

# Configuration Manual

MSc Research Project  
Master of Science in Cybersecurity

Dinal Varma  
Student ID: 23241021

School of Computing  
National College of Ireland

Supervisor: Dr. Evgeniia Jayasekera

National College of Ireland  
Project Submission Sheet  
School of Computing



<b>Student Name:</b>	Dinal Varma
<b>Student ID:</b>	23241021
<b>Programme:</b>	Master of Science in Cybersecurity
<b>Year:</b>	2024
<b>Module:</b>	MSc Research Project
<b>Supervisor:</b>	Dr. Evgeniia Jayasekera
<b>Submission Due Date:</b>	12/12/2024
<b>Project Title:</b>	Configuration Manual
<b>Word Count:</b>	1483
<b>Page Count:</b>	23

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

**ALL** internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

<b>Signature:</b>	Dinal Sunil Varma
<b>Date:</b>	10th December 2024

**PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:**

Attach a completed copy of this sheet to each project (including multiple copies).	<input type="checkbox"/>
<b>Attach a Moodle submission receipt of the online project submission</b> , to each project (including multiple copies).	<input type="checkbox"/>
<b>You must ensure that you retain a HARD COPY of the project</b> , both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	<input type="checkbox"/>

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

<b>Office Use Only</b>	
Signature:	
Date:	
Penalty Applied (if applicable):	

# Configuration Manual

Dinal Varma  
23241021

## 1 Introduction

This configuration manual serves as a guide to assist in setting up the project and reproducing the outcomes. This document contains references to all of the code libraries, hardware specs, and software specifications used in the artefact's implementation. This configuration manual is a key component of the research and the proposed concept along with the implementation and the setup on the cloud.

## 2 Hardware and Software Specifications

### 2.1 Hardware Specifications

- Operating system: Windows 11 Home
- Processor: Intel i5-12500H CPU @ 2.50GHz
- System Type: 64-bit operating system, x64-based processor
- Hard Disk: 512GB SSD
- Installed physical memory RAM: 16GB

### 2.2 Software Specifications

- Visual Studio Code (IDE)
- Python (version = 3.11.5)
- Jupyter Notebook (version = 7.3.0)

## 3 Dataset Information and Collection

The dataset used in this research is public repository - Cybersecurity: Suspicious Web Threat Interactions JanCSG (2024). The data can be directly downloaded from the link provided and to be stored in the appropriate folder where the files reside. Web traffic records gathered by AWS CloudWatch are included in this dataset in order to identify suspicious activity and possible attack attempts. In order to find unusual patterns, the data was created by tracking traffic to a production web server using a variety of detection methods.

## 4 Python Libraries Used

The following python libraries were used overall to develop and train the machine learning model along with the web application and the log consumer script:

Library	Version
pandas	2.2.3
numpy	2.0.2
seaborn	0.13.2
matplotlib	3.9.3
networkx	3.4.2
joblib	1.4.2
scikit-learn	1.5.2
tensorflow	2.18.0
keras	3.7.0
flask	3.1.0
geoip2	4.8.1
python-dotenv	1.0.1
boto3	1.35.76

Table 1: Python Libraries

Optional: To install all the above requirements in one go, please run the following command (ensure ‘requirements.txt’ file provided in the artefact is present in the same directory):

```
pip install -r requirements.txt
```

## 5 Data Preprocessing and Generation

Synthetic Data Generation was employed to simulate potential normal behaviours because the dataset only includes suspicious data. To address this issue and to enhance the dataset synthetic normal traffic data was generated to balance the dataset, simulating realistic web traffic patterns using the code below in Figure 1.

```
#Generate synthetic non-malicious data
num_entries = 10000
synthetic_data = pd.DataFrame({
    'creation_time': pd.date_range(start=data['creation_time'].min(), periods=num_entries, freq='10T'),
    'end_time': pd.date_range(start=data['creation_time'].min(), periods=num_entries, freq='10T') + pd.to_timedelta(np.random.randint(1, 60, size=num_entries), unit='T'),
    'bytes_in': np.random.choice([np.random.normal(1e4, 2e3), np.random.lognormal(9, 1), np.random.uniform(1e3, 3e4)], num_entries),
    'bytes_out': np.random.choice([np.random.normal(1e4, 2e3), np.random.lognormal(9, 1), np.random.uniform(1e3, 3e4)], num_entries),
    'src_ip': np.random.choice(data['src_ip'].tolist(), num_entries),
    'src_ip_country_code': np.random.choice(['US', 'GB', 'CA', 'AU', 'IN', 'CN', 'NL', 'AE', 'RU'], num_entries, p=[0.3, 0.1, 0.1, 0.05, 0.1, 0.05, 0.05, 0.05, 0.2]),
    'protocol': 'HTTPS',
    'response_code': np.random.choice([200, 200, 200, 200, 404, 500, 301, 302], num_entries),
    'dst_port': 443,
    'dst_ip': np.random.choice(data['dst_ip'].unique(), num_entries),
    'rule_names': np.random.choice(['No Alert', 'User Authentication', 'Data Query', 'Normal Web Traffic'], num_entries),
    'observation_name': np.random.choice(['User Activity', 'System Maintenance', 'User Login', 'Normal Operation'], num_entries),
    'source_meta': np.random.choice(['AWS_VPC_Flow', 'Internal_System_Log'], num_entries),
    'source_name': np.random.choice(['prod_webserver', 'dev_webserver'], num_entries),
    'time': pd.date_range(start='2024-04-20', periods=num_entries, freq='T'),
    'detection_types': np.random.choice(['normal_activity', 'user_login', 'data_access', 'waf_rule', 'no_detection'], num_entries),
    'label': 0 # Label for non-malicious data
})

# Calculate duration for synthetic data
synthetic_data['duration'] = (synthetic_data['end_time'] - synthetic_data['creation_time']).dt.total_seconds()

# Combine the original and synthetic data
combined_data = pd.concat([data, synthetic_data]).reset_index(drop=True)

# Apply label noise: Flip 20% of labels to simulate real-world mislabeling
indices = combined_data.index.tolist()
random.shuffle(indices)
num_noisy_labels = int(0.2 * len(indices)) # 20% label noise
noisy_indices = indices[:num_noisy_labels]
for idx in noisy_indices:
    combined_data.at[idx, 'label'] = 1 - combined_data.at[idx, 'label'] # Flip the label

# Increase overlap in bytes_in and bytes_out features by modifying distributions
non_malicious_indices = combined_data[combined_data['label'] == 0].index
malicious_sample_indices = np.random.choice(combined_data[combined_data['label'] == 1].index, size=len(non_malicious_indices), replace=True)
combined_data.loc[non_malicious_indices, 'bytes_in'] = combined_data.loc[malicious_sample_indices, 'bytes_in'].values * np.random.uniform(0.8, 1.2, size=len(non_malicious_indices))
combined_data.loc[non_malicious_indices, 'bytes_out'] = combined_data.loc[malicious_sample_indices, 'bytes_out'].values * np.random.uniform(0.8, 1.2, size=len(non_malicious_indices))

# Introduce additional noise in numerical features
combined_data['bytes_in'] = combined_data['bytes_in'] * np.random.uniform(0.5, 1.5, size=len(combined_data))
combined_data['bytes_out'] = combined_data['bytes_out'] * np.random.uniform(0.5, 1.5, size=len(combined_data))
combined_data.to_csv('./final_new_dataset.csv') # saving the final merged dataset
```

Figure 1: Synthetic Data Generation

The data preprocessing involved removing the duplicates and the addition of a new feature column called 'duration\_seconds' as shown in Figure 2 below:

```
1 df_unique = data.drop_duplicates()
2 df_unique['creation_time'] = pd.to_datetime(df_unique['creation_time'])
3 df_unique['end_time'] = pd.to_datetime(df_unique['end_time'])
4 df_unique['src_ip_country_code'] = df_unique['src_ip_country_code'].str.upper()
5 print("Unique Datasets Information:")
6 df_unique.info()

Unique Datasets Information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10282 entries, 0 to 10281
Data columns (total 19 columns):
#   Column              Non-Null Count  Dtype
---  -
0   Unnamed: 0           10282 non-null  int64
1   bytes_in             10282 non-null  float64
2   bytes_out            10282 non-null  float64
3   creation_time        10282 non-null  datetime64[ns]
4   end_time             10282 non-null  datetime64[ns]
5   src_ip              10282 non-null  object
6   src_ip_country_code  10282 non-null  object
7   protocol             10282 non-null  object
8   response.code        10282 non-null  int64
9   dst_port             10282 non-null  int64
10  dst_ip              10282 non-null  object
11  rule_names          10282 non-null  object
12  observation_name     10282 non-null  object
13  source.meta         10282 non-null  object
14  source.name         10282 non-null  object
15  time               10282 non-null  object
16  detection_types     10282 non-null  object
17  duration            10282 non-null  float64
18  label              10282 non-null  int64
dtypes: datetime64[ns](2), float64(3), int64(4), object(10)
memory usage: 1.5+ MB

1 # Feature engineering: Calculate duration of connection
2 df_unique['duration_seconds'] = (df_unique['end_time'] - df_unique['creation_time']).dt.total_seconds()
```

