



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

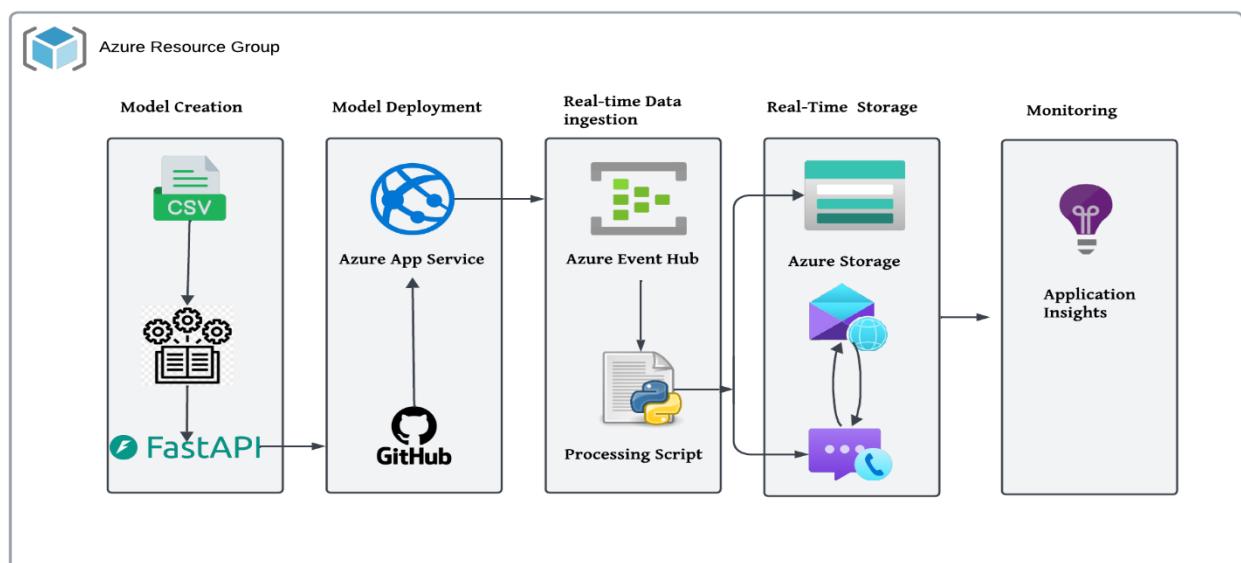
I- Problematic:.....	3
II- Project Architecture.....	3
III- Model Creation:.....	3
1. Data Collection and Processing.....	3
1.1. Collection:.....	3
1.2. Processing.....	4
2. Features selection	4
3. Model training and selection.....	4
IV- Model Deployment:.....	5
1. API creation	5
2. Deployment.....	5
V- Real-time Data Ingestion:	10
1. Creating an API for data	10
2. Configuring Event Hub and The Sending Script	11
VI- Classify Streaming Data in Real-Time:.....	18
1. Creating a Blob storage to save data	18
2. Creating the Script	20
VII- Notification systems to send Real-time alerts:	22
1. Creating Azure Email Communication Service	22
2. Creating Azure Communication Service	24
3. Creating The Sending -Email Function	27
VIII- Monitoring the App:.....	28
1. Azure Event Hub.....	28
2. Application insights:.....	29
3. App services:	30

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

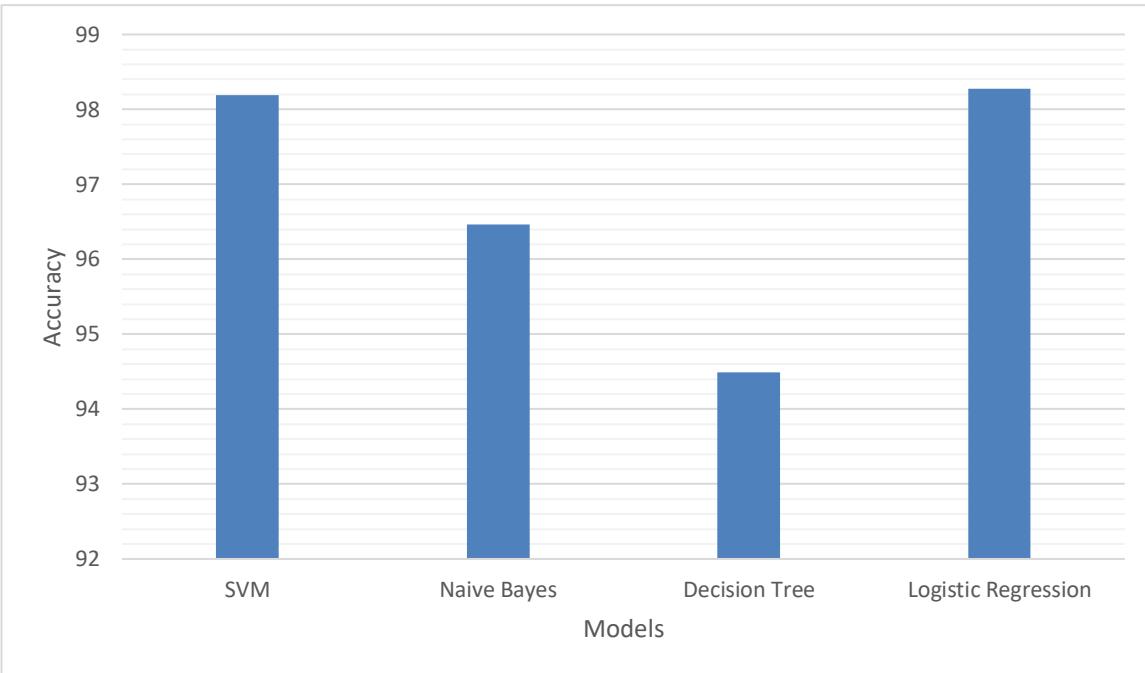


Figure 1: Accuracy of Models on test set

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

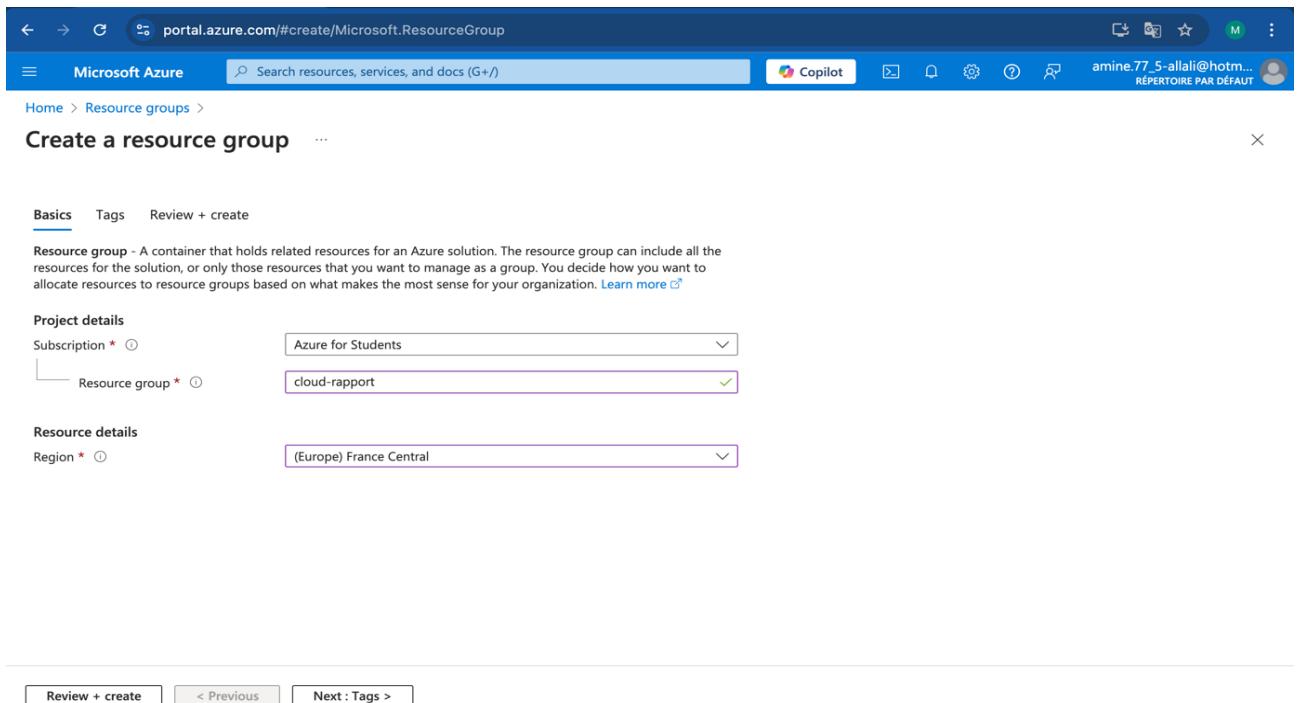


Figure 2: Creating a resource group

The screenshot shows the 'Resource groups' page in the Azure portal. A red arrow points to the 'cloud-rapport' resource group in the list, which is highlighted. The table lists three resource groups:

Name	Subscription	Location	...
cloud-rapport	Azure for Students	France Central	...
cloud-rg	Azure for Students	France Central	...
DefaultResourceGroup-PAR	Azure for Students	France Central	...

