real-time spam detection system, we used **Azure Event Hubs** to capture and monitor real-time data streams, enabling the system to handle high-throughput message ingestion and efficiently that may indicate processing delays or failures.

Azure App Service provides a scalable platform for hosting the spam detection application, offering integrated monitoring capabilities such as health checks, performance metrics, and resource utilization tracking.

tools like **Azure Application Insights**, the system continuously collects logs, traces, and performance data to identify potential bottlenecks or errors in the spam detection pipeline. These insights, ensuring high availability and responsiveness. Additionally,

1. Azure Event Hub

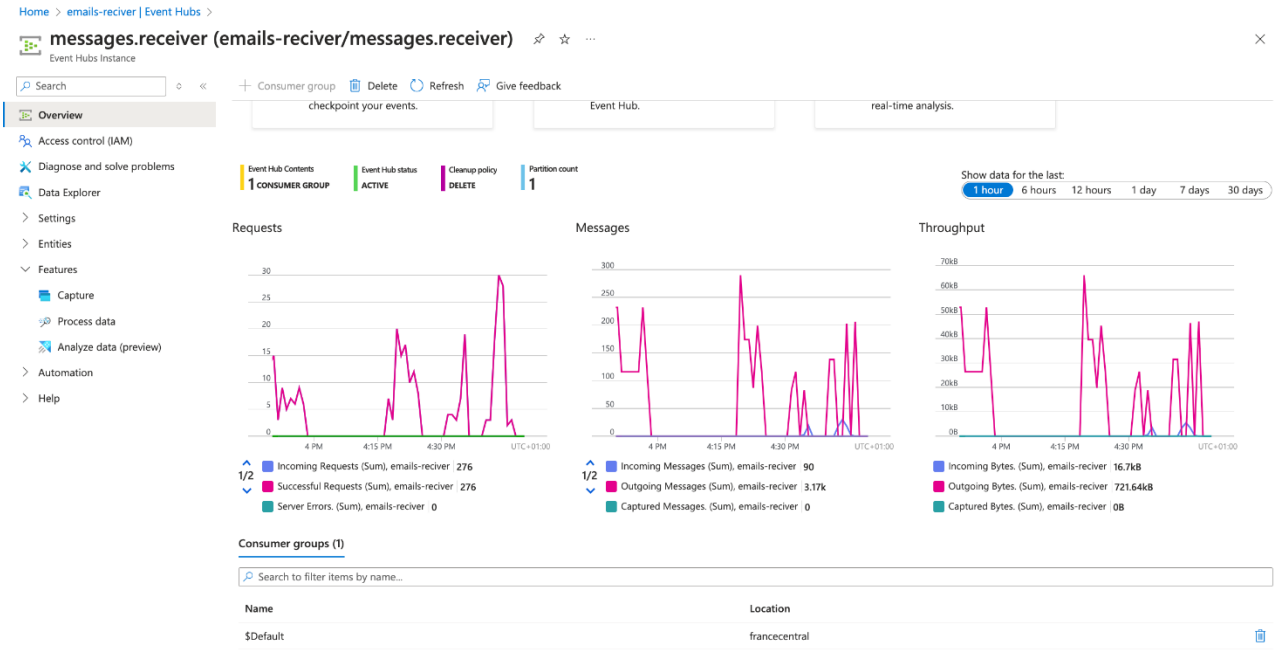


Figure 42 : Monitoring using Azure event hubs

Request: This metric tracks the number of API requests made to Event Hubs, giving us helps us ensure that the system is receiving the expected number of requests and can highlight any unusual spikes or drops in activity.

Message: The message metric focuses on the volume of messages being sent and received by Event Hubs. This is crucial for understanding the message flow and data load. By monitoring the message count, we can detect any discrepancies or performance issues, such as delays in message processing or missed messages.

Throughput: Throughput measures the rate at which data is being processed by Event Hubs, typically measured in megabytes per **second (MBps)**. This helps us monitor the speed and capacity at which the system can handle incoming and outgoing data.

2. Application insights:

Azure Application Insights also provides three key monitoring metrics: Request, Dependency, and Performance.