Figure 2: Data Preprocessing

```
1 # Preparing column transformations - using StandardScaler for numerical features
2 scaler = StandardScaler()
3 scaled_features = scaler.fit_transform(df_unique[['bytes_in', 'bytes_out', 'duration_seconds']])
4
5 # OneHotEncoder for categorical
6 encoder = OneHotEncoder(sparse_output=False)
7 enc_features = encoder.fit_transform(df_unique[['src_ip_country_code']])
8
9 # Combining transformed features back into the DataFrame
10 scaled_columns = ['scaled_bytes_in', 'scaled_bytes_out', 'scaled_duration_seconds']
11 enc_columns = encoder.get_feature_names_out(['src_ip_country_code'])
12
13 scaled_df = pd.DataFrame(scaled_features, columns=scaled_columns, index=df_unique.index)
14 encoded_df = pd.DataFrame(enc_features, columns=enc_columns, index=df_unique.index)
15
16 transformed_df = pd.concat([df_unique, scaled_df, encoded_df], axis=1)
17 transformed_df.head()
```

Figure 3: Data Preprocessing - Encoding

## 6 Exploratory Data Analysis

The following code demonstrate the Exploratory Data Analysis Process that was followed to gain initial insights about the dataset and to check the distribution, correlation of the data at hand and proceed accordingly:

```
1 corr_df = transformed_df.select_dtypes(include=['float64', 'int64'])
2 corr_mat = corr_df.corr()
3 plt.figure(figsize=(10, 8))
4 sns.heatmap(corr_mat, annot=True, fmt=".2f", cmap='coolwarm')
5 plt.title('Correlation Matrix Heatmap')
6 plt.show()
```

Figure 4: Correlation Matrix Code

```

1 plt.figure(figsize=(8, 8))
2 data['src_ip_country_code'].value_counts().plot(kind='pie', autopct='%1.1f%%', startangle=140, cmap='tab10')
3 plt.title('Source IP Country Codes Distribution')
4 plt.ylabel('')
5 plt.show()

```

Figure 5: Pie Chart Generation

```

1 # Convert 'creation_time' to datetime format
2 data['creation_time'] = pd.to_datetime(data['creation_time'])
3
4 data.set_index('creation_time', inplace=True)
5
6 plt.figure(figsize=(11, 7))
7 plt.plot(data.index, data['bytes_in'], label='Bytes In', marker='o')
8 plt.plot(data.index, data['bytes_out'], label='Bytes Out', marker='o')
9 plt.title('Web Traffic Analysis Over Time')
10 plt.xlabel('Time')
11 plt.ylabel('Bytes')
12 plt.legend()
13 plt.grid(True)
14 plt.xticks(rotation=45)
15 plt.tight_layout()
16 plt.show()

```

Figure 6: Web Traffic Analysis Over Time Code

```

1 G = nx.Graph()
2
3 for idx, row in data.iterrows():
4     G.add_edge(row['src_ip'], row['dst_ip'])
5
6 plt.figure(figsize=(14, 10))
7 nx.draw_networkx(G, with_labels=True, node_size=20, font_size=8, node_color='skyblue', font_color='darkblue')
8 plt.title('Network Interaction between Source and Destination IPs')
9 plt.axis('off')
10 plt.show()

```

Figure 7: Network Graph Code

```

1 plt.figure(figsize=(11, 7))
2 sns.histplot(data['duration'], kde=True, color='blue')
3 plt.title('Duration distribution')
4 plt.xlabel('Duration (seconds)')
5 plt.ylabel('Frequency')
6 plt.show()

```

Figure 8: Histogram of Duration Distribution

## 7 Machine Learning Models

The following Machine Learning Models were used to test and evaluate their performance on the dataset.

## 7.1 Isolation Forest

```
1 # Assuming ~10% contamination based on common anomaly threshold
2 iso_forest = IsolationForest(contamination=0.2, random_state=42)
3 iso_forest.fit(X)
✓ 0.1s
```

IsolationForest

IsolationForest(contamination=0.2, random\_state=42)

Figure 9: Isolation Forest Model

## 7.2 Random Forest

```
1 rf = RandomForestClassifier(n_estimators=50, random_state=42)
2 rf.fit(X, y)
✓ 0.4s
```

RandomForestClassifier

RandomForestClassifier(n\_estimators=50, random\_state=42)

Figure 10: Random Forest Model

## 7.3 Support Vector Classifier (SVC)

```
1 model_svc = SVC(kernel='rbf', random_state=42)
2 model_svc.fit(X, y)
✓ 1.9s
```

SVC

SVC(random\_state=42)

Figure 11: Support Vector Classifier (SVC) Model

## 7.4 Gradient Boosting

```
1 grad_boost = GradientBoostingClassifier(random_state=42)
2 grad_boost.fit(X, y)
✓ 1.1s
```

GradientBoostingClassifier

GradientBoostingClassifier(random\_state=42)

Figure 12: Gradient Boosting Model

## 7.5 Neural Network

```
1 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
2
3 model = Sequential([
4     Dense(64, activation='relu', input_shape=(X_train.shape[1],)),
5     Dense(32, activation='relu'),
6     Dense(16, activation='relu'),
7     Dense(1, activation='sigmoid') # Output layer for binary classification
8 ])
9
10 model.compile(optimizer=Adam(learning_rate=0.001),
11               loss=BinaryCrossentropy(),
12               metrics=[BinaryAccuracy()])
13
14 history = model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=100, batch_size=1024, verbose=1)
15
```

Figure 13: Basic Neural Network Model

```
1 tuned_model = Sequential([
2     Dense(128, input_shape=(X_train_prepared.shape[1],), activation='relu', kernel_regularizer=l2(0.01)),
3     BatchNormalization(),
4     Dropout(0.5),
5     Dense(64, activation='relu', kernel_regularizer=l2(0.01)),
6     BatchNormalization(),
7     Dropout(0.3),
8     Dense(32, activation='relu', kernel_regularizer=l2(0.01)),
9     Dropout(0.2),
10    Dense(1, activation='sigmoid')
11 ])
12
13 tuned_model.compile(optimizer=Adam(learning_rate=0.001), loss='binary_crossentropy', metrics=['accuracy'])
14
15 history = tuned_model.fit(X_train_prepared, y_train, epochs=50, validation_data=(X_val_prepared, y_val))
16
```

Figure 14: Fine Tuned Neural Network Model

## 8 Experimental Setup

This section gives the step by step for setting up and deploying the system on the AWS Cloud environment along with the services essential for running of the prediction and alerting script.