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

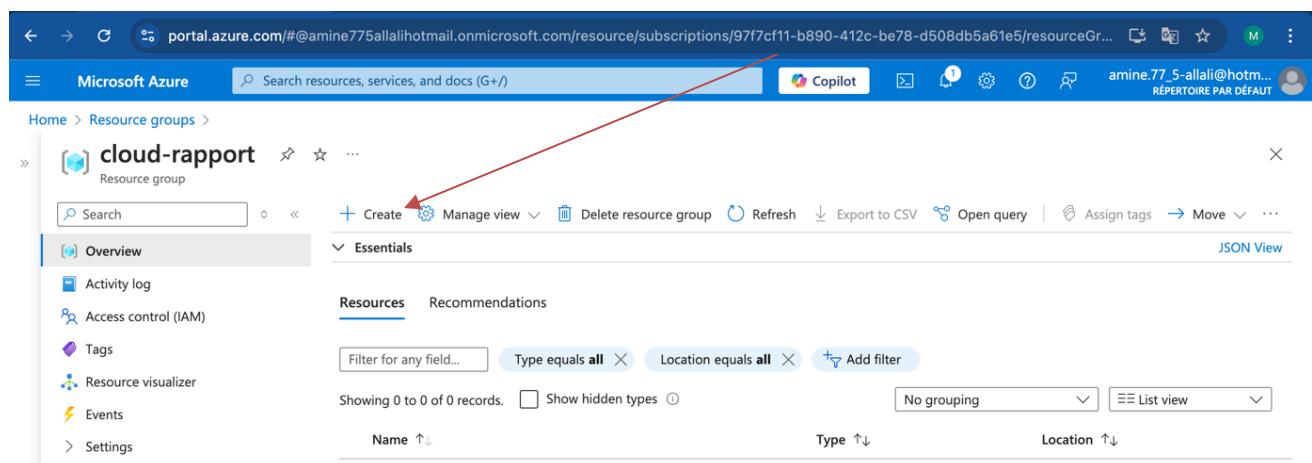


Figure 4: creating a new resource

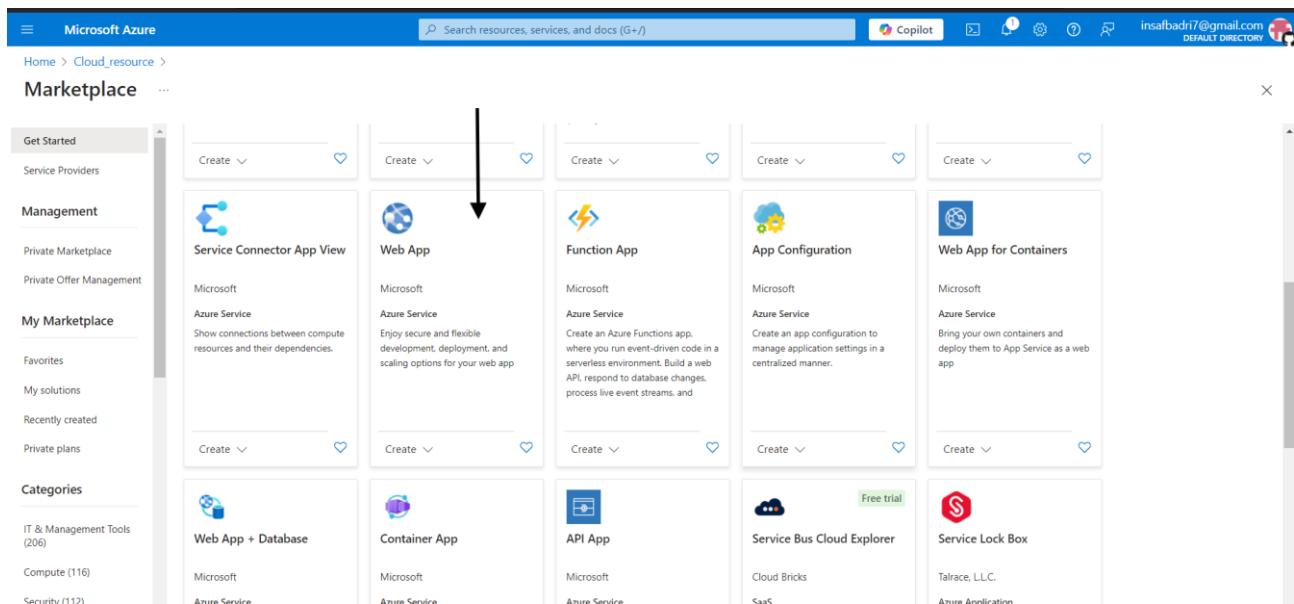


Figure 5: creating a Web App

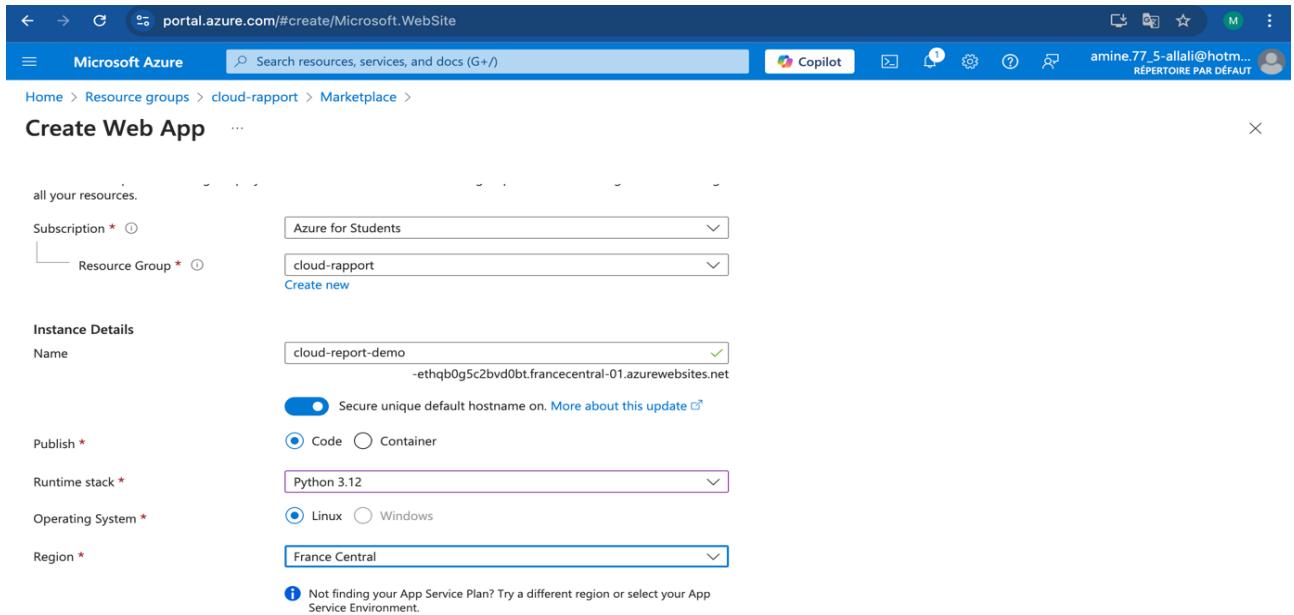


Figure 6: configuring the Web App

Pricing plans

App Service plan pricing tier determines the location, features, cost and compute resources associated with your app.
[Learn more](#)

Linux Plan (France Central) *	(New) cloud-report-demo
Create new	
Pricing plan	Free F1 (Shared infrastructure)
Explore pricing plans	

Figure 7: configuring the Web App

The screenshot shows the 'Overview' page for the deployed web app. Deployment details include deployment name, subscription, start time, and correlation ID. Next steps include managing deployments and protecting the app with authentication. Side panels provide links to Cost Management, Microsoft Defender for Cloud, and free Microsoft tutorials.

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

cloud-report-demo | Deployment Center

Source* GitHub

Repository* cloud-report-demo

Branch* main

Figure 9: Deployment of the Web App

cloud-report-demo | Configuration

Startup Command unicorn main.app --host 0.0.0.0 --port 8000

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

Time	Commit ID	Logs	Commit Author	Status	Message
Sunday, December 15, 2024 (3)	8d07749	App Logs	N/A	Success (Active)	OneDeploy
12/15/2024, 8:54:13 PM +01:00	2bb0ad6	Build/Deploy Log	alialamine	Success	Add or update the Azure App Service build and deployment workflow config
12/15/2024, 8:52:54 PM +01:00					

Figure 11: Checking if deployment is succeeded

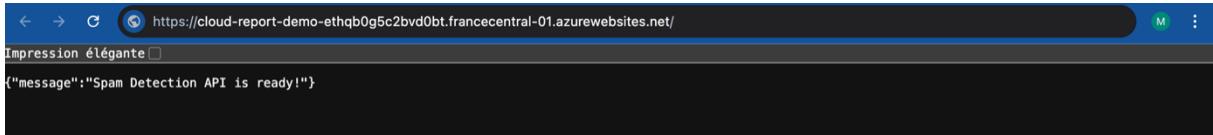


Figure 12: API

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

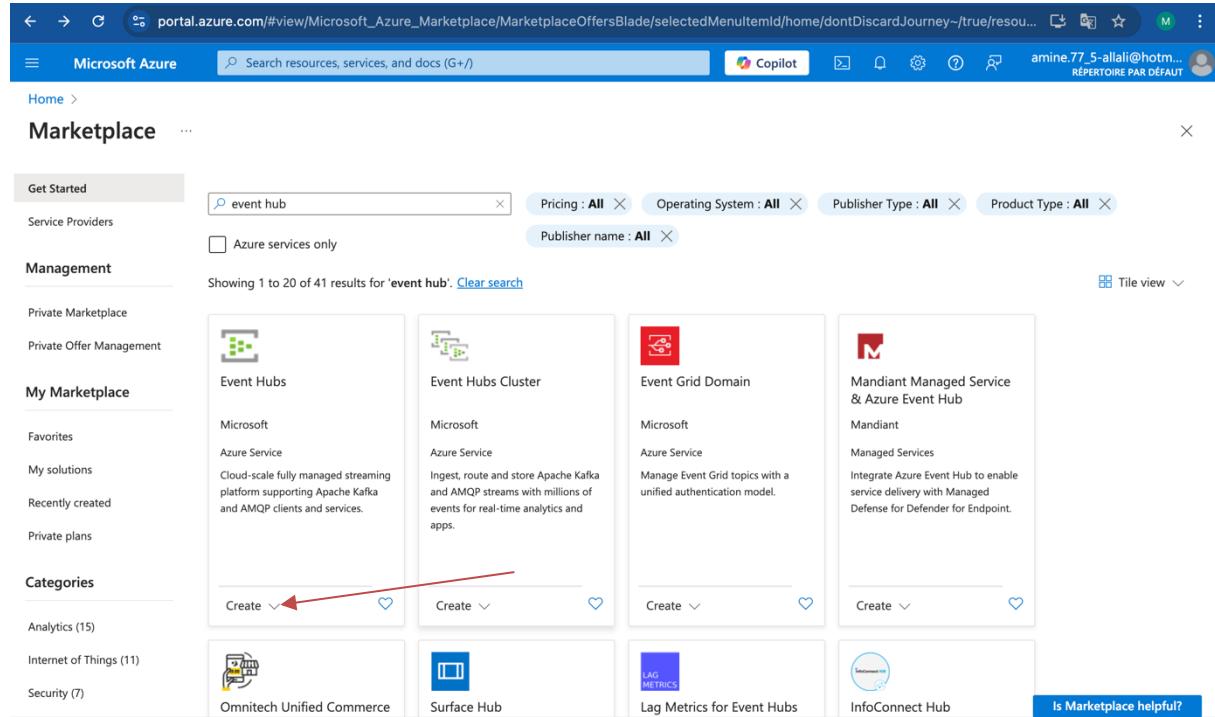
We created a custom API integrated into the same *main.py* file as our previously deployed prediction API. This new API, accessible through the route */send-email*, dynamically fetches and returns data from a CSV dataset. Each time the API is called, it randomly selects a row from the CSV file and returns 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