But before that we have to enable the app insights in our app service:

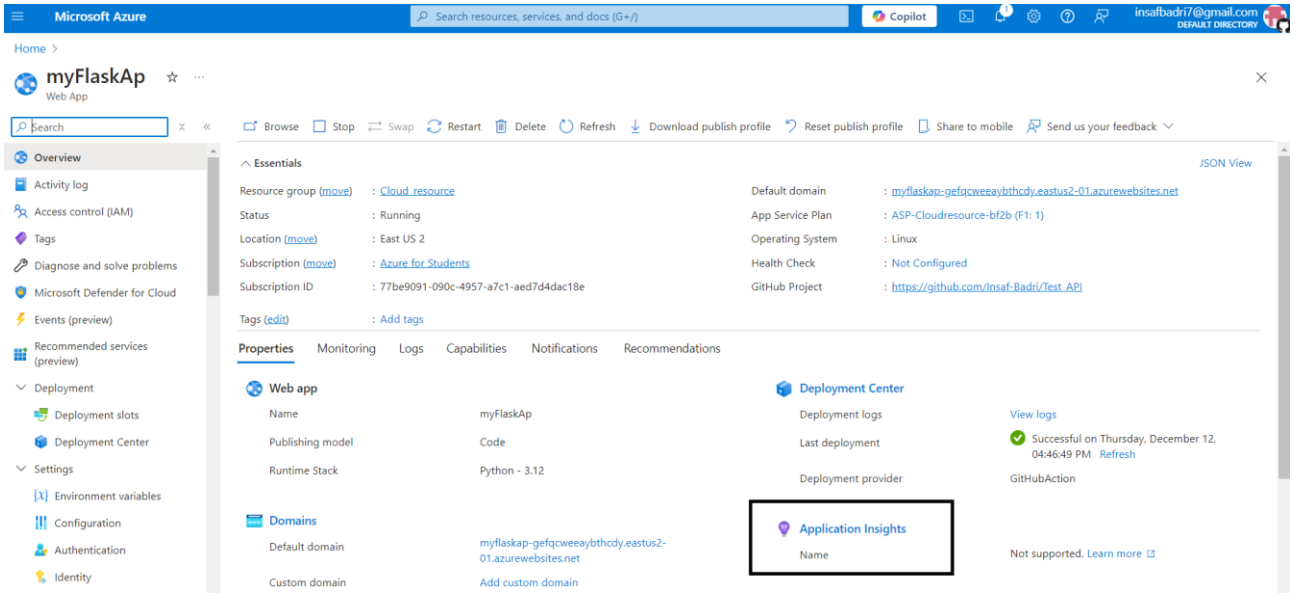


Figure 43: Enable the App insights

Then we can access to our App insights to see the different key of monitoring:

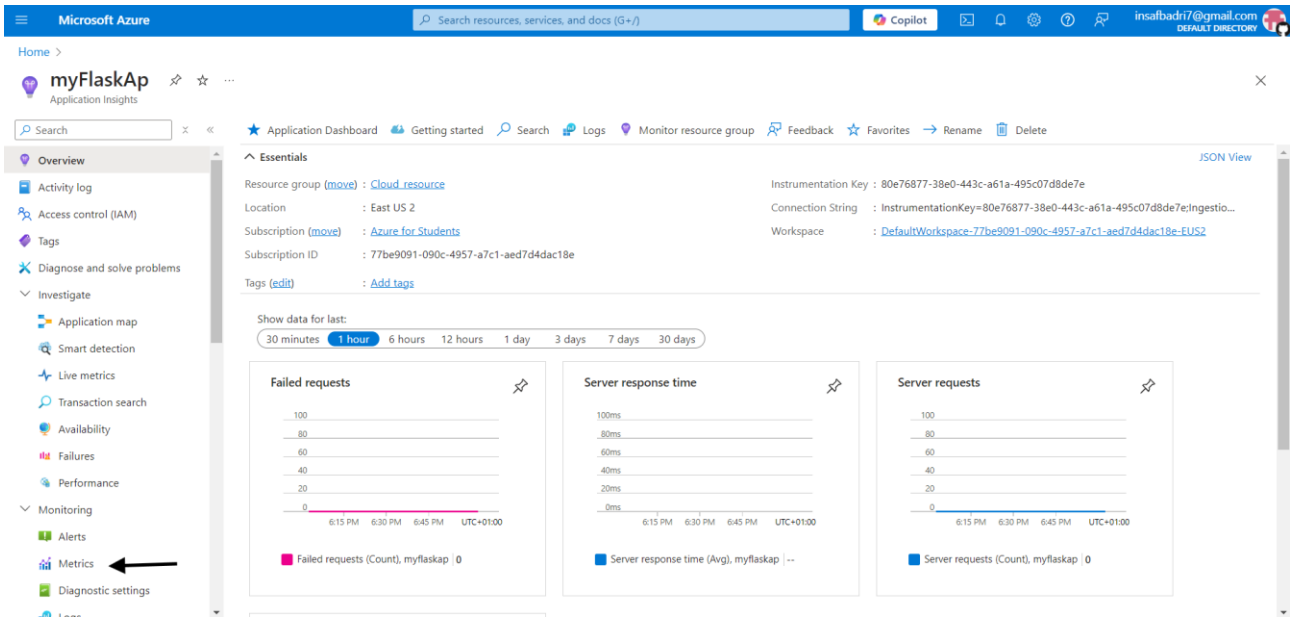


Figure 44: Monitoring using App Insights

3. App services:

Azure App Service monitoring provides three key metrics: Requests, Errors, and Resource Usage.

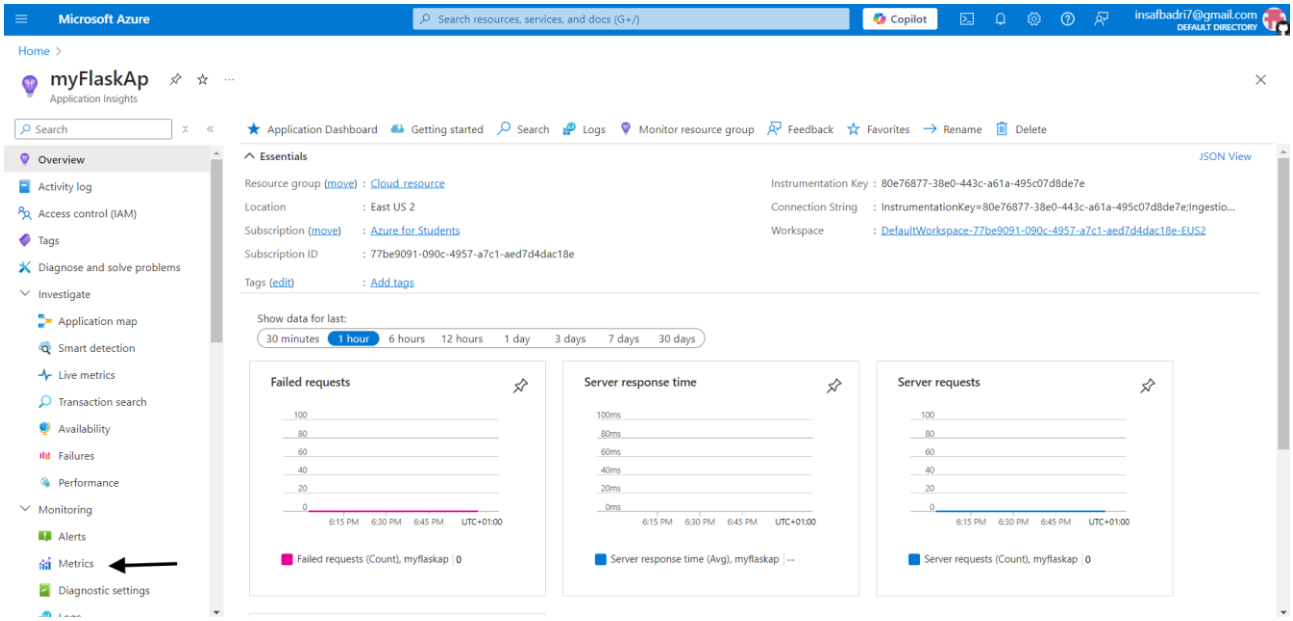


Figure 45: App Service Monitoring

Requests: The request metric tracks the number of requests made to the application. Monitoring requests helps us understand the traffic volume, ensuring the system is responding appropriately to user interactions.