### 8.1 Web-Application

The web application was developed using Flask framework since it is a lightweight web framework which is easy to deploy and integrates easily with other web technologies. The one html page was developed as a prototype to simulate the web application and was served using Flask

The basic flask code for setting up 2 API Endpoints is defined in the Figure 15 below. It also includes setup for logging with 'traffic.log' file containing all of the application logs.



```

import os
import time
import logging
import datetime
from flask import Flask, request, render_template, jsonify
import geoip2.webservice
from dotenv import load_dotenv
load_dotenv()

app = Flask(__name__)

# Set up logging
logging.basicConfig(filename='./traffic.log', level=logging.INFO)

# GeoIP API credentials
GEOIP_ACCOUNT = os.getenv("GEOIP_ACCOUNT")
GEOIP_LICENSE_KEY = os.getenv("GEOIP_LICENSE_KEY")

@app.route('/')
def home():
    return render_template('index.html')

@app.route('/api', methods=['GET', 'POST'])
def backend_api():
    incoming_data = request.get_json()
    # simulating some processing of the incoming data
    time.sleep(5)
    return jsonify(incoming_data), 200

if __name__ == "__main__":
    app.run(debug=True)

```

Figure 15: Basic Flask Web Application Code

The Flask framework also provides with a inbuilt way to log each request through the ‘@app.before\_request’ and ‘@app.after\_request’ decorators as seen in the Figure 16 below:

```

@app.before_request
def start_timer():
    request.start_time = datetime.datetime.now()

@app.after_request
def log_request(response):
    try:
        # Ensure the response data is buffered and accessible
        if response.direct_passthrough:
            response.direct_passthrough = False

        # Capture request and response details
        bytes_in = request.headers.get('Content-Length') or 0
        bytes_out = len(response.get_data(as_text=True)) # Access data safely
        src_ip = request.remote_addr
        protocol = request.scheme
        response_code = response.status_code
        dst_port = 443 if protocol == 'HTTPS' else 80
        end_time = datetime.datetime.now()
        duration = (end_time - request.start_time).total_seconds()

        try:
            # Map src_ip to country code using GeoIP
            with geoip2.webservice.Client(GEOIP_ACCOUNT, GEOIP_LICENSE_KEY, host='geolite.info') as client:
                geo_response = client.country(src_ip)
                src_ip_country_code = geo_response.country.iso_code
        except geoip2.errors.AddressNotFoundError:
            src_ip_country_code = 'Unknown'

        # Log entry format
        log_entry = (f"bytes_in={bytes_in}, bytes_out={bytes_out}, src_ip={src_ip}, src_ip_country_code={src_ip_country_code}, "
                    f"protocol={protocol}, response.code={response_code}, dst_port={dst_port}, duration={duration}")
        app.logger.info(log_entry)
    except Exception as e:
        app.logger.error(f"Failed to log request: {e}")
    return response

```

Figure 16: Flask Request Logging

## 8.2 GeoIP Service

For getting the country code from the incoming IP Address, a service called ‘GeoIP’ MaxMind (no date) was utilized. Below show the steps to get the ‘Account ID’ and ‘LICENSE\_KEY’:

1. Go to Maxmind website and create a free account
2. On the account dashboard, go to ‘Manage License Keys’ section and click on ‘Generate New license key’ to generate and get the license key.
3. Note down the ‘Account ID’ and ‘LICENSE\_KEY’ from this page and replace the values in the flask code as seen in the figure 15 above.

## 8.3 Prediction and Alerting Script

This script loads the final model and model preprocessor pipeline from the AWS S3 Bucket and constantly reads the AWS Cloudwatch logs from the log group specified to run the models on the incoming data to detect if the incoming logs are anomalous or not and if the model predicts them to be anomalous, then it sends out an email to the concerned stakeholders using the ‘send\_email’ function that utilizes the Mailgun Email Service to send emails.

```
# Initialize a session using Amazon CloudWatch
session = boto3.Session(region_name=os.getenv("AWS_REGION_NAME"),
                        aws_access_key_id=os.getenv("AWS_ACCESS_KEY_ID"),
                        aws_secret_access_key=os.getenv("AWS_ACCESS_SECRET_KEY"))

log_group_name = 'MyAppTrafficLogs' # AWS Cloudwatch the log group name
logs_client = session.client('logs')

downloads = {
    "model":{
        "url":"https://zerodaymlmodel.s3.ap-south-1.amazonaws.com/final_model.keras",
        "path":"./final_model.keras"
    },
    "preprocessor":{
        "url":"https://zerodaymlmodel.s3.ap-south-1.amazonaws.com/preprocessor.joblib",
        "path":"./preprocessor.joblib"
    }
}

def send_email(confidence, timestamp, df):
    MAILGUN_API_KEY = os.getenv("MAILGUN_API_KEY")
    MAILGUN_API_DOMAIN = os.getenv("MAILGUN_API_DOMAIN")
    recipient_email = os.getenv("RECIPIENT_EMAIL")

    confidence = int(round(confidence, 2) * 100)
    timestamp = datetime.fromtimestamp(timestamp/1000).strftime('%Y-%m-%d %H:%M:%S')
    detail_dict = {
        "Source Ip": df['src_ip'].item(),
        "Country Code of incoming request": df['src_ip_country_code'].item(),
        "Incoming Bytes": df['bytes_in'].item(),
        "Outgoing Bytes": df['bytes_out'].item(),
        "Duration of the connection/request": f"{df['duration'].item()} seconds",
        "Response Code": df['response.code'].item(),
        "Protocol": df['protocol'].item(),
    }
    details = "\n".join([f"{key}: {value}" for key, value in detail_dict.items()])
    message = (
        f"There has been a suspicious activity on the website with a confidence score of {confidence}%!\n\n"
        f>Please log in to the website and check the logs of the incoming request at time {timestamp}.\n\n"
        f"Details of the activity:\n{details}"
    )
    res = requests.post(
        f"https://api.mailgun.net/v3/{MAILGUN_API_DOMAIN}/messages",
        auth=("api", MAILGUN_API_KEY),
        data={"from": "Alerts <mailgun@sandbox70c739d9ac8d434c85f6b911419b31b7.mailgun.org>",
              "to": [recipient_email],
              "subject": "Suspicious Activity Detected!",
              "text": message})
    res.raise_for_status()
    return res.status_code
```

Figure 17: Alerting Script

```

def get_latest_log_events(log_group_name):
    for name, item in downloads.items():
        url = item['url']
        path = item['path']
        if not url:
            print(f"URL for {name} is missing.")
            continue
        if not os.path.exists(path):
            print(f"Downloading {name} from {url} to {path} ...")
            response = requests.get(url)
            if response.status_code == 200:
                with open(path, 'wb') as f:
                    f.write(response.content)
                print(f"{name.capitalize()} downloaded successfully.")
            else:
                print(f"Failed to download {name}. Status code: {response.status_code}")

    # Load the model
    model = load_model(downloads['model']['path'])
    # Load the preprocessor
    preprocessor = joblib.load(downloads['preprocessor']['path'])

    processed_event_hashes = set() # Track processed events using a hash of timestamp and message

    while True:
        try:
            # Get the latest log stream
            response = logs_client.describe_log_streams(
                logGroupName=log_group_name,
                orderBy='LastEventTime',
                descending=True,
                limit=1
            )
            if not response['logStreams']:
                logging.info("No log streams found.")
                time.sleep(5)
                continue

            latest_log_stream = response['logStreams'][0]['logStreamName']

            # Get the last 5 log events from the latest log stream
            events_response = logs_client.get_log_events(
                logGroupName=log_group_name,
                logStreamName=latest_log_stream,
                limit=3,
                startFromHead=False # Get the latest events
            )

```

Figure 18: Alerting Script

```

events = events_response.get('events', [])
if events:
    for event in events:
        timestamp = event['timestamp']
        message = event['message']
        # Create a unique hash for the event
        event_hash = hashlib.sha256(f"{timestamp}-{message}".encode()).hexdigest()
        if event_hash in processed_event_hashes:
            # Skip already processed events
            continue

        # Add the hash to the processed set
        processed_event_hashes.add(event_hash)
        if 'bytes_in' not in message:
            continue

        cols = ['bytes_in', 'bytes_out', 'src_ip', 'src_ip_country_code', 'protocol', 'response.code', 'dst_port', 'duration']
        matches = re.findall(r"(\w+\.\w+|\w+)=([\w\.\.]+)", message)

        log_data = {key: value.strip(',') for key, value in matches}
        # Ensure all columns are in the dictionary (missing columns will default to None)
        log_data = {col: log_data.get(col, None) for col in cols}

        df = pd.DataFrame([log_data])
        numeric_cols = ['bytes_in', 'bytes_out', 'response.code', 'dst_port', 'duration']
        for col in numeric_cols:
            df[col] = pd.to_numeric(df[col], errors='coerce')

        # Transform the record and make predictions
        transformed_record = preprocessor.transform(df)
        prediction = model.predict(transformed_record, verbose=None)

        if prediction[0][0] >= 0.5:
            confidence = prediction[0][0]
            # Trigger alerts or take actions if suspicious
            logging.info(f"ALERT: Suspicious activity detected! Confidence: {confidence:.2f}\n\n")
            send_email(confidence, timestamp, df)
        else:
            confidence = 1 - prediction[0][0]
            logging.info(f"Normal activity. Confidence: {confidence:.2f}\n\n")

        # Sleep before the next fetch
        time.sleep(3)
except Exception as e:
    logging.error(f"An error occurred: {e}")
    time.sleep(10)

if __name__ == "__main__":
    get_latest_log_events(log_group_name)

