2024

Code explanation and Implement Changes

PROJECT ECHO

MINH DANG

Contents

A) Introduction:

Project Echo was founded in Trimester 3 of 2022 by Stephan Kokkas, Andrew Kudilczak and Daniel Gladman. The project aims to support global conservationists by developing advanced audio classification systems that can facilitate the non-intrusive monitoring, discovery and tracking endangered species and their predators within their natural habitats. The system of IoT will collect data and transmit to a center server via an API, where an AI-driven model classifies species and record vital data. Then, the Human Machine Interface (provides real time visualization of simulated animals and vocalization events on an interactive map.

In this report, I will guide you through the flow of building the contents when Docker build the app so that you can implement changes directly to the main component of the apps. This will focus only for Engine Component.

B) Definition:

1) Docker:

Docker is an open-source platform designed to simplify the development, deployment, and management of applications by using containerization. Containers are lightweight, portable, and self-sufficient units that package an application along with all its dependencies, libraries, and configurations. This ensures that the application runs consistently across different environments, eliminating the "it works on my machine" problem. Docker achieves this by leveraging containerization technology, which uses the host operating system's kernel features to isolate processes while sharing the same OS resources. This approach makes Docker containers highly efficient compared to traditional virtual machines. Docker's ecosystem includes tools like Docker Engine (to build and run containers), Docker Compose (to manage multi-container applications), and Docker Hub (a repository for sharing and distributing containerized applications). It is widely used in software development, DevOps, and cloud computing for streamlining workflows, improving scalability, and enabling rapid deployment.

2) Docker Compose:

Docker Compose is a tool provided by Docker that simplifies the process of defining and managing multi-container Docker applications. It allows developers to define the services, networks, and volumes their application requires in a single YAML configuration file, typically named docker-compose.yml. This file describes how each container should be built, configured, and connected, making it easy to orchestrate multiple containers as part of a single application.

With Docker Compose, you can:

1. **Define Multi-Container Applications**: Compose enables you to define multiple services in one file. For example, a web application might need a web server, a database, and a caching service.
2. **Manage Dependencies**: Compose automatically handles the dependencies between containers, ensuring that services start in the correct order and are properly linked.

3. **Simplify Workflow**: You can manage your entire application stack with simple commands like docker-compose up to start the application and docker-compose down to stop and clean up.
4. **Use Shared Configuration**: Docker Compose supports the reuse of configurations through YAML anchors, variables, and overriding files, making it easier to handle different environments like development, testing, and production.
5. **Integrate with Docker Networking**: Compose creates a dedicated network for your application, enabling seamless communication between containers without manual configuration.

Key Commands:

- docker-compose up: Builds, (re)creates, and starts all containers as defined in the YAML file.
- docker-compose down: Stops and removes containers, networks, and other resources created by up.
- docker-compose logs: Displays logs from all running containers.

C) Detail of the code:

1) Docker-compose.yml:

Starts with the first file docker-compose.yml, this is the file where docker compose uses to start building the app. For Engine, the content includes:

- Version: the file format version being used.
- Services: defines the containers that make up the application.
- Build: Create the app using the file “dockerfile” in directory “context”. The arguments are listed out as parameters to build this.
- Image: Name the docker image after build.
- Container name, command, network, ports, volumes: Specify container name running for service, execution command, network target, map host port to container port and mount the configuration of google cloud (contains credential keys).
- Stdin_open: Set the status of keeping standard input for interactive mode.
- Tty: Allocate a pseudo TTY (a terminal interface) for container.

As can be seen in the code below, Engine components are divided into two parts. While the first one handles the learning model, the second one includes important codes to execute and use the model

```yaml
version: "3.8"

services:
  model_server:
    build:
      context: ./Engine
      dockerfile: Model.Dockerfile
      args:
        BASE_IMAGE: tensorflow/serving:2.3.0
        # BASE_IMAGE: emacski/tensorflow-serving # Uncomment if on macos and comment above line
    image: ts-echo-model
    container_name: ts-echo-model-cont
    command: --model_config_file=/models/models.config
    networks:
      - echo-net
    ports:
      - "8501:8501"
    volumes:
      - credentials_volume:/root/.config/gcloud/
    stdin_open: false
    tty: true

  echo_engine:
    build:
      context: ./Engine
      dockerfile: Engine.Dockerfile
      args:
        BASE_IMAGE: tensorflow/tensorflow:latest-gpu
        # BASE_IMAGE: armswdev/tensorflow-arm-neoverse:r24.03-tf-2.15.1-onednn-acl #Uncomment if on MacOS and comment above line
    image: ts-echo-engine
    container_name: ts-echo-engine-cont
    networks:
      - echo-net
    volumes:
      - credentials_volume:/root/.config/gcloud/
    stdin_open: true
    tty: true
```

Figure 1: Engine part in Docker Compose File.