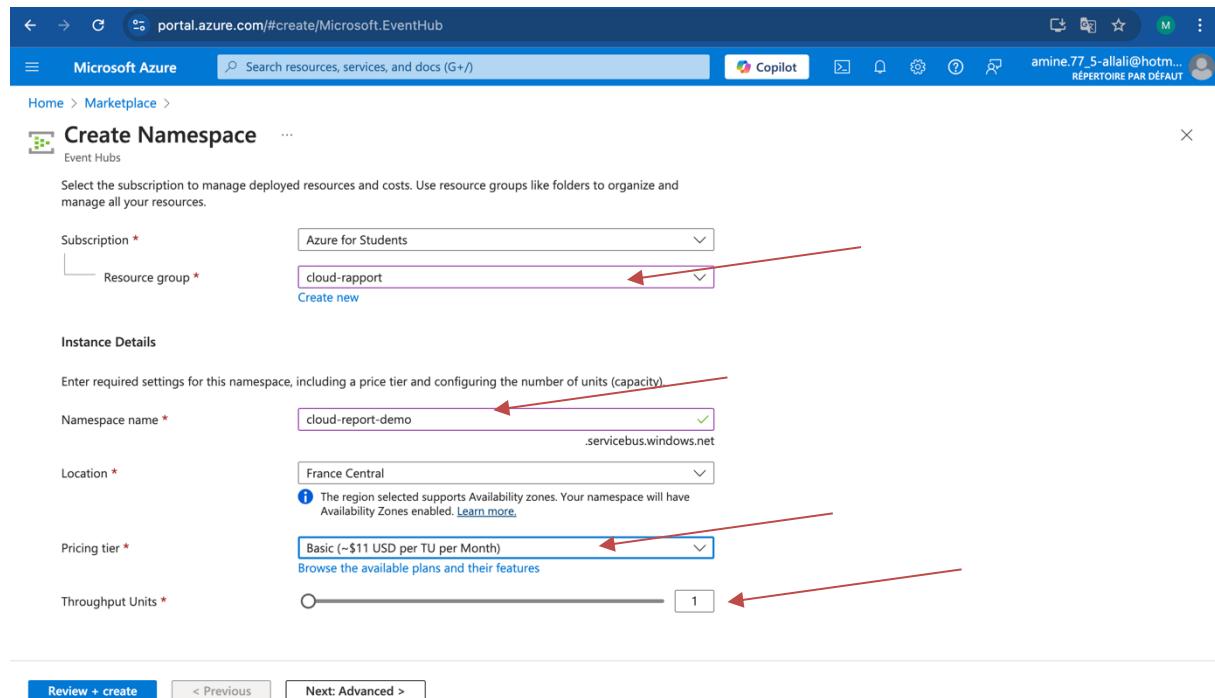
- ✓ Creating an Event Hub namespace

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



The screenshot shows the Microsoft Azure Marketplace search results for 'event hub'. The search bar at the top contains 'event hub'. Below the search bar, there are filters: 'Pricing : All', 'Operating System : All', 'Publisher Type : All', 'Product Type : All', and 'Publisher name : All'. There is also a checkbox for 'Azure services only'. The results section shows four items: 'Event Hubs' (Microsoft), 'Event Hubs Cluster' (Microsoft), 'Event Grid Domain' (Microsoft), and 'Mandiant Managed Service & Azure Event Hub' (Mandiant). Each item has a 'Create' button. A red arrow points to the 'Create' button for 'Event Hubs'.

Figure 13: Creating an Event Hub



The screenshot shows the 'Create Namespace' wizard in the Microsoft Azure portal. The first step is 'Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.' It shows a dropdown for 'Subscription' (Azure for Students) and a dropdown for 'Resource group' (cloud-report). A red arrow points to the 'Resource group' dropdown. The second step is 'Instance Details'. It asks for 'Enter required settings for this namespace, including a price tier and configuring the number of units (capacity.)'. It includes fields for 'Namespace name' (cloud-report-demo), 'Location' (France Central), 'Pricing tier' (Basic), and 'Throughput Units' (1). Red arrows point to each of these fields. At the bottom, there are buttons for 'Review + create', '< Previous', and 'Next: Advanced >'.

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.