```

Figure 19: Alerting Script

## 8.4 Mailgun Email Service

For sending the alerting emails in case of suspicious behaviour, Mailgun Mailgun (no date) was utilized which provides an easy way to send emails. The following steps allow to create an account and obtain the ‘API\_KEY’ and ‘DOMAIN’:

1. Create a free account on the mailgun website
2. On the dashboard page, click on ‘API Keys’ as shown below:

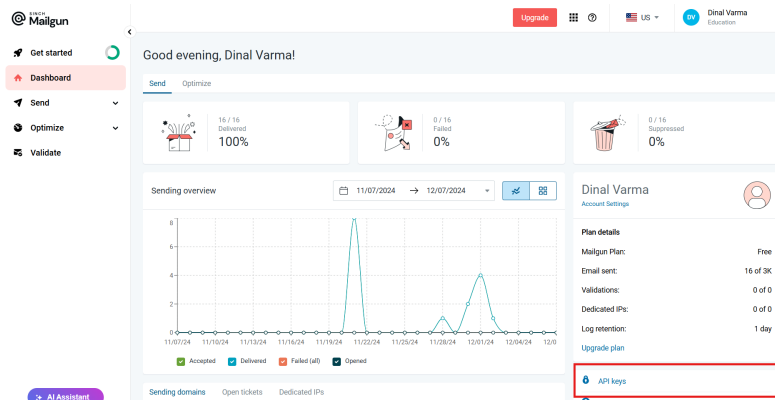


Figure 20: Mailgun Dashboard

3. Then click on ‘Add new Key’ button to generate API\_KEY and save it.
4. Next, on the dashboard home page, click on the ‘Sending’ submenu from the left sidebar and open ‘Domains’ tab. Here, a default **Sandbox** domain will be created.
5. Next, in order to send emails using this sandbox, the recipient needs to be verified:

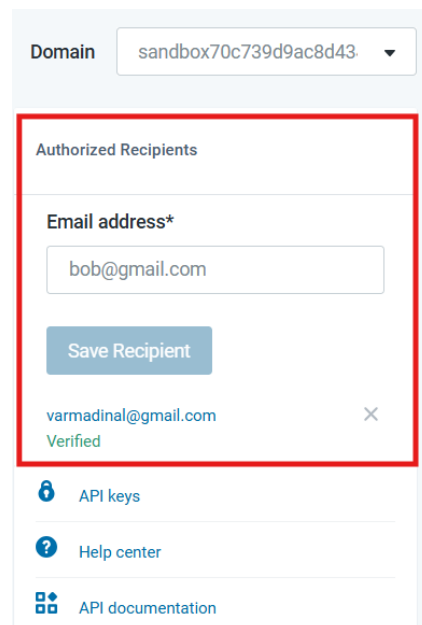
The image shows the 'Authorized Recipients' form in the Mailgun interface. At the top, the 'Domain' is set to 'sandbox70c739d9ac8d43'. The form has a section for adding new recipients with the label 'Email address\*' and a text input field containing 'bob@gmail.com'. Below the input field is a 'Save Recipient' button. Underneath, a list of existing recipients shows 'varmadinal@gmail.com' with a green 'Verified' status and a close icon. At the bottom of the form are links for 'API keys', 'Help center', and 'API documentation'.

Figure 21: Mailgun Email Verify

6. Replace the ‘API\_KEY’ and ‘DOMAIN’ obtained from above steps in the script to start sending/receiving emails.

## 8.5 Amazon Web Services (AWS) Setup and Deployment

For the system deployment, Amazon Web Service (AWS) Cloud was chosen to setup and deploy the web application along with the prediction and alerting script as well. Below steps show the process to setup and deploy the Flask based Web-application used in this research as mentioned above:

### AWS EC2 Instance Configuration:

1. Create an Free account on Amazon AWS Cloud AWS (no date)
2. Search for the EC2 service from the services tab and click on ‘Launch Instance Button’ to create and start an EC2 Instance.
3. On the screen, give the server a name and choose the ‘Amazon Linux 2023 AMI’ instance image which is free tier eligible:

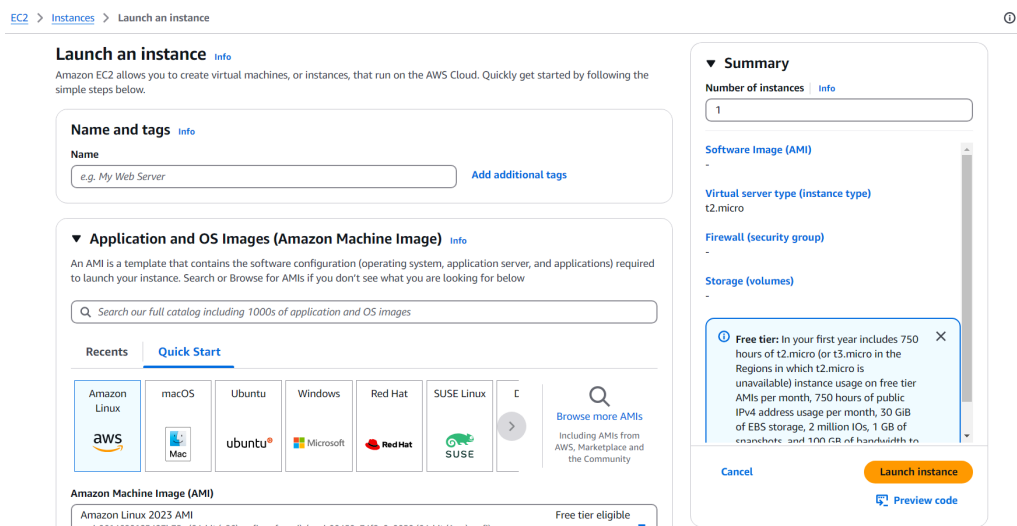


Figure 22: Launch an Amazon EC2 Instance

4. Next, choose ‘t2.micro’ as the instance type which is included in the free tier. And also, create a new Key pair of type RSA and ‘.pem’ type extension for connecting to the ec2 instance as shown below:

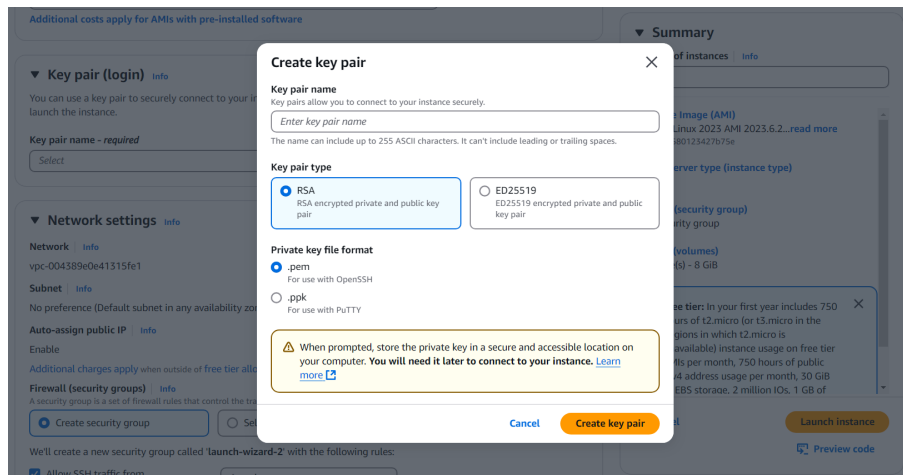


Figure 23: Create Key Pair to connect to EC2 Instance

- For the 'Network Settings', choose 'Create security Group' and enable SSH traffic from your IP Address (If known) or anywhere and also ensure you enable 'HTTP' and 'HTTPS' traffic to allow traffic to connect to the EC2 instance via web-application.