2) Model and Engine Docker File:

Moving on to the building files that are used to set up environment. Below is the code for both files:

Model.Dockerfile

```
# Use an official TensorFlow serving image as the base image
# FROM emacski/tensorflow-serving
ARG BASE_IMAGE
FROM ${BASE_IMAGE}
# RUN apt-get update && apt-get install -y build-essential

# Copy the model to the container
COPY models/ ./models/
COPY models.config ./models/


# make the container directory for credentials
# RUN mkdir -p ~/.config/gcloud/

# Set the environment variable for the model name and version
# ENV MODEL_NAME=echo_model
# ENV MODEL_VERSION=1

# Expose the port for TensorFlow serving
EXPOSE 8501
```

Engine.Dockerfile

```
# Use an official TensorFlow GPU image as the base image
ARG BASE_IMAGE
FROM ${BASE_IMAGE}
#FROM python:3.9
USER root
# Set the working directory
WORKDIR /app

# Copy the requirements.txt file into the container
COPY requirements.txt ./

# Copy yamnet to the container
COPY yamnet_dir/ ./yamnet_dir/

# Install any needed Python packages specified in requirements.txt
ENV DEBIAN_FRONTEND=noninteractive

RUN apt-get update
RUN apt-get install -y libopenexr-dev dos2unix
RUN apt-get install pkg-config -y
RUN python3 -m pip install --upgrade pip
RUN pip download -r requirements.txt
RUN pip install -r requirements.txt

#Comment below line to run in mac M-chips
RUN pip install tensorflow==2.15.0


RUN echo "deb [signed-by=/usr/share/keyrings/cloud.google.gpg] https://packages.cloud.google.com/

# make the container directory for credentials
WORKDIR /root
RUN mkdir -p .config/gcloud/

# Copy the rest of the application code into the container
WORKDIR /app
COPY ./echo_engine.py ./
COPY ./echo_engine.sh ./
COPY ./echo_engine.json ./
COPY ./echo_credentials.json ./
COPY ./helpers ./helpers
RUN chmod +x ./echo_engine.sh
RUN dos2unix ./echo_engine.sh

CMD ["/app/echo_engine.sh"]
```

Figure 2: Model and Engine Dockerfile.

As can be seen in the code, while in the model file, the system only needs to copy the files to the target. This includes the model itself and its configuration. However, for the main code of engine, the code needs to update the environments before installing the requirements lists. The important libraries are:

```
audiomentations
diskcache
geopy
google-cloud
google-cloud-storage
ipykernel
librosa
matplotlib
numpy
paho-mqtt==1.6.1
pandas
pymongo[srv]
soundfile
# tensorflow==2.15.0
xgboost
# torch==1.9.0+cpu
# torchvision==0.10.0+cpu
```

Figure 3: Required libraries.

- Google cloud: Gain access to Echo database.
- Librosa: Process audio inputs.
- Tensorflow and torch: Assist in using training models.

Once installing the required libraries, the Docker will copy all required engine files into the work directory (which is “app”). These files include:

- Echo Credential: The credential key to access Echo Database.
- Helper: Additional functions. In this case, this is the conversion from Mel spectrogram to CAM.
- Echo engine: While the json files includes the configuration of the Echo app, sh file check the credential key to establish the connection with database server. In this guide, I will focus on the main echo engine python file.