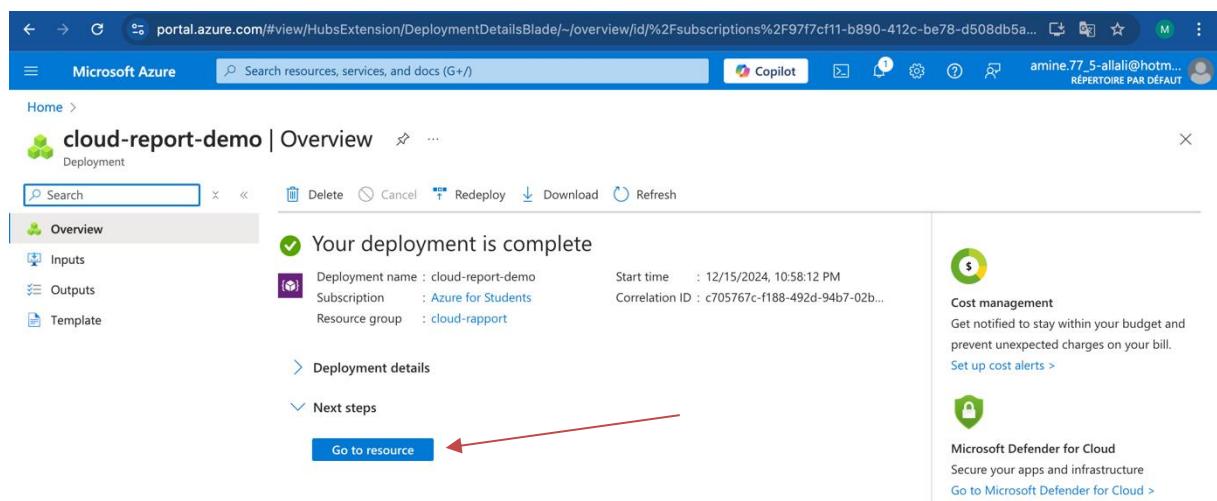


Figure 14: Creation of the Event Hub succeeded

✓ 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

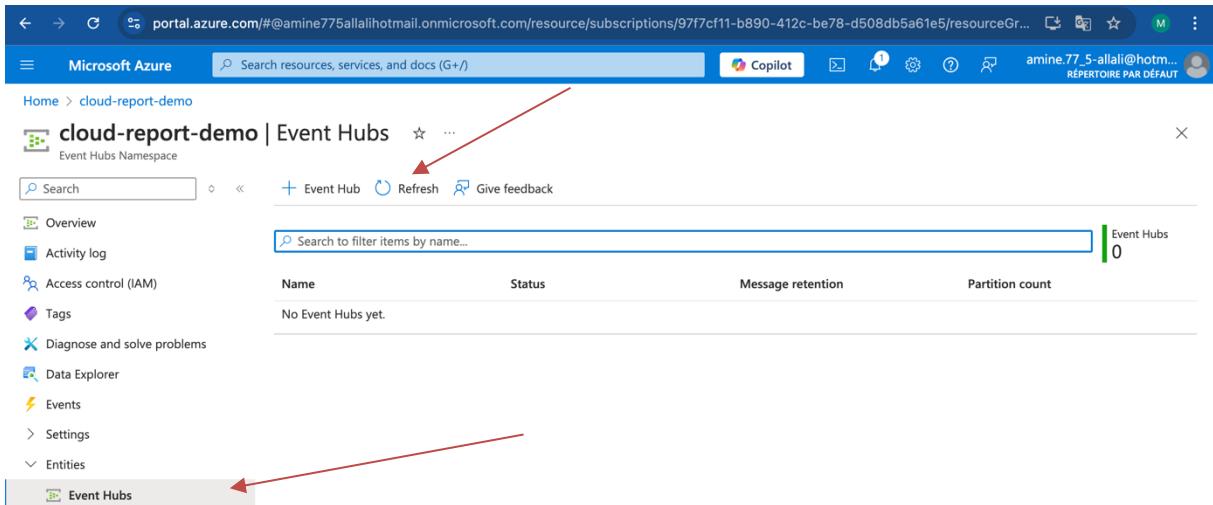


Figure 15: Creation of an Event Hub

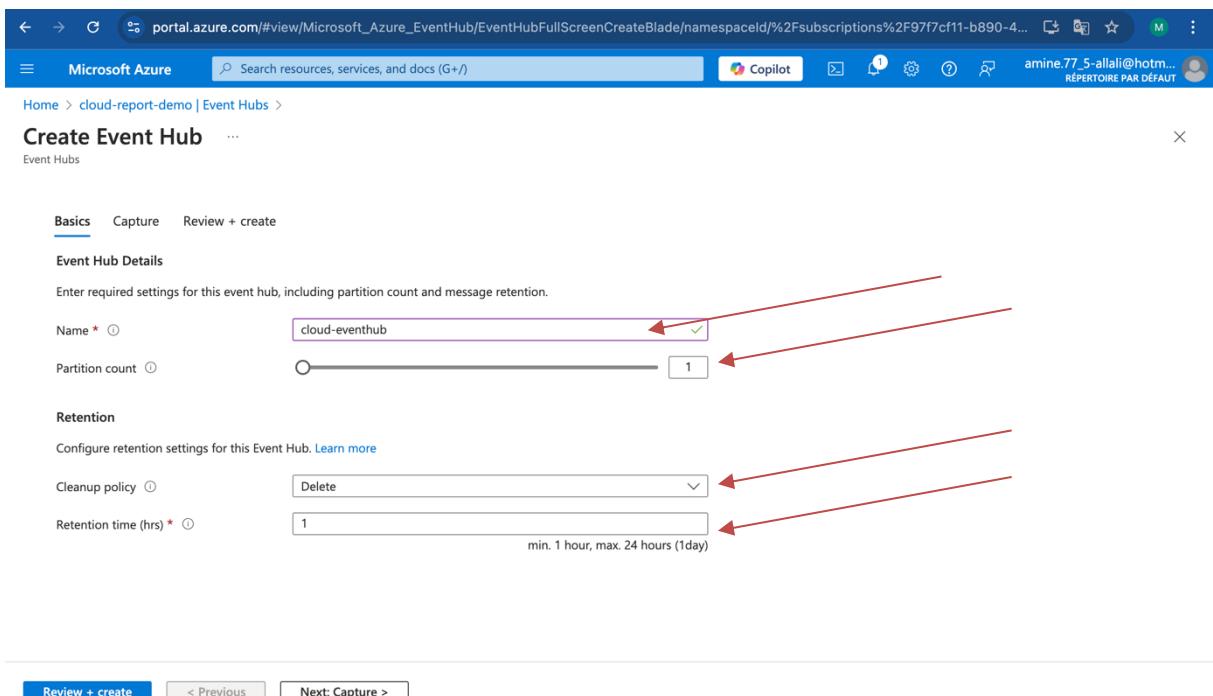


Figure 16: Creation of an Event Hub

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

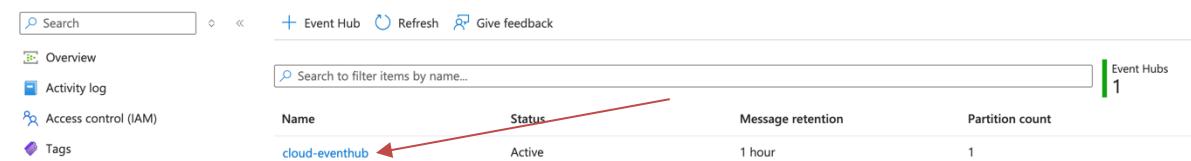


Figure 17: Creation of the Event Hub succeeded

✓ 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.

The screenshot shows the Azure Event Hubs Overview page for the 'cloud-eventhub' instance. The left sidebar includes options like Overview, Access control (IAM), Diagnose and solve problems, Data Explorer, Settings (with Shared access policies selected), Configuration, Properties, Locks, Entities, Features, Capture, Process data, Analyze data (preview), Automation, and Help. The main content area displays resource details: Resource group (move) to 'cloud-rapport', Location to 'France Central', Subscription (move) to 'Azure for Students', Subscription ID to '97f7cf11-b890-412c-be78-d508db5a61e5', Partition count to '1', Status to 'Active', Created on 'Sunday, December 15, 2024 at 23:18:31 GMT+1', Updated on 'Sunday, December 15, 2024 at 23:18:31 GMT+1', and Cleanup policy to 'Delete'. Below the details are four cards: 'Capture events', 'Process data', 'Connect', and 'Checkpoint'. A red arrow points from the 'Shared access policies' link in the sidebar to the 'Add' button in the top right of the main content area.

Figure 18: Event Hub Overview

The screenshot shows the 'Shared access policies' page for the 'cloud-eventhub' instance. The left sidebar is identical to Figure 18. The main content area has an 'Add' button highlighted with a red arrow. Below it is a search bar labeled 'Search to filter items...'. The 'Policy' section shows the message 'no policies have been set up yet.' The 'Claims' section is also visible.

Figure 19: Event Hub Shared access policies

The screenshot shows the Microsoft Azure portal interface for creating a Shared Access Policy. On the left, there's a sidebar with various options like Overview, Access control (IAM), and Shared access policies (which is currently selected). The main area shows a list of policies under 'cloud-eventhub'. A new policy is being created with the name 'key1'. The 'Claims' section is empty. On the right, there's a modal window titled 'Add SAS Policy' with fields for 'Policy name' (set to 'key1'), checkboxes for 'Manage', 'Send', and 'Listen' (all checked), and a 'Create' button at the bottom.

Figure 20: Creating an Event Hub Shared access policies

This screenshot shows the details of the 'key1' Shared Access Policy. The policy is listed under the 'cloud-eventhub' Event Hub. The 'Policy' section shows the name 'key1'. The 'Claims' section lists 'Manage', 'Send', and 'Listen' permissions. Below the policy details, there are fields for 'Primary key' and 'Secondary key', each containing a long hex-encoded string. There are also sections for 'Connection string-primary key' and 'Connection string-secondary key', each with a corresponding URL. A checkbox for 'Show AMQP connection strings' is present but unchecked.