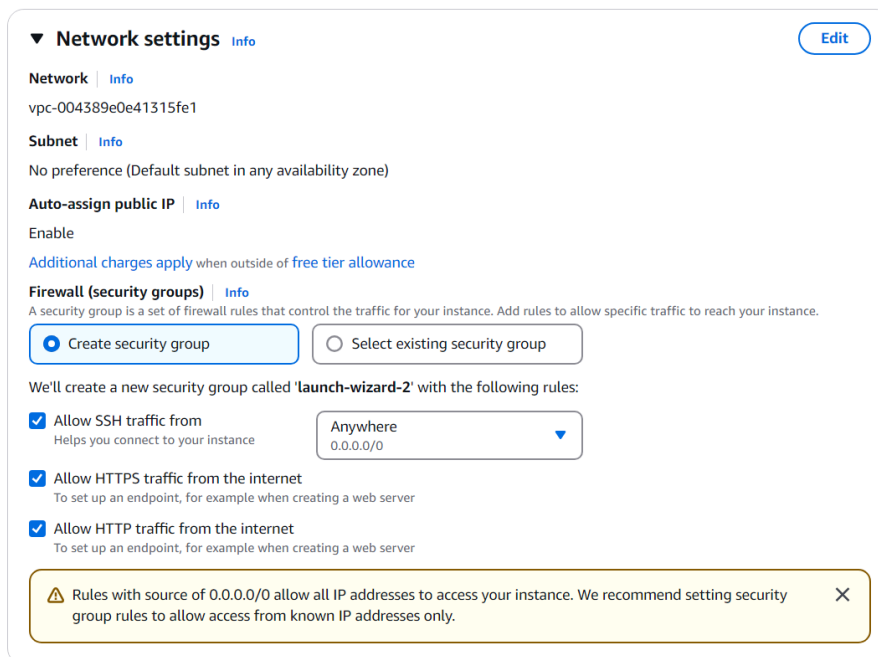


Figure 24: Network Settings for EC2

- Now click on 'Launch Instance' and the instance will now be ready and up and running to be connected.
- On newly created instance page, go to 'Actions' and select 'Security' and then click on 'Modify IAM Role'.
- Click on 'Create new IAM Role' Option as shown below:

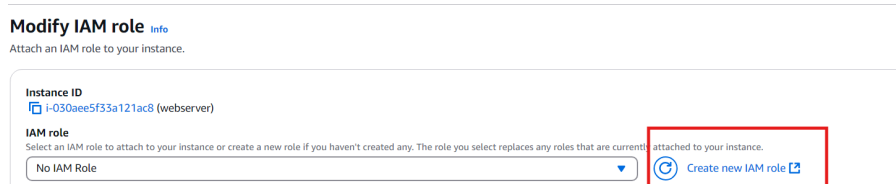


Figure 25: Modify IAM Role Screen

9. On the next screen, click on 'Create Role' option and assign the below trusted entity and use cases:

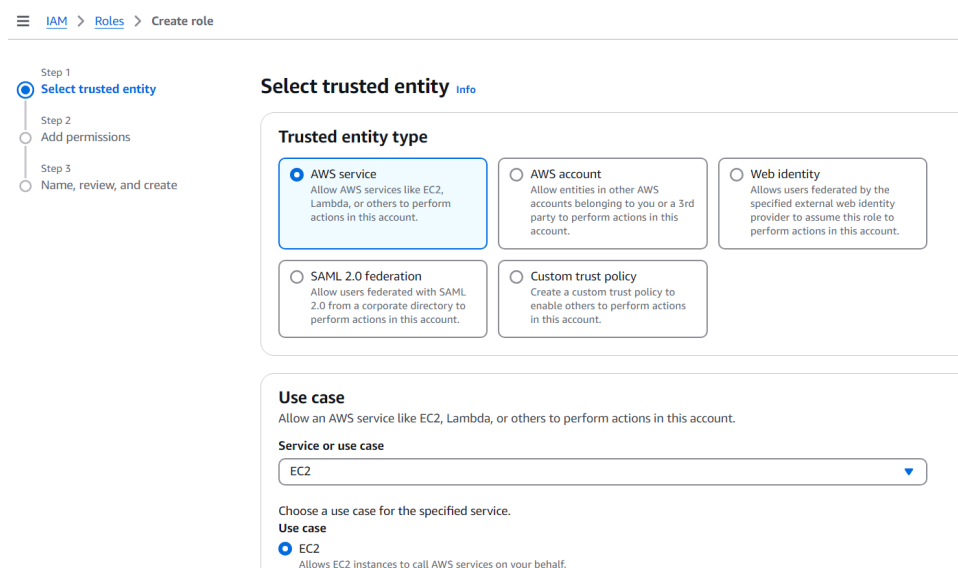


Figure 26: Create IAM Role

10. For the permissions on the next screen, search for 'Cloudwatch' in the search bar and select the below permission policy shown:

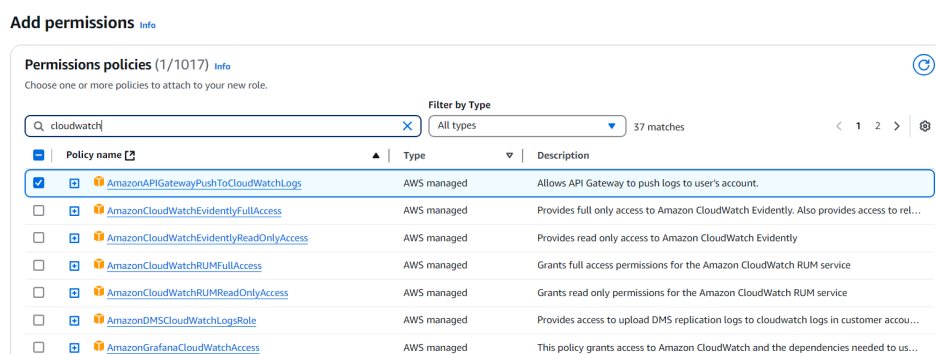


Figure 27: Assign Permission Policy to IAM Role

11. Review the details and provide a name for the Role created:

## Name, review, and create

**Role details**

**Role name**  
Enter a meaningful name to identify this role.  
  
Maximum 64 characters. Use alphanumeric and '+,=, @, -, \_' characters.

**Description**  
Add a short explanation for this role.  
  
Maximum 1000 characters. Use letters (A-Z and a-z), numbers (0-9), tabs, new lines, or any of the following characters: \_+ =, @, -, \_/[()!#\$%^&\*()~''

## Step 1: Select trusted entities

**Trust policy**

```
1 {  
2   "Version": "2012-10-17",  
3   "Statement": [  
4     {  
5       "Effect": "Allow",  
6       "Action": [  
7         "sts:AssumeRole"  
8       ],  
9       "Principal": {  
10        "Service": [  
11          "ec2.amazonaws.com"  
12        ]  
13      }  
14    }  
15  ]  
16 }
```

Figure 28: IAM Role Review

- Click on 'Create Role' to save the role and create the new Role attached to the EC2 instance created earlier.
- Verify on the EC2 instance settings that this newly created Role is now attached to the EC2 instance screen:

**Instance summary for i-030aee5f33a121ac8 (webserver)** Info

**Instance ID**  
i-030aee5f33a121ac8

**IPv6 address**  
-

**Hostname type**  
IP name: ip-172-31-37-10.ap-south-1.compute.internal

**Answer private resource DNS name**  
IPv4 (A)

**Auto-assigned IP address**  
-

**IAM Role**  
EC2\_ROLE

**IMDSv2**  
Required

**Operator**  
-

**Public IPv4 address**  
-

**Instance state**  
Stopped

**Private IP DNS name (IPv4 only)**  
ip-172-31-37-10.ap-south-1.compute.internal

**Instance type**  
t2.micro

**VPC ID**  
vpc-004389e0e41315fe1

**Subnet ID**  
subnet-093110986dcd80319

**Instance ARN**  
arn:aws:ec2:ap-south-1:212829096712:instance/i-030aee5f33a121ac8

**Private IPv4 addresses**  
172.31.37.10

**Public IPv4 DNS**  
-

**Elastic IP addresses**  
-

**AWS Compute Optimizer finding**  
Opt-in to AWS Compute Optimizer for recommendations. | Learn more

**Auto Scaling Group name**  
-

**Managed**  
false

**▼ Security details**

**IAM Role**  
EC2\_ROLE

**Security groups**  
sg-0346a2b0b6ec174b7 (launch-wizard-1)

**Owner ID**  
212829096712

**Launch time**  
Thu Dec 05 2024 14:28:03 GMT+0000 (Greenwich Mean Time)

Figure 29: EC2 Instance IAM Role Details

## Security Groups Configuration:

Go to the newly created EC2 instance and click on the security group assigned to it.