3) Echo Engine.py:

This is the main processing files where changes can be implemented to find the best performance of the application.

a. Main:

The main function contains two steps:

- Initialize Echo object.
- Run the Execute function.

```
if __name__ == "__main__":

    engine = EchoEngine()
    engine.execute()
```

Figure 4: Main execution.

a. Initialize:

```python
def __init__(self) -> None:

    # Load the engine config JSON file into a dictionary
    try:
        file_path = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'echo_engine.json')
        with open(file_path, 'r') as f:
            self.config = json.load(f)
        self.config['AUDIO_WINDOW'] = None
        print(f"Echo Engine configuration successfully loaded", flush=True)
    except:
        print(f"Could not engine config : {file_path}")

    # Load the project echo credentials into a dictionary
    try:
        file_path = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'echo_credentials.json')
        with open(file_path, 'r') as f:
            self.credentials = json.load(f)
        print(f"Echo Engine credentials successfully loaded", flush=True)
    except:
        print(f"Could not engine credentials : {file_path}")

    # Setup database client and connect
    try:
        # database connection string
        self.connection_string=f"mongodb://{self.credentials['DB_USERNAME']}:{self.credentials['DB_PASSWORD']}@{self.config['DB_HOSTNAME']}/EchoNet"

        myclient = pymongo.MongoClient(self.connection_string)
        self.echo_store = myclient["EchoNet"]
        print(f"Found echo store database names: {myclient.list_database_names()}", flush=True)
    except:
        print(f"Failed to establish database connection", flush=True)
```

Figure 5: Initialization.

For initialization, the function loads the json config files, followed by echo credential key to access to database. Then, initial connection string is set up for client and database connect for MongoDB.

b. Execute function:

```python
def execute(self):
    print("Engine started.")
    client = paho.Client()
    client.on_subscribe = self.on_subscribe
    client.on_message = self.on_message

    # retry connection until this succeeds
    connected = False
    while not connected:
        try:
            client.connect(self.config['MQTT_CLIENT_URL'], self.config['MQTT_CLIENT_PORT'])
            connected=True
        except:
            time.sleep(1)

    print(f'Subscribing to MQTT: {self.config["MQTT_CLIENT_URL"]} {self.config["MQTT_PUBLISH_URL"]}')
    client.subscribe(self.config['MQTT_PUBLISH_URL'])

    print("Retrieving species names from GCP")
    self.class_names = self.gcp_load_species_list()

    for cs in self.class_names:
        print(f" class name {cs}")

    print("Engine waiting for audio to arrive...")
    client.loop_forever()
```

Figure 6: Execute function.

Using the paho-mqtt library, the function initializes an MQTT client, followed by configures on_subscribe and on_message callbacks for subscribing topics and processing incoming messages. Then, using the configuration, the system will establish a connection to the MQTT and will keep trying until it reached. Finally, the function load species names and enters an infinite listening loop.

c. On subscribe and on message:

For on subscribe, there is only one print notification for debugging purpose.

```python
def on_subscribe(self, client, userdata, mid, granted_qos):
    print(f"Subscribed: message id {mid} with qos {granted_qos}")
```

Figure 7: Subscribe function.