Figure 21: View the Event Hub Shared access policies created

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

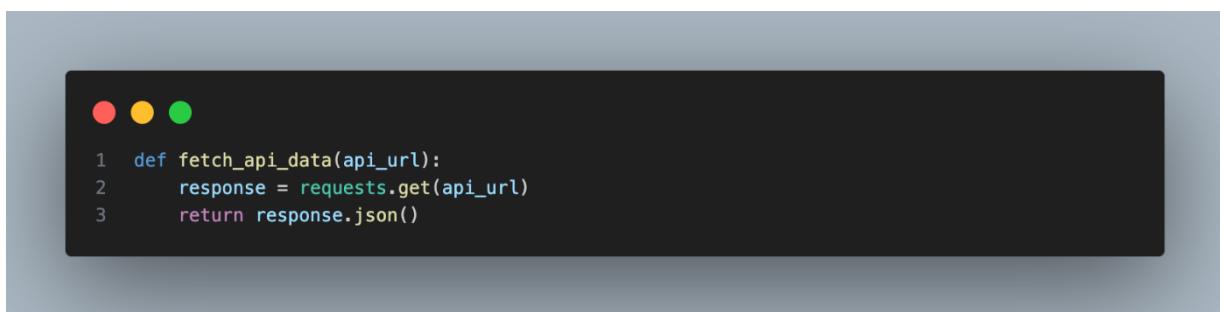
To consume and ingest data into our Event Hub, we will use the Python script below, leveraging the Azure Event Hub library. The script defines three functions

- ✓ **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.



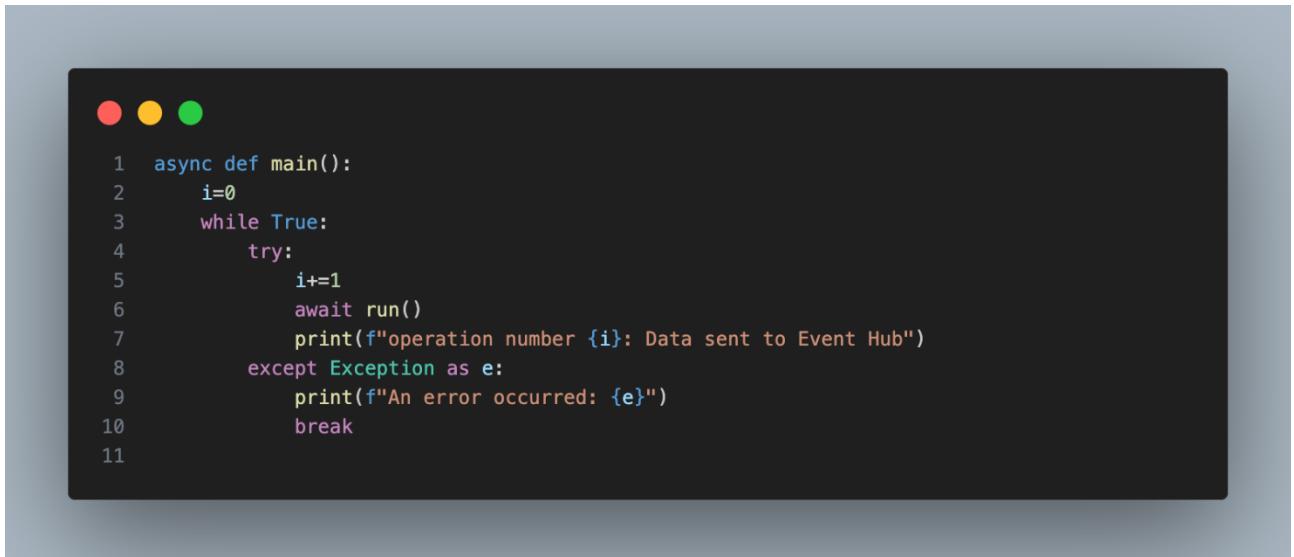
```
● ● ●
1  async def run():
2      producer = EventHubProducerClient.from_connection_string(
3          conn_str=EVENT_HUB_CONNECTION_STR,
4          eventhub_name=EVENT_HUB_NAME
5      )
6
7      async with producer:
8
9          api_url = "https://spam-api-e7c4ayf6dfcvdfh.francecentral-01.azurewebsites.net/send-email"
10         api_response = fetch_api_data(api_url)
11
12         api_response = fetch_api_data(api_url)
13         event_data = json.dumps(api_response)
14
15         event_data_bytes = event_data.encode("utf-8")
16
17         event_data_batch = await producer.create_batch()
18         event_data_batch.add(EventData(event_data_bytes))
19
20         await producer.send_batch(event_data_batch)
21
22
23
```

Figure 22: run function that send data



```
● ● ●
1  def fetch_api_data(api_url):
2      response = requests.get(api_url)
3      return response.json()
```

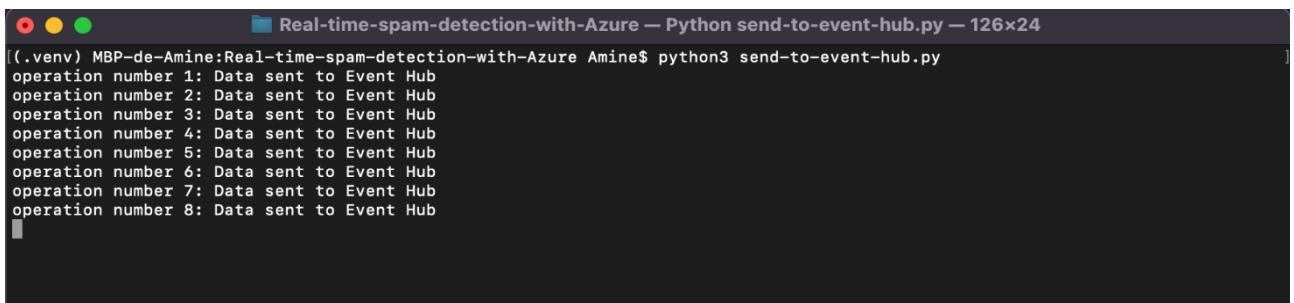
Figure 23: run function that send data



```
1  async def main():
2      i=0
3      while True:
4          try:
5              i+=1
6              await run()
7              print(f"operation number {i}: Data sent to Event Hub")
8          except Exception as e:
9              print(f"An error occurred: {e}")
10             break
11
```

Figure 24: main function

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



```
(.venv) MBP-de-Amine:Real-time-spam-detection-with-Azure Amine$ python3 send-to-event-hub.py
operation number 1: Data sent to Event Hub
operation number 2: Data sent to Event Hub
operation number 3: Data sent to Event Hub
operation number 4: Data sent to Event Hub
operation number 5: Data sent to Event Hub
operation number 6: Data sent to Event Hub
operation number 7: Data sent to Event Hub
operation number 8: Data sent to Event Hub
```

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:

The screenshot shows the Microsoft Azure portal with the URL <https://portal.azure.com/#@amine775allali@hotmail.onmicrosoft.com/resource/subscriptions/97f7cf11-b890-412c-be78-d508db5a61e5/resourceGroups/messagesreceiver/providers/Microsoft.EventHub/eventHubs/messages.receiver>. The page title is "messages.receiver (emails-reciever/messages.receiver) | Data Explorer". The left sidebar is expanded to show the "Data Explorer" section, which is highlighted with a blue border. A red arrow points from the "Data Explorer" link to the "View events" button at the bottom of the sidebar. Another red arrow points from the "Content Type" column header in the main table to the "Event body" tab below it. The main content area displays a table of received events with columns: Sequence Number, Offset, Partition ID, Enqueued Time, and Content Type. The "Content Type" column is currently empty. The "Event body" tab is selected.