- Ensure the 'Inbound rules' match the below configuration:



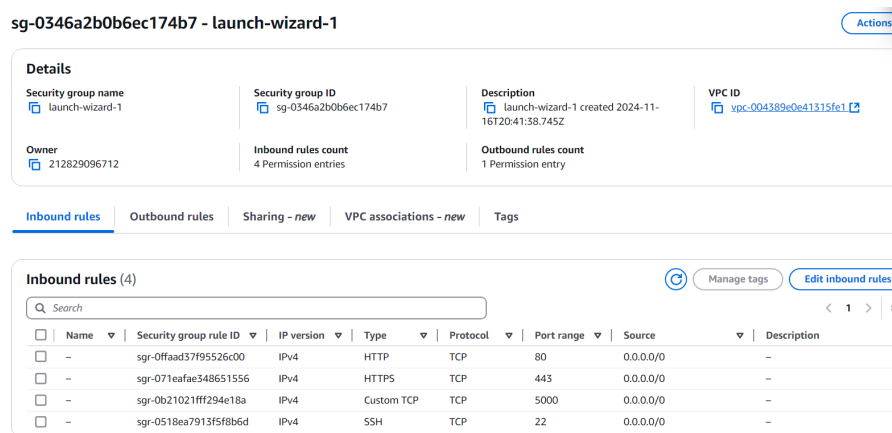


Figure 30: Inbound Rules for Security Groups

2. 'Outbound rules' configuration to allow server to send traffic to anywhere:

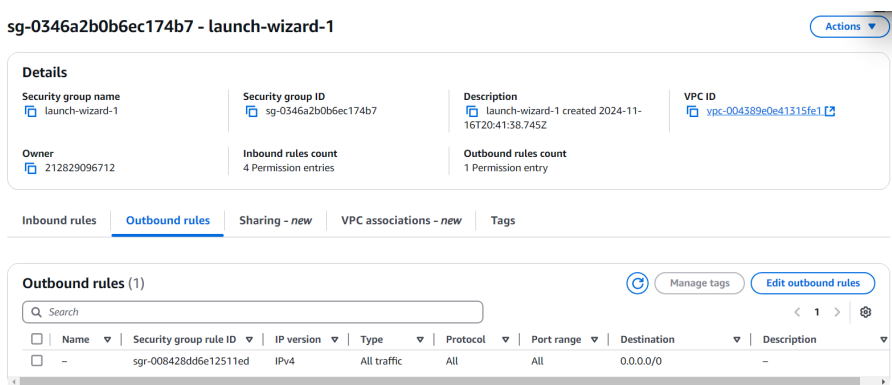


Figure 31: Outbound Rules for Security Groups

## Access Key ID AND Secret Configuration:

To obtain the 'Access Key ID' and 'Secret' for a user, a new user to be created using the steps below:

1. Go to 'IAM' service in the account and click on 'Users' on the left submenu.
2. Click on 'Create User' button and specify the User details like below:

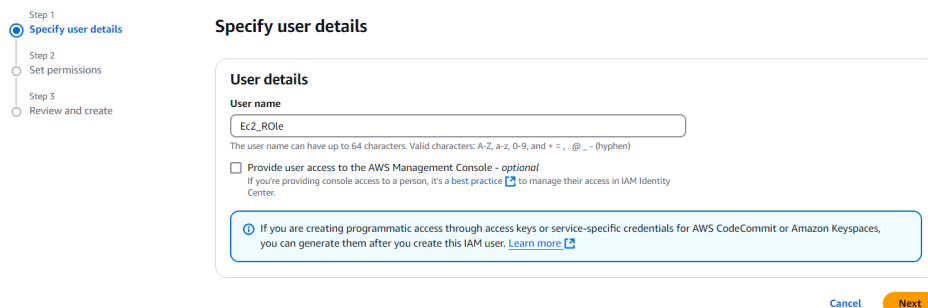


Figure 32: Create a IAM User

- For the permissions on the next screen, search for ‘Cloudwatch’ in the search bar and select the below permission policy shown:

**Set permissions**  
Add user to an existing group or create a new one. Using groups is a best-practice way to manage user's permissions by job functions. [Learn more](#)

**Permissions options**

☐ Add user to group  
Add user to an existing group, or create a new group. We recommend using groups to manage user permissions by job function.

☐ Copy permissions  
Copy all group memberships, attached managed policies, and inline policies from an existing user.

☒ Attach policies directly  
Attach a managed policy directly to a user. As a best practice, we recommend attaching policies to a group instead. Then, add the user to the appropriate group.

**Permissions policies (1/1314)** [Create policy](#)

Choose one or more policies to attach to your new user.

Filter by Type

cloudwatch 52 matches

Policy name	Type	Attached ent...
<input checked="" type="checkbox"/> <a href="#">AmazonAPIGatewayPushToCloudWatchLogs</a>	AWS managed	2
<input type="checkbox"/> <a href="#">AmazonCloudWatchEvidentlyFullAccess</a>	AWS managed	0
<input type="checkbox"/> <a href="#">AmazonCloudWatchEvidentlyReadOnlyAccess</a>	AWS managed	0

Figure 33: IAM User Permissions

- Go to this newly created user and go to ‘Security Credentials’ tab, scroll down to ‘Access keys’ and click on ‘Create access key’ and select the use case as shown below:

Step 1  
**Access key best practices & alternatives**

Step 2 - optional  
Set description tag

Step 3  
Retrieve access keys

**Access key best practices & alternatives** [Info](#)

Avoid using long-term credentials like access keys to improve your security. Consider the following use cases and alternatives.

**Use case**

☒ **Command Line Interface (CLI)**  
You plan to use this access key to enable the AWS CLI to access your AWS account.

☐ **Local code**  
You plan to use this access key to enable application code in a local development environment to access your AWS account.

☐ **Application running on an AWS compute service**  
You plan to use this access key to enable application code running on an AWS compute service like Amazon EC2, Amazon ECS, or AWS Lambda to access your AWS account.

☐ **Third-party service**  
You plan to use this access key to enable access for a third-party application or service that monitors or manages your AWS resources.

☐ **Application running outside AWS**  
You plan to use this access key to authenticate workloads running in your data center or other infrastructure outside of AWS that needs to access your AWS resources.

☐ **Other**  
Your use case is not listed here.

Figure 34: Create Access Key

- The Access key ID and secret key are shown in the page, note them down for use in the prediction and alerting script:

**Access key created**  
This is the only time that the secret access key can be viewed or downloaded. You cannot recover it later. However, you can create a new access key any time.

Step 1  
Access key best practices & alternatives

Step 2 - optional  
Set description tag

Step 3  
**Retrieve access keys**

### Retrieve access keys [info](#)

**Access key**  
If you lose or forget your secret access key, you cannot retrieve it. Instead, create a new access key and make the old key inactive.

Access key	Secret access key
AKIATDDM2JMEJH6TNGPI	***** <a href="#" style="color: #0070c0;">Show</a>

**Access key best practices**

- Never store your access key in plain text, in a code repository, or in code.
- Disable or delete access key when no longer needed.
- Enable least-privilege permissions.
- Rotate access keys regularly.

For more details about managing access keys, see the [best practices for managing AWS access keys](#).

[Download .csv file](#)
[Done](#)

Figure 35: IAM Role Access Key ID and Secret

## AWS Cloudwatch Logs Configuration:

To configure the AWS Cloudwatch to get the incoming logs from the webapplication hosted on EC2 instance, follow the steps below:

1. Go to the 'Cloudwatch Management Console' in AWS dashboard and open the 'Log Groups' Screen.
2. Create a new Log Group on the section as shown below:

### Create log group

**Log group details** [info](#)

CloudWatch Logs offers two log classes: Standard and Infrequent Access. [Learn more about the features offered by each log class.](#)

**Log group name**

**Retention setting**

**Log class** [info](#)

**KMS key ARN - optional**

**Tags**

A tag is a label that you assign to an Amazon Web Services resource. Each tag consists of a key and an optional value. You can use tags to search and filter your resources or track your Amazon Web Services costs.

No tags are associated with this log group.

Add new tag

You can add up to 50 more tag(s).