For on message, the function includes multiple steps:

```python
def on_message(self, client, userdata, msg):
    print("Recieved audio message, processing via engine model...")
    try:
        audio_event = json.loads(msg.payload)
        print(audio_event['timestamp'])

        audio_clip = ""
        image = None
        sample_rate = 0

        if(audio_event['audioFile'] == "Recording_Mode"): # classic model
            # convert to string representation of audio to binary for processing
            print("Recording_Mode")
            audio_clip = self.string_to_audio(audio_event['audioClip'])

            image, audio_clip, sample_rate = self.combined_pipeline(audio_clip, "Recording_Mode

            # update the audio event with the re-sampled audio
            audio_event["audioClip"] = self.audio_to_string(audio_clip)

            image = tf.expand_dims(image, 0)

            #returned is melspectrogram with cam overlay,
            #TODO: Can add this image to database
            cam = melspectrogram_to_cam.convert(image)
            image_list = image.numpy().tolist()

            # Run the model via tensorflow serve
            data = json.dumps({"signature_name": "serving_default", "inputs": image_list})
            url = self.config['MODEL_SERVER']
            headers = {"content-type": "application/json"}
            json_response = requests.post(url, data=data, headers=headers)
            model_result    = json.loads(json_response.text)
            predictions = model_result['outputs'][0]

            # Predict class and probability using the prediction function
            predicted_class, predicted_probability = self.predict_class(predictions)

            print(f'Predicted class : {predicted_class}')
            print(f'Predicted probability : {predicted_probability}')

            # populate the database with the result
            self.echo_api_send_detection_event(
                audio_event,
                sample_rate,
                predicted_class,
```

```python
def on_message(self, client, userdata, msg):
            self.echo_api_send_detection_event(
                audio_event,
                sample_rate,
                predicted_class,
                predicted_probability)


            image = tf.expand_dims(image, 0)

            image_list = image.numpy().tolist()

        elif(audio_event['audioFile'] == "Recording_Mode_V2"):
            # convert to string representation of audio to binary for processing
            print("Recording_Mode_V2")
            sample_rate = 16000
            audio_clip = self.string_to_audio(audio_event['audioClip'])
            file = io.BytesIO(audio_clip)
            #wav = 'yamnet_dir/cat-goat-dingo.wav'
            data_frame, audio_clip = self.sound_event_detection(file, sample_rate)
            iteration_count = 0

            for index, row in data_frame.iterrows():
                #start_time end_time    echonet_label_1 echonet_confidence_1
                start_time = float(row['start_time'])
                end_time = float(row['end_time'])
                predicted_class = row['echonet_label_1']

                if predicted_class == "Sus_Scrofa":
                    predicted_class = "Sus Scrofa"

                predicted_probability = round(float(row['echonet_confidence_1']) * 100.0, 2)

                print(f'Predicted class : {predicted_class}')
                print(f'Predicted probability : {predicted_probability}')

                audio_subsection = self.load_specific_subsection(audio_clip, start_time, end_time, sample_rate)

                # update the audio event with the re-sampled audio
                audio_event["audioClip"] = self.audio_to_string(audio_subsection)

                new_lat = audio_event['animalEstLLA'][0]
                new_lon = audio_event['animalEstLLA'][1]

                if(iteration_count > 0):
                    lat = audio_event['animalEstLLA'][0]
                    lon = audio_event['animalEstLLA'][1]
```

```python
class EchoEngine():
    def on_message(self, client, userdata, msg):

                    if(iteration_count > 0):
                        lat = audio_event['animalEstLLA'][0]
                        lon = audio_event['animalEstLLA'][1]

                        new_lat, new_lon = self.generate_random_location(lat, lon, 50, 100)

                    new_lla = [new_lat, new_lon, 0.0]

                    audio_event['animalEstLLA'] = new_lla
                    audio_event['animalTrueLLA'] = new_lla

                    # populate the database with the result
                    self.echo_api_send_detection_event(
                        audio_event,
                        sample_rate,
                        predicted_class,
                        predicted_probability)

                    iteration_count = iteration_count + 1

                else: # simulate animals mode
                    # convert to string representation of audio to binary for processing
                    audio_clip = self.string_to_audio(audio_event['audioClip'])

                    image, audio_clip, sample_rate = self.combined_pipeline(audio_clip, "Animal_Mode")

                    #returned is melspectrogram with cam overlay,
                    #TODO: Can add this image to database
                    cam = melspectrogram_to_cam.convert(image)


                    # update the audio event with the re-sampled audio
                    audio_event["audioClip"] = self.audio_to_string(audio_clip)

                    image = tf.expand_dims(image, 0)

                    image_list = image.numpy().tolist()

                    # Run the model via tensorflow serve
                    data = json.dumps({"signature_name": "serving_default", "inputs": image_list})
                    url = self.config['MODEL_SERVER']
                    headers = {"content-type": "application/json"}
                    json_response = requests.post(url, data=data, headers=headers)
                    model_result    = json.loads(json_response.text)
                    predictions = model_result['outputs'][0]
```

```python
                    # update the audio event with the re-sampled audio
                    audio_event["audioClip"] = self.audio_to_string(audio_clip)

                    image = tf.expand_dims(image, 0)

                    image_list = image.numpy().tolist()

                    # Run the model via tensorflow serve
                    data = json.dumps({"signature_name": "serving_default", "inputs": image_list})
                    url = self.config['MODEL_SERVER']
                    headers = {"content-type": "application/json"}
                    json_response = requests.post(url, data=data, headers=headers)
                    model_result    = json.loads(json_response.text)
                    predictions = model_result['outputs'][0]

                    # Predict class and probability using the prediction function
                    predicted_class, predicted_probability = self.predict_class(predictions)

                    print(f'Predicted class : {predicted_class}')
                    print(f'Predicted probability : {predicted_probability}')

                    # populate the database with the result
                    self.echo_api_send_detection_event(
                        audio_event,
                        sample_rate,
                        predicted_class,
                        predicted_probability)

                    image = tf.expand_dims(image, 0)

                    image_list = image.numpy().tolist()

            except Exception as e:
                # Catch the exception and print it to the console
                print(f"An error occurred: {e}", flush=True)
```

Figure 8: on_message function.