Sequence Nu...	Offset	Partition ID	Enqueued Time	Content Type
1039	215216	0	lun. 16 déc. 24, 12:05:23 AM UTC...	
1040	215456	0	lun. 16 déc. 24, 12:05:25 AM UTC...	
1041	215704	0	lun. 16 déc. 24, 12:05:27 AM UTC...	
1042	215912	0	lun. 16 déc. 24, 12:05:29 AM UTC...	
1043	216152	0	lun. 16 déc. 24, 12:05:30 AM UTC...	
1044	216392	0	lun. 16 déc. 24, 12:05:32 AM UTC...	
1045	216624	0	lun. 16 déc. 24, 12:05:34 AM UTC...	
1046	216848	0	lun. 16 déc. 24, 12:05:36 AM UTC...	
1047	217056	0	lun. 16 déc. 24, 12:05:37 AM UTC...	
1048	217304	0	lun. 16 déc. 24, 12:05:39 AM UTC...	

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

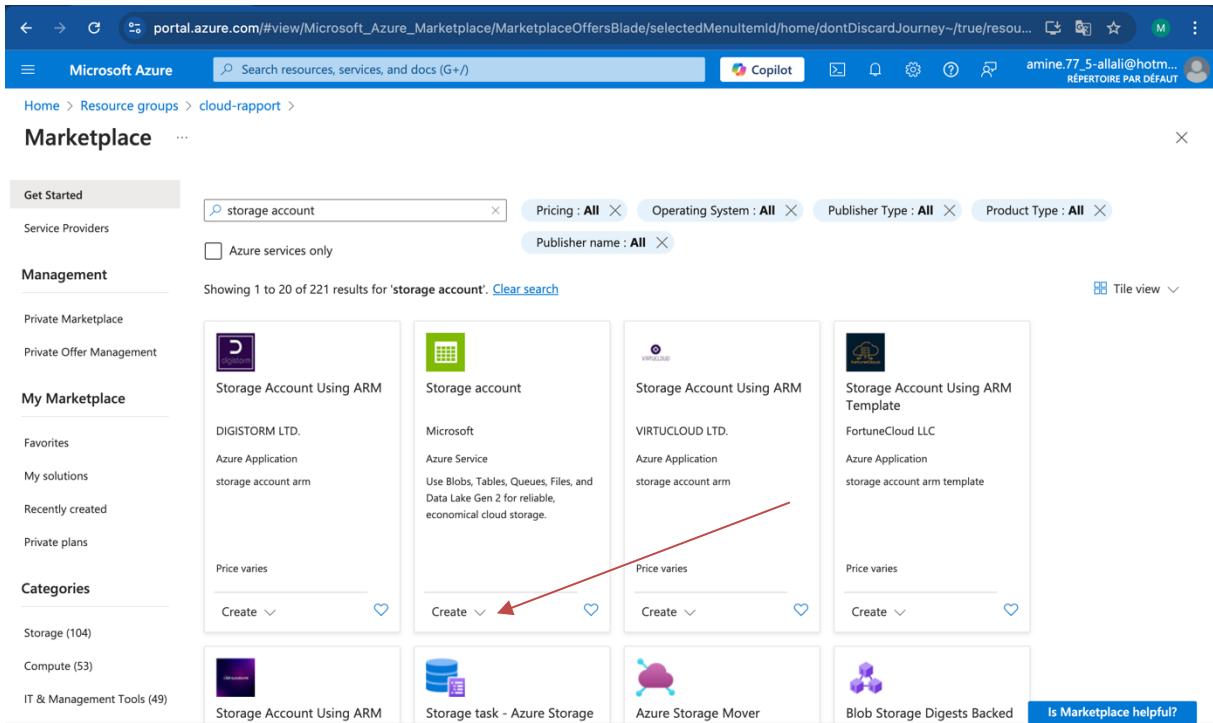


Figure 26: Creating a storage account

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.

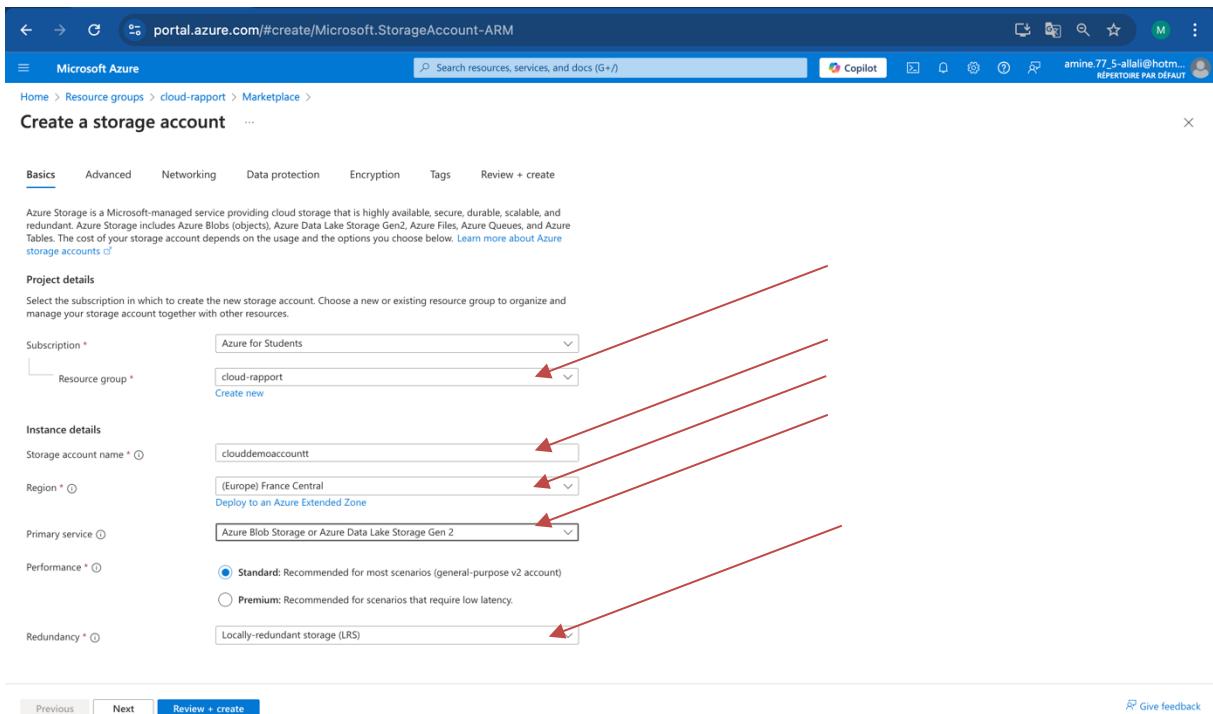


Figure 27: Creating a storage account (Blob storage)

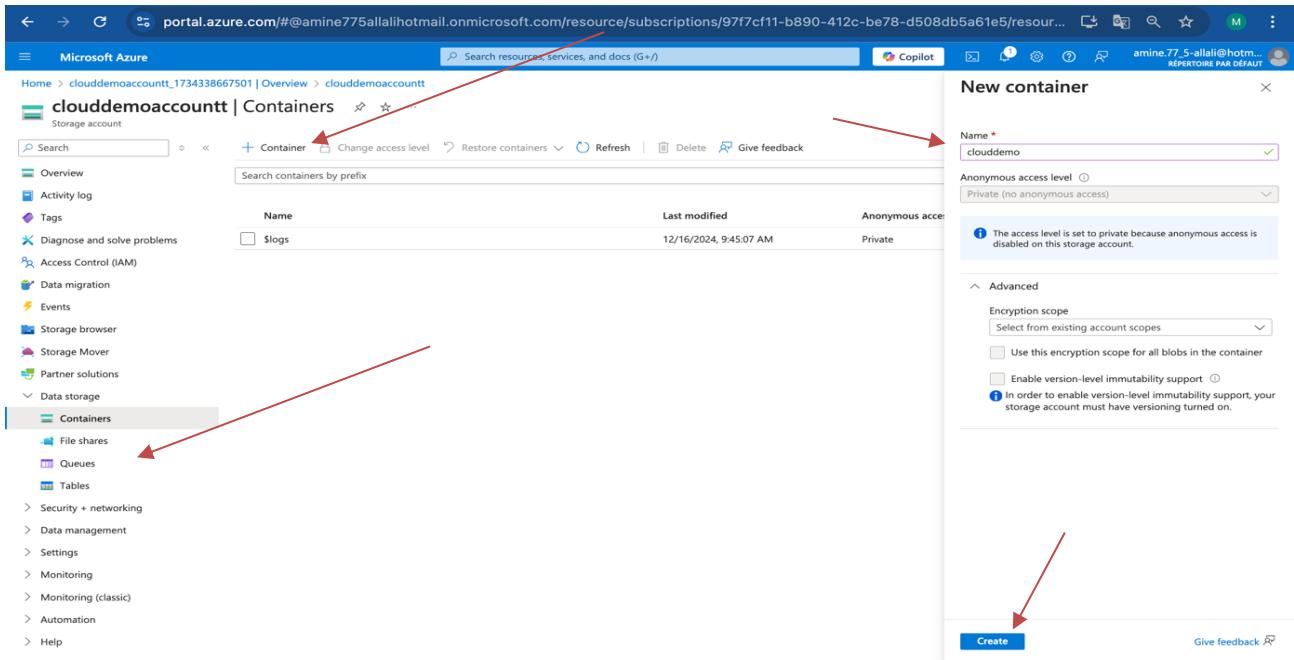


Figure 28: Creating a container to store our Json files

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

```

1  def save_to_blob_storage(data, file_name):
2
3      try:
4          blob_service_client = BlobServiceClient.from_connection_string(CONN_STRING_BLOB)
5
6          blob_client = blob_service_client.get_blob_client(container=CONTAINER_NAME,
7                                              blob=file_name)
8
9
10         blob_client.upload_blob(data, overwrite=True)
11         print(f"Data saved to blob storage as {file_name}")
12
13     except Exception as ex:
14         print(f"Failed to save data to blob storage: {ex}")
15