[Cancel](#)
[Create](#)

Figure 36: Cloudwatch Create Log Group

3. Click on the newly created 'Log Group' and then create a new 'Log Stream'.

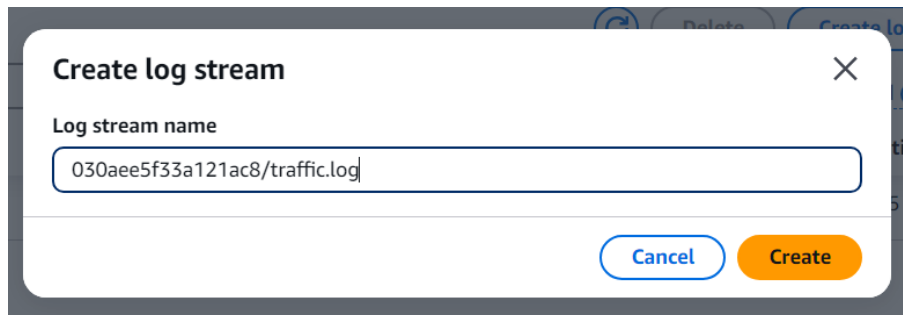


Figure 37: Log Stream Creation

4. The cloudwatch logs setup is now completed

### Running the Web-Application:

For running the web-application on AWS EC2 instance created, follow the below steps:

1. Connect to the EC2 Instance using any terminal by copying the connect command from 'connect' tab of the EC2 instance created:

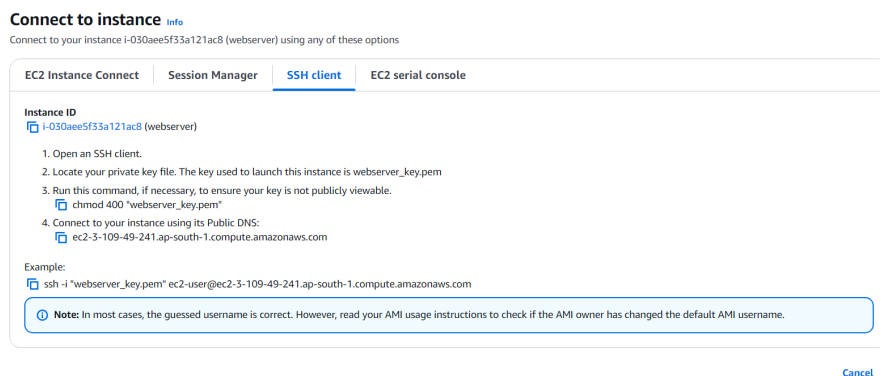


Figure 38: Connect to EC2 Instance

2. Copy the 'flask\_app' folder into the ec2 instance home directory.

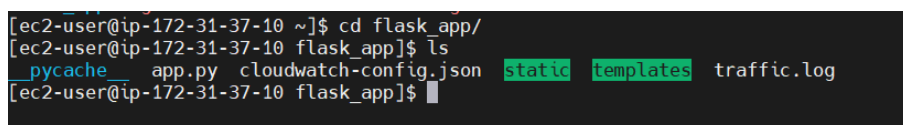


Figure 39: EC2 Instance Flask Directory

3. Next, cd into the 'flask\_app' folder and install the python pip and other dependent libraries using the commands below:

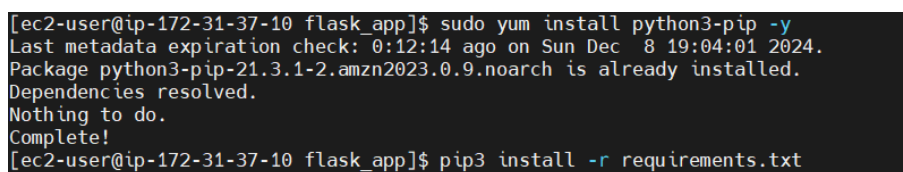


Figure 40: Install pip and python libraries

4. After installation, the next step is to put flask as a system service that starts on any reboot or keeps running as a service. For this copy the file contents of the file 'flask\_app.service' and run the following command to open vim editor and paste the file contents inside this service file:

```
'sudo vim /etc/systemd/system/flask_app.service'
```

```
[Unit]
Description=Gunicorn instance to serve Flask app
After=network.target

[Service]
User=ec2-user
Group=ec2-user
WorkingDirectory=/home/ec2-user/flask_app
Environment="PATH=/home/ec2-user/.local/bin:/usr/bin:/bin"
ExecStart=/home/ec2-user/.local/bin/gunicorn --workers 3 --bind 0.0.0.0:5000 app:app
StandardOutput=journal
StandardError=journal

[Install]
WantedBy=multi-user.target
```

Figure 41: Flask Service Code

5. Next, run the command below to enable the flask service to start and to keep running in background:

```
'sudo systemctl enable flask_app'
```

6. Then run the system daemon reload command for the changes to take effect:

```
'sudo systemctl daemon-reload'
```

7. Next, start the flask service using the command 'start' and verify its status:

```
'sudo systemctl start flask_app'
'sudo systemctl status flask_app'
```

Verify the status of the flask service:

```
[ec2-user@ip-172-31-37-10 flask_app]$ sudo systemctl status flask_app
● flask_app.service - Gunicorn instance to serve Flask app
   Loaded: loaded (/etc/systemd/system/flask_app.service; enabled; preset: disabled)
   Active: active (running) since Sun 2024-12-08 19:03:55 UTC; 25min ago
     Main PID: 1996 (gunicorn)
       Tasks: 11 (1 limit: 1113)
      Memory: 158.0M
         CPU: 4.301s
    CGroup: /system.slice/flask_app.service
            └─1996 /usr/bin/python3 /home/ec2-user/.local/bin/gunicorn --workers 3 --bind 0.0.0.0:5000 app:app
              └─2006 /usr/local/bin/ngrok http 5000
                └─2080 /usr/bin/python3 /home/ec2-user/.local/bin/gunicorn --workers 3 --bind 0.0.0.0:5000 app:app
                  └─2087 /usr/bin/python3 /home/ec2-user/.local/bin/gunicorn --workers 3 --bind 0.0.0.0:5000 app:app
                    └─2102 /usr/bin/python3 /home/ec2-user/.local/bin/gunicorn --workers 3 --bind 0.0.0.0:5000 app:app

Dec 08 19:03:55 ip-172-31-37-10.ap-south-1.compute.internal systemd[1]: Starting flask_app.service - Gunicorn instance to serve Flask app...
Dec 08 19:03:55 ip-172-31-37-10.ap-south-1.compute.internal systemd[1]: Started flask_app.service - Gunicorn instance to serve Flask app.
Dec 08 19:03:55 ip-172-31-37-10.ap-south-1.compute.internal gunicorn[1996]: [2024-12-08 19:03:55 +0000] [1996] [INFO] Starting gunicorn 23.0.0
Dec 08 19:03:55 ip-172-31-37-10.ap-south-1.compute.internal gunicorn[1996]: [2024-12-08 19:03:55 +0000] [1996] [INFO] Listening at: http://0.0.0.0:5000 (1996)
Dec 08 19:03:55 ip-172-31-37-10.ap-south-1.compute.internal gunicorn[1996]: [2024-12-08 19:03:55 +0000] [1996] [INFO] Using worker: sync
Dec 08 19:03:55 ip-172-31-37-10.ap-south-1.compute.internal gunicorn[2080]: [2024-12-08 19:03:55 +0000] [2080] [INFO] Booting worker with pid: 2080
Dec 08 19:03:55 ip-172-31-37-10.ap-south-1.compute.internal gunicorn[2087]: [2024-12-08 19:03:55 +0000] [2087] [INFO] Booting worker with pid: 2087
Dec 08 19:03:55 ip-172-31-37-10.ap-south-1.compute.internal gunicorn[2102]: [2024-12-08 19:03:55 +0000] [2102] [INFO] Booting worker with pid: 2102
[ec2-user@ip-172-31-37-10 flask_app]$
```

Figure 42: Flask Service Status

8. To configure the cloudwatch agent, use the file 'cloudwatch-config.json' provided below:

```
> {} cloudwatch-config.json > ...
{
  "logs": {
    "logs_collected": {
      "files": {
        "collect_list": [{
          "file_path": "/home/ec2-user/flask_app/traffic.log",
          "log_group_name": "MyAppTrafficLogs",
          "log_stream_name": "030aee5f33a121ac8/traffic.log",
          "timezone": "UTC"
        }]
      }
    }
  }
}
```

Figure 43: AWS Cloudwatch Config

Ensure the values and the paths inside this file match the corresponding paths and names setup in the earlier steps.