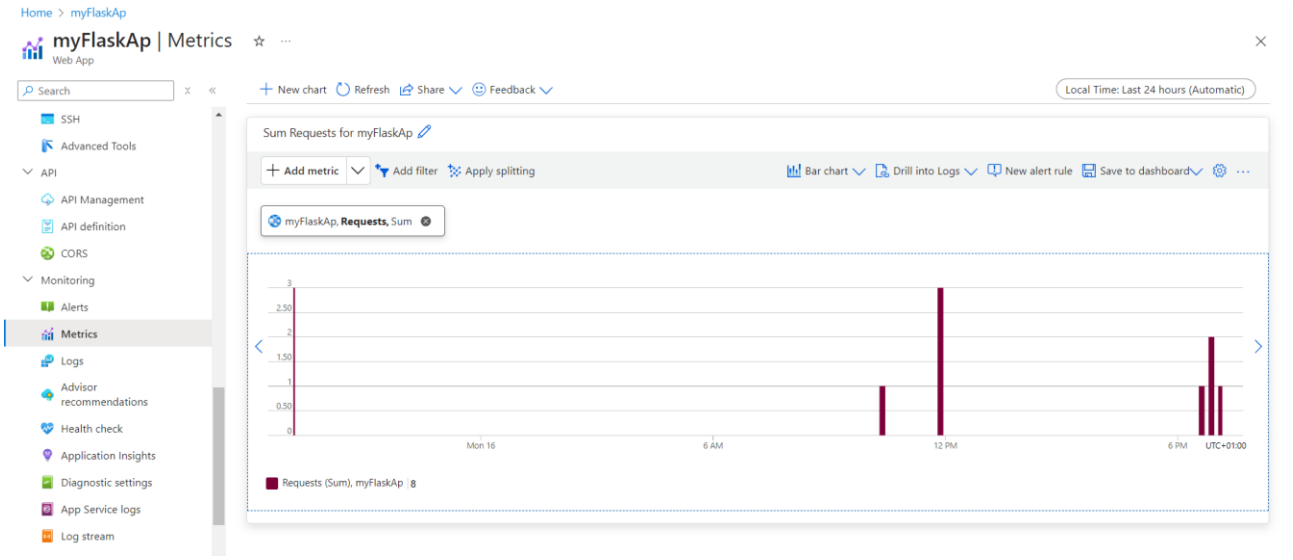


Figure 46: Request metrics

Here we have the Data in:

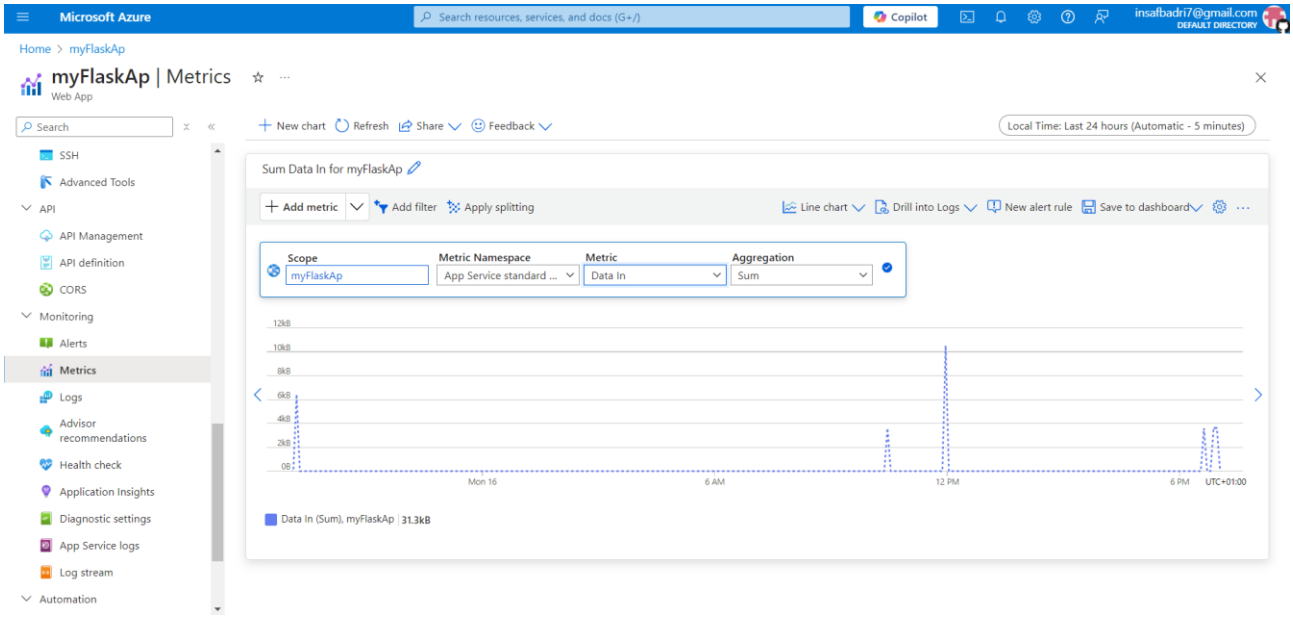


Figure 47: Data in Chart