```

Figure 29: `save_to_blob_storage` function



```
1 def main():
2     client = EventHubConsumerClient.from_connection_string(
3         conn_str=CONNECTION_STR,
4         consumer_group=CONSUMER_GROUP,
5         eventhub_name=EVENTHUB_NAME
6     )
7
8     with client:
9         client.receive(
10             on_event=on_event,
11             starting_position="-1",
12         )
13
14
```

Figure 30: Main function



```
1
2 def on_event(partition_context, event):
3
4     data = json.loads(event.body_as_str())
5     email = data.get("email")
6     message = data.get("message")
7     sender = data.get("sender")
8
9     response = requests.post(PREDICT_API_URL, json={"email": email, "message": message,
10 "sender": sender})
11     prediction = response.json().get("prediction")
12
13     logging.info(f"Predicted result for {message}: {prediction}")
14
15     print(f"Predicted result for {message}: {prediction}")
16     print()
17
18     if prediction == "Spam":
19         send_email(email, message, sender)
20
21     data_with_prediction = {
22         "email": email,
23         "message": message,
24         "sender": sender,
25         "prediction": prediction
26     }
27
28     save_to_blob_storage(json.dumps(data_with_prediction), f"{str(time.time())}.replace
29     (".", "_")}_data.json")
30     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

The screenshot shows the Azure Marketplace search results for 'communication services'. The search bar at the top contains 'communication services'. Below the search bar, there are filters: Pricing: All, Operating System: All, Publisher Type: All, Product Type: All, and Publisher name: All. A checkbox for 'Azure services only' is also present. The results section displays six items:

Service Provider	Service Name	Description	Pricing	Actions	
Microsoft	Communication Services	Create a Communication Services resource for web connected application development	Create	Heart icon	
Microsoft	Email Communication Services	Create an Email Communication Services resource for adding email capabilities into your applications	Create	Heart icon	
Infobip d.o.o.	Infobip Communication Services	Embed omnichannel communications solutions into your existing apps and services with Infobip APIs	Starts at Free	Subscribe	Heart icon
CustomerMinds Limited	Which50 Customer Communication Platform	Customer Communication Platform - Email, SMS, QR Codes, Web Forms, Landing Pages, eDocs	Price varies	Subscribe	Heart icon
Citrix	Citrix Connector Appliance for Cloud Services	Citrix Cloud Connector Appliance	Price varies	Create	Heart icon
Informatica	Informatica Intelligent Cloud Services-BYOL	Azure Application	Price varies	Create	Heart icon

Figure 32: Creating Email Communication Service

We will fill the necessary fields then create the service

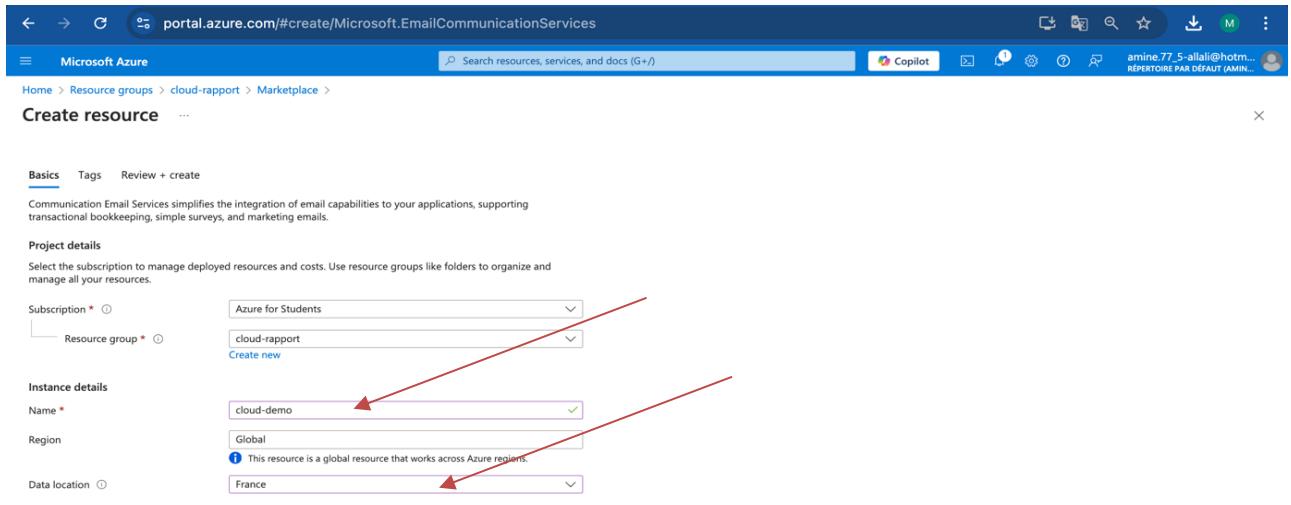


Figure 33: Creating Email Communication Service

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

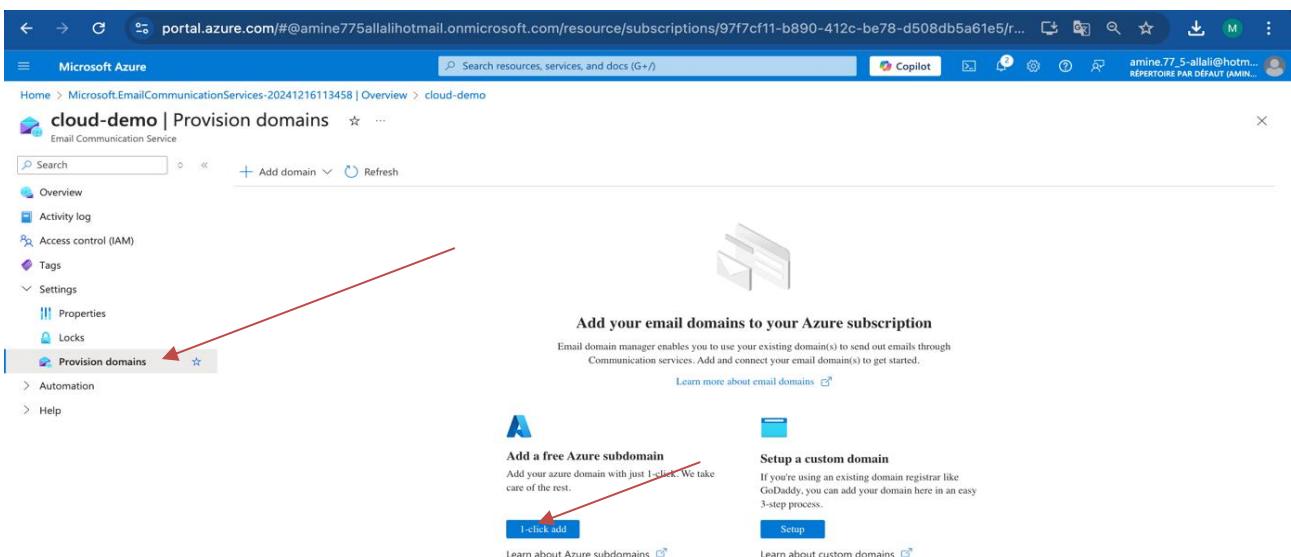


Figure 34: Creating Email Communication Service domain

After that we can see that the domain has been added

The screenshot shows the Microsoft Azure portal interface. The URL is portal.azure.com/#@amine775allalihotmail.onmicrosoft.com/resource/subscriptions/97f7cf11-b890-412c-be78-d508db5a61e5/r.... The page title is "Microsoft Azure" and the sub-page title is "cloud-demo | Provision domains". A green success message at the top says "Azure subdomain has been successfully deployed." Below it, there's a table with one item. The first column is "Domain name" with value "b5e998aa-4422-496e-993e-c219...". The second column is "Domain type" with value "Azure subdomain". The third column is "Domain status" with value "Verified". The fourth column is "SPF status" with value "Verified". The fifth column is "DKIM status" with value "Verified". The sixth column is "DKIM2 status" with value "Verified". A red arrow points to the "Domain name" field.

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

The screenshot shows the Microsoft Azure Marketplace. The URL is portal.azure.com/#view/Microsoft_Azure_Marketplace/MarketplaceOffersBlade/selectedMenuItemId/home/dontDiscardJourney-/.... The search bar shows "communication services". The results list includes "Communication Services" by Microsoft, "Email Communication Services" by Microsoft, "Infobip Communication Services" by Infobip d.o.o., "Which50 Customer Communication Platform" by CustomerMinds Limited, "Citrix Connector Appliance for Cloud Services" by Citrix Software Group, and "Informatica Intelligent Cloud Services-BYOL" by Informatica. A red arrow points to the "Create" button under the "Communication Services" listing.

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

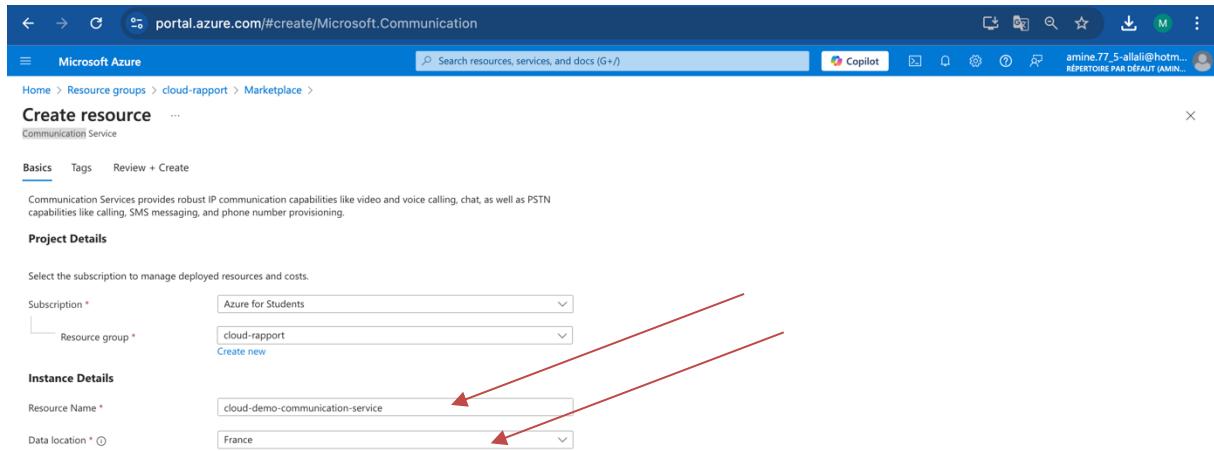


Figure 37: Creating a communication service

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

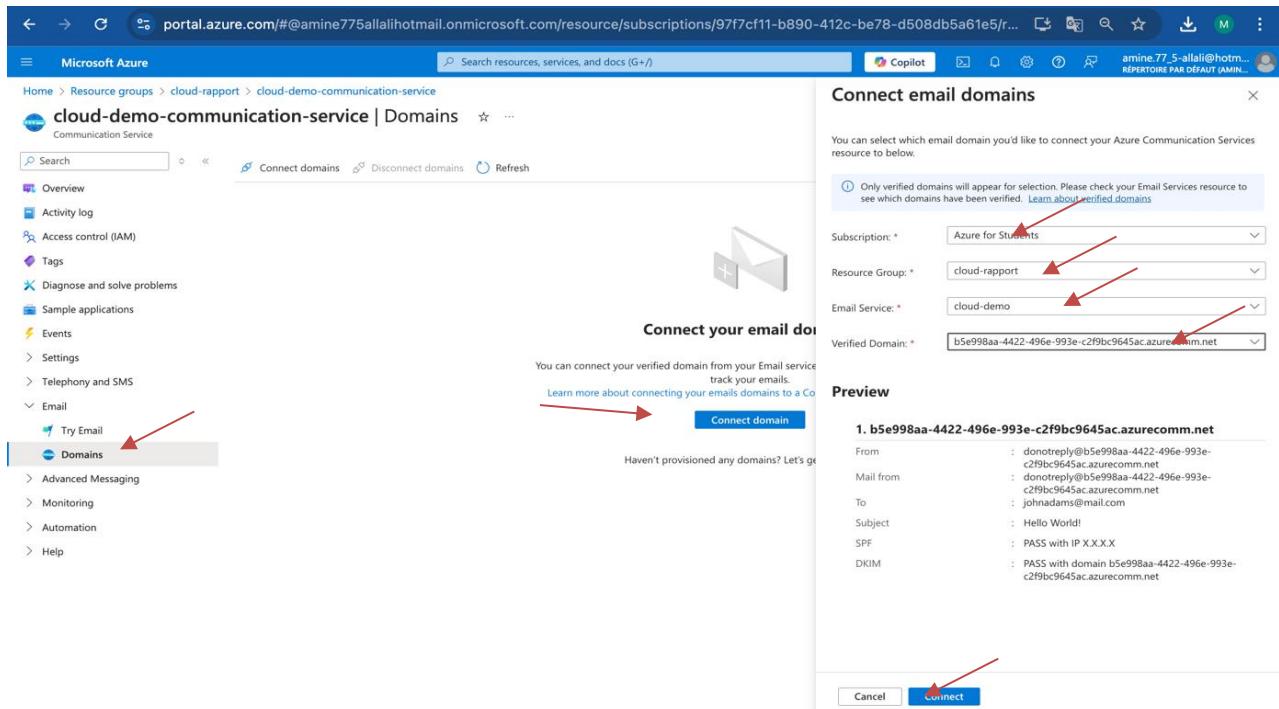


Figure 38: Connect our domain to communication service

The screenshot shows the Microsoft Azure portal interface. The URL is portal.azure.com/#@amine775allalihotmail.onmicrosoft.com/resource/subscriptions/97f7cf11-b890-412c-be78-d508db5a61e5/r.... The page title is "Microsoft Azure". The main content area is titled "cloud-demo-communication-service | Domains". It shows a table with one row: "Domain name" is "b5e998aa-4422-496e-993e-c2f9bc9645ac.azurecomm.net" and "Status" is "Connected". A red arrow points to the "cloud-demo" entry in the table.

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.

The screenshot shows the Microsoft Azure portal interface. The URL is portal.azure.com/#@amine775allalihotmail.onmicrosoft.com/resource/subscriptions/97f7cf11-b890-412c-be78-d508db5a61e5/r.... The page title is "Microsoft Azure". The main content area is titled "cloud-demo-communication-service | Try Email". On the left sidebar, under the "Email" section, the "Try Email" option is highlighted with a red arrow. The main form has fields for "Send email from" (selected as "b5e998aa-4422-496e-993e-c2f9bc9645ac.azurecomm.net"), "Sender email username" ("DoNotReply"), "Recipient email address" ("anirlinux111@gmail.com"), "Subject" ("Test Email"), and "Body" ("Hello world via email."). Below the body field, a message says "Email sent successfully!" next to a blue "Send" button. To the right, there is a code editor window with C# code for sending an email using the Azure Communication Library.