- Run the following command to enable cloudwatch to start collecting and pushing the logs into the aws cloudwatch log group configured earlier:

```
'sudo /opt/aws/amazon-cloudwatch-agent/bin/amazon-cloudwatch-agent-ctl -a fetch-config -m ec2 -c file:/home/ec2-user/flask_app/cloudwatch-config.json -s'
```

- Then restart the cloudwatch agent using the below command:

```
'sudo systemctl restart amazon-cloudwatch-agent'
```

- The logs will start to come in to the log group created with the configurations done as below:

The screenshot shows the AWS CloudWatch console. The breadcrumb navigation at the top indicates the path: CloudWatch > Log groups > MyAppTrafficLogs > 030aee5f33a121ac8/traffic.log. The left sidebar contains various navigation links. The main content area is titled 'Log events' and includes a search bar and filters. Below this, a table lists log events with their timestamps and messages. The messages are JSON-formatted logs containing HTTP request details. For example, one event shows an INFO message with details like 'bytes\_in=0, bytes\_out=8577, src\_ip=47.74.39.39, src\_ip\_country\_code=JP, protocol=http, response.code=200, dst\_port=80'. Another event shows an ERROR message: 'ERROR:app:failed to log request: 'utf-8' codec can't decode byte 0xff in position 0: invalid start byte'.

Figure 44: AWS Cloudwatch Logs

## 9 Model Results

```
1 # Predict (-1 for anomaly, 1 for normal)
2 predictions = iso_forest.predict(X)
3 predictions = [1 if x == -1 else 0 for x in predictions]
4
5 # Evaluate model performance
6 report = classification_report(y, predictions, target_names=["Normal Web Traffic", "Suspicious Web Traffic"])
7 print(report)
8
9 print("CONFUSION MATRIX: \n", confusion_matrix(y, predictions))
10
11 iso_cv_score = cross_val_score(iso_forest, X, y, cv=5, scoring='f1_weighted')
12 iso_cv_score.mean()
```

✓ 1.0s

	precision	recall	f1-score	support
Normal Web Traffic	0.79	0.80	0.80	8040
Suspicious Web Traffic	0.24	0.22	0.23	2242
accuracy			0.68	10282
macro avg	0.51	0.51	0.51	10282
weighted avg	0.67	0.68	0.67	10282

CONFUSION MATRIX:

```
[[6467 1573]
 [1758  484]]
```

... np.float64(0.07283084999882561)

Figure 45: Isolation Forest Evaluation

```
1 predictions_rf = rf.predict(X)
2
3 report_rf = classification_report(y, predictions_rf, target_names=["Normal Web Traffic", "Suspicious Web Traffic"])
4 print(report_rf)
5
6 print("CONFUSION MATRIX: \n", confusion_matrix(y, predictions_rf))
```

✓ 0.2s

	precision	recall	f1-score	support
Normal Web Traffic	1.00	1.00	1.00	8040
Suspicious Web Traffic	1.00	1.00	1.00	2242
accuracy			1.00	10282
macro avg	1.00	1.00	1.00	10282
weighted avg	1.00	1.00	1.00	10282

CONFUSION MATRIX:

```
[[8040  0]
 [  4 2238]]
```

Figure 46: Random Forest Evaluation

```
1 predictions_gb = grad_boost.predict(X)
2
3 report_gb = classification_report(y, predictions_gb, target_names=["Normal Web Traffic", "Suspicious Web Traffic"])
4 print(report_gb)
5
6 print("CONFUSION MATRIX: \n", confusion_matrix(y, predictions_gb))
```

✓ 0.0s

	precision	recall	f1-score	support
Normal Web Traffic	0.80	1.00	0.89	8040
Suspicious Web Traffic	0.90	0.11	0.19	2242
accuracy			0.80	10282
macro avg	0.85	0.55	0.54	10282
weighted avg	0.82	0.80	0.74	10282

CONFUSION MATRIX:

```
[[8013  27]
 [2003 239]]
```

Figure 47: Gradient Boosting Evaluation

```

1 predictions_svc = model_svc.predict(x)
2
3 report_svc = classification_report(y, predictions_svc, target_names=["Normal Web Traffic", "Suspicious Web Traffic"])
4 print(report_svc)
5
6 print("CONFUSION MATRIX: \n", confusion_matrix(y, predictions_svc))
7
8 svm_cv_scores = cross_val_score(model_svc, X, y, cv=5, scoring='f1_weighted')
9
10 print('Cross-Validation Score for Support Vector: ', svm_cv_scores.mean())

```

✓ 123s

	precision	recall	f1-score	support
Normal Web Traffic	0.78	1.00	0.88	8840
Suspicious Web Traffic	1.00	0.00	0.01	2242
accuracy			0.78	10282
macro avg	0.89	0.50	0.44	10282
weighted avg	0.83	0.78	0.69	10282

CONFUSION MATRIX:  
[[8840 0]  
[2232 10]]

Cross-Validation Score for Support Vector: 0.68616887925916

Figure 48: Support Vector Classifier (SVC) Evaluation

```

1 def plot_metrics(history, title_prefix=""):
2     # Plot accuracy
3     plt.figure(figsize=(12, 5))
4     plt.subplot(1, 2, 1)
5     plt.plot(history.history['binary_accuracy'], label='Train Accuracy')
6     plt.plot(history.history['val_binary_accuracy'], label='Validation Accuracy')
7     plt.title(f"{title_prefix} Accuracy")
8     plt.xlabel('Epochs')
9     plt.ylabel('Accuracy')
10    plt.legend()
11
12    # Plot loss
13    plt.subplot(1, 2, 2)
14    plt.plot(history.history['loss'], label='Train Loss')
15    plt.plot(history.history['val_loss'], label='Validation Loss')
16    plt.title(f"{title_prefix} Loss")
17    plt.xlabel('Epochs')
18    plt.ylabel('Loss')
19    plt.legend()
20    plt.show()
21

```

Figure 49: Function for Plotting of Evaluation Metrics for Neural Network

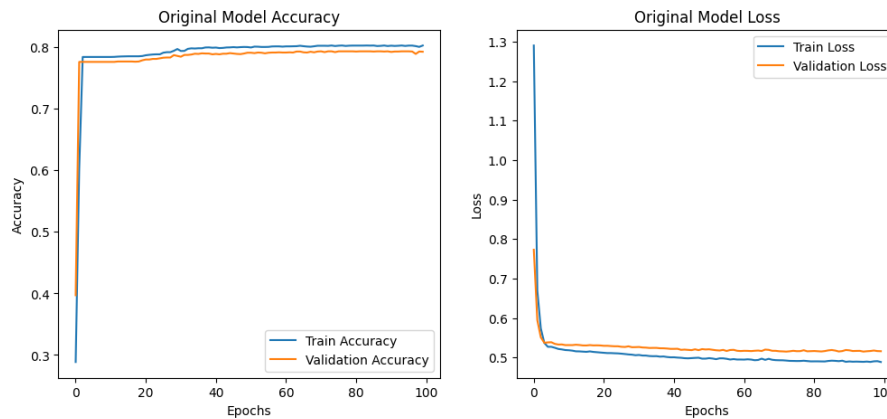


Figure 50: Neural Network Evaluation



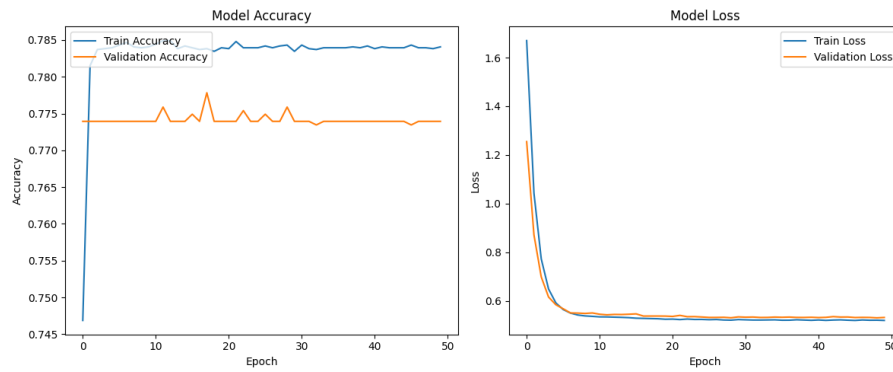


Figure 51: Fine-tuned Neural Network Evaluation

## References

AWS (no date). Cloud computing services - amazon web services (aws).

**URL:** <https://aws.amazon.com/>

JanCSG (2024).

**URL:** <https://www.kaggle.com/datasets/jancsg/cybersecurity-suspicious-web-threat-interactions>

Mailgun (no date). Transactional email api service for developers.

**URL:** <https://www.mailgun.com/>

MaxMind (no date). Geolite2 free geolocation data.

**URL:** <https://dev.maxmind.com/geoip/geolite2-free-geolocation-data/>