- Receive and process audio message by printing log and parse in coming message.
- Initialize variables: audio clip, image, sample rate, etc.

- There are three cases in this:

1. **Recording mode:**

- Convert the audioClip (string) to a binary representation for processing.
- Process the audio through combined_pipeline to obtain image, re-sampled audio_clip, and sample_rate.
- Update the audioClip in the audio_event dictionary with the re-sampled audio.
- Expand the image tensor for further processing.
- Convert the spectrogram image to a Class Activation Map (CAM) overlay using melspectrogram_to_cam.
- Prepare the image data for inference by converting it into a list format.
- Send the image data to a TensorFlow model server for prediction via HTTP POST request.
- Parse the model server response to extract predictions.
- Determine the predicted class and probability using predict_class.
- Print the predicted class and probability to the console.
- Send the detection event details to the database using echo_api_send_detection_event.

2. **Recording mode V2:**

- Convert the audioClip to binary and process it via sound_event_detection to obtain a dataframe and processed audio.
- Iterate through the rows of the dataframe:
- Extract start_time, end_time, echonet_label_1, and echonet_confidence_1 for each detected sound event.
- Adjust the predicted class if necessary (e.g., changing "Sus_Scrofa" to "Sus Scrofa").
- Print the predicted class and probability.
- Extract audio subsections based on start and end times using load_specific_subsection.
- Update the audioClip in the audio_event dictionary with the audio subsection.
- Randomize location attributes (animalEstLLA and animalTrueLLA) for subsequent iterations.
- Send the detection event details to the database.

3. **Default:**

- Convert the audioClip to binary and process it via combined_pipeline to obtain image, re-sampled audio_clip, and sample_rate.
- Generate a CAM overlay using melspectrogram_to_cam.
- Update the audioClip in the audio_event dictionary.
- Prepare the image data for TensorFlow model server inference.
- Send the image data for prediction and parse the response to extract predictions.
- Determine the predicted class and probability using predict_class.
- Print the predicted class and probability.
- Send the detection event details to the database.

4. **Error handling:**

- Catch and record any error.

- These are the functions that have been called for this function:

  - **String to audio and audio to string:** Handling base 64 encoding and decoding to ensure that the audio binary and string inputs are validated, avoid malformed data error.
  - **Combined pipeline:**

```python
class EchoEngine():
    def combined_pipeline(self, audio_clip, mode):

        sample_rate = 0

        if(str(mode) == "Recording_Mode"):
            print("recording mode 1")
            # Create a file-like object from the bytes.
            file = io.BytesIO(audio_clip)

            # Load the audio data with librosa
            audio_clip, sample_rate = librosa.load(file, sr=self.config['AUDIO_SAMPLE_RATE'])

        elif(str(mode) == "Recording_Mode_V2"):
            print("recording mode 2")
            # Create a file-like object from the bytes.
            file = io.BytesIO(audio_clip)

            # Load the audio data with librosa
            audio_clip, sample_rate = librosa.load(file, sr=self.config['AUDIO_SAMPLE_RATE'])

        else:
            print("animal mode")
            # Create a file-like object from the bytes.
            file = io.BytesIO(audio_clip)

            # Load the audio data with librosa
            audio_clip, sample_rate = librosa.load(file, sr=self.config['AUDIO_SAMPLE_RATE'])

        # keep right channel only
        if audio_clip.ndim == 2 and audio_clip.shape[0] == 2:
            audio_clip = audio_clip[1, :]

        # cast to float32 type
        audio_clip = audio_clip.astype(np.float32)

        # analyse a random 5 second subsection
        audio_clip = self.load_random_subsection(audio_clip, duration_secs=self.config['AUDIO_CLIP_DURATION'])