```

C# JavaScript Java Python cURL
Copy Insert my connection string
1  from azure.communication.email import EmailClient
2
3  def main():
4      try:
5          connection_string = "null"
6          client = EmailClient.from_connection_string(connection_string)
7
8          message = (
9              "senderAddress": "DoNotReply@b5e998aa-4422-496e-993e-c2f9bc9645ac.
azurecomm.net",
10             "recipients": [
11                 {"to": [{"address": "anir-linux111@gmail.com"}]
12             },
13             "content": {
14                 "subject": "Test Email",
15                 "plainText": "Hello world via email.",
16                 "html": """
17                     <html>
18                         <body>
19                             Hello world via email.
20                         </body>
21                     </html>
22                 
```

Figure 40: testing email sending

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

The screenshot shows a Gmail inbox. The top bar includes standard navigation icons. The main area shows an email from "DoNotReply <DoNotReply@b5e998aa-4422-496e-993e-c2f9bc9645ac.fr1.azurecomm.net>" to the user, received at 12:19 PM (3 minutes ago). The subject of the email is "Hello world via email.". At the bottom of the email preview, there are three buttons: "Hey what's up?", "It works!", and "Got it!".

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)



```
1  def send_email(email, message, sender):
2      try:
3          connection_string = conn_string_email_sender
4          client = EmailClient.from_connection_string(connection_string)
5
6          message = {
7              "senderAddress": f"{senderADD}",
8              "recipients": {
9                  "to": [{"address": f"{email}"}]
10             },
11             "content": {
12                 "subject": "SPAM NOTIFICATION",
13                 "plainText": "Hello world via email.",
14                 "html": f"""
15                     <html>
16                         <body>
17                             <h3>Hello a <a style="color:red;">SPAM</a> was detected !!
18                             <h4>It was sent by :{sender}</h4>
19                             <h4>The Message :</h4>
20                             <p>{message}</p>
21                         </body>
22                     </html>"""
23             },
24         },
25     }
26
27     poller = client.begin_send(message)
28     result = poller.result()
29     print("Message sent")
30
31 except Exception as ex:
32     print(ex)
33
```

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

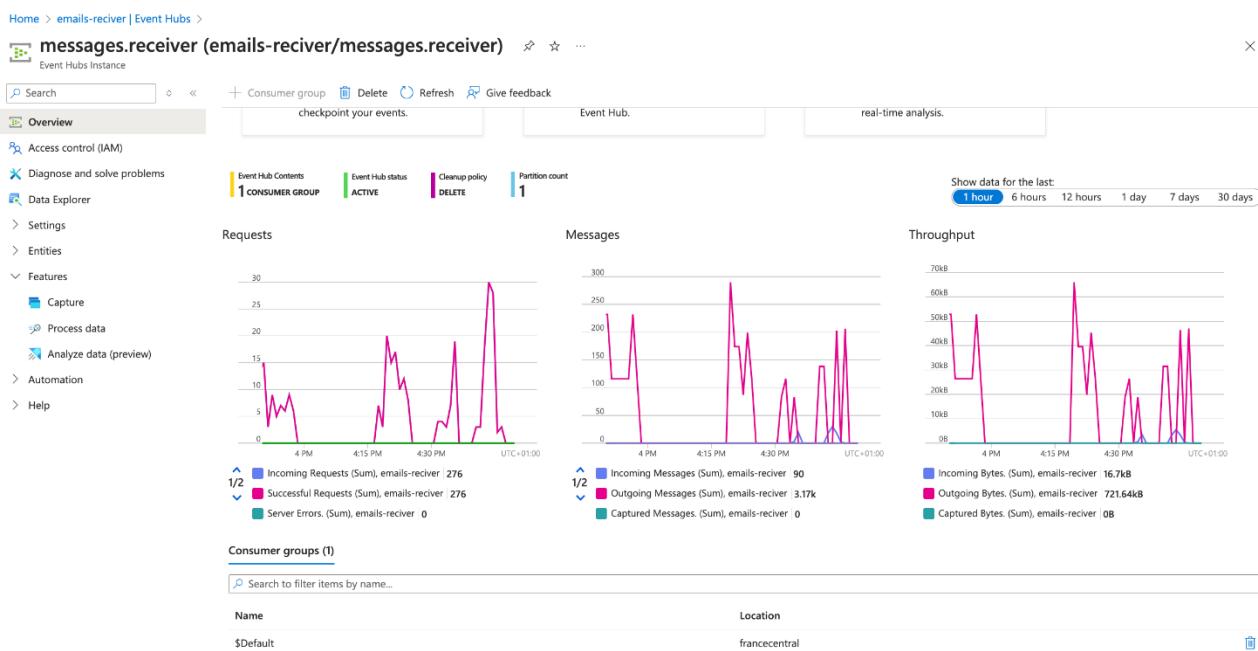


Figure 42 : Monitoring using Azure event hubs

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:

The screenshot shows the Azure portal interface for managing a 'Web App'. The left sidebar lists various settings like Overview, Activity log, and Deployment. The main content area displays the 'myFlaskApp' configuration under the 'Essentials' tab. It includes details such as Resource group, Status, Location, Subscription, Tags, and Properties. The 'Properties' tab is selected, showing the 'Web app' and 'Domains' sections. In the 'Web app' section, there's a 'Deployment Center' panel with deployment logs, last deployment, and provider information. A red box highlights the 'Application Insights' section, which shows a status message: 'Not supported. Learn more'.

Figure 43: Enable the App insights

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

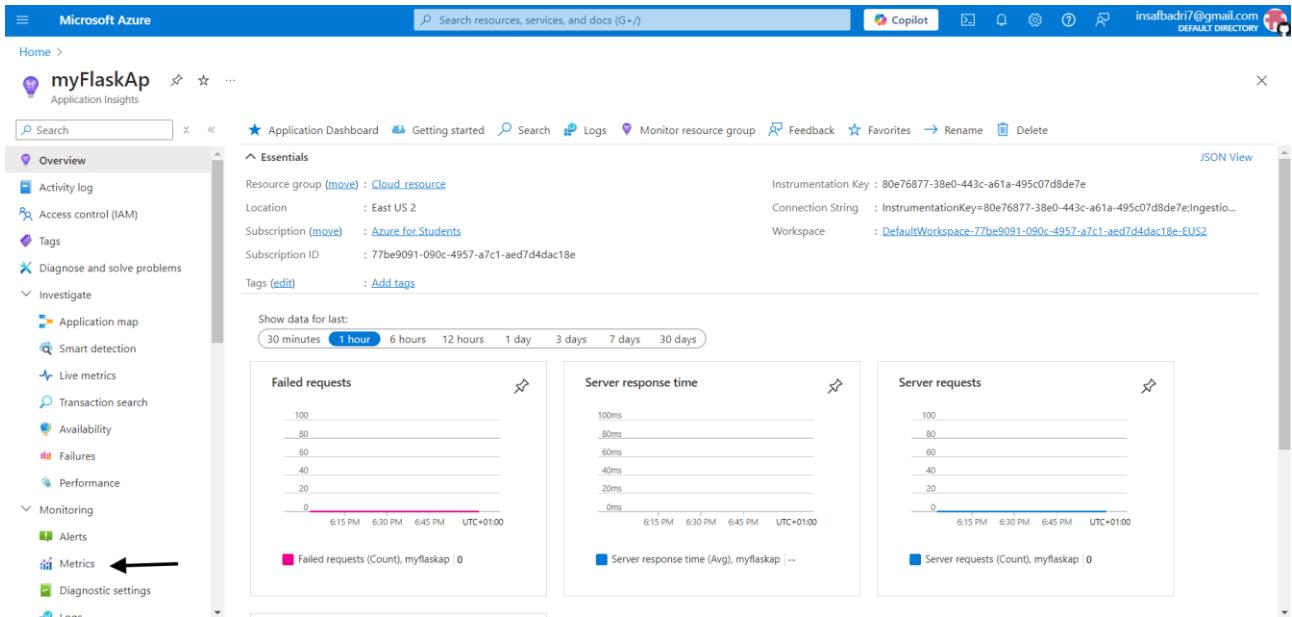


Figure 44: Monitoring using App Insights

3. App services:

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

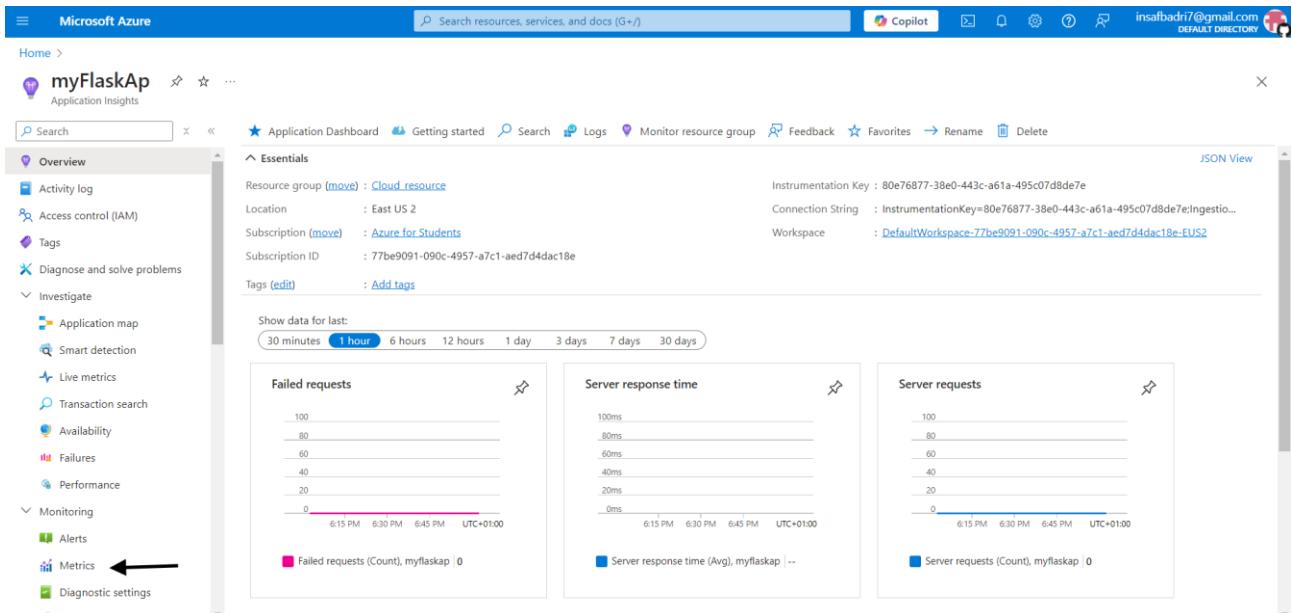


Figure 45: App Service Monitoring

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.

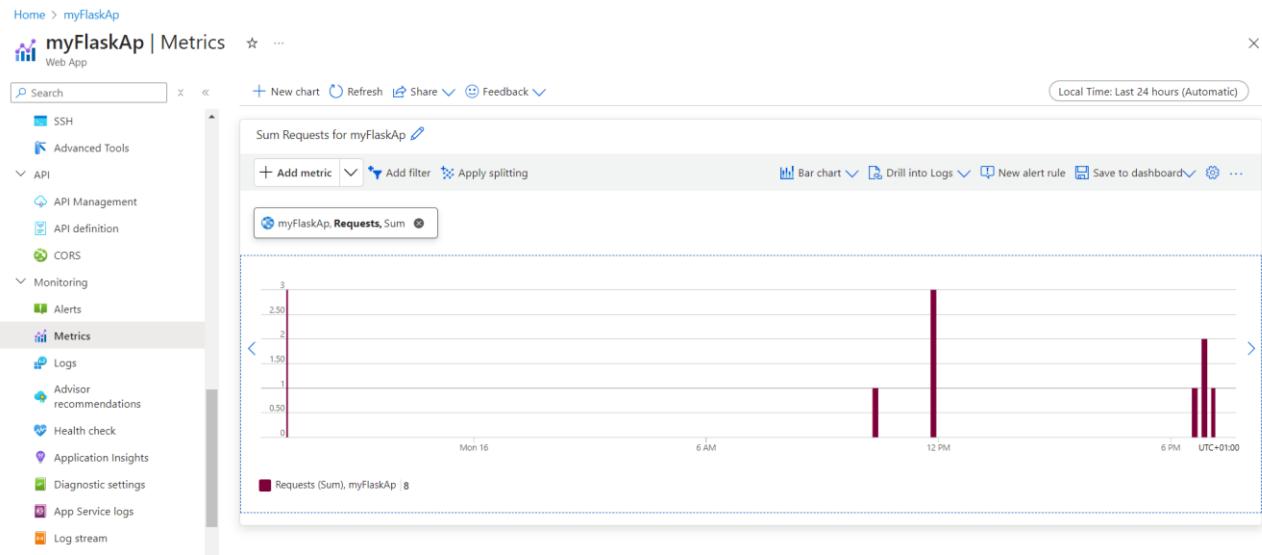


Figure 46: Request metrics

Here we have the Data in:

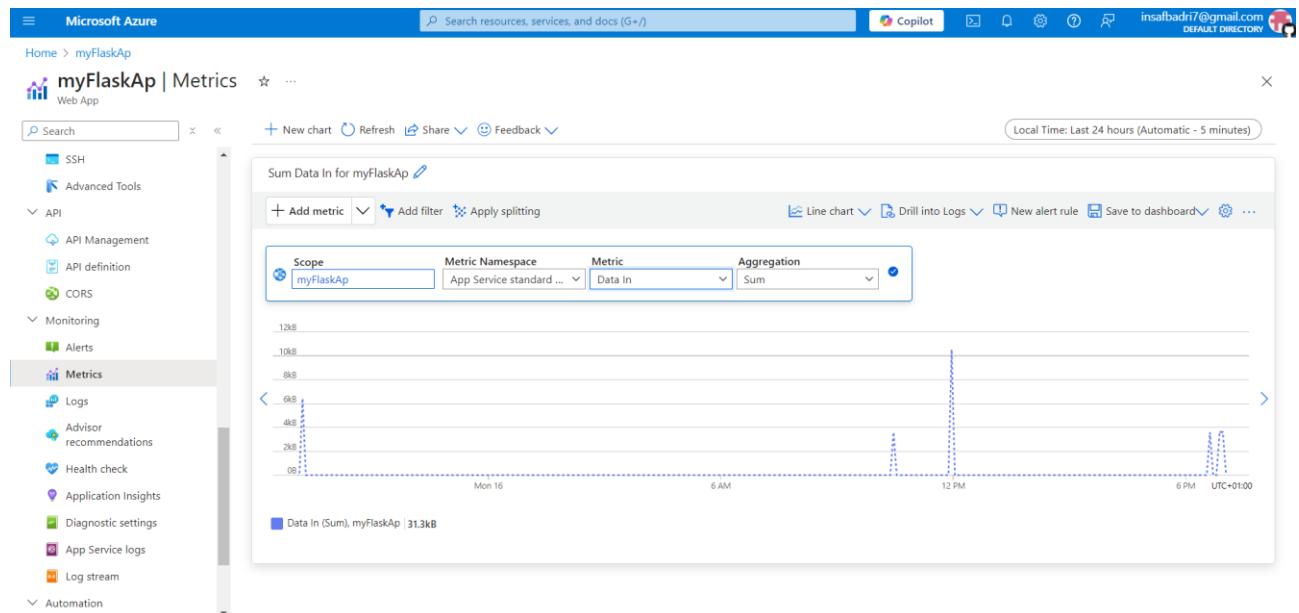


Figure 47: Data in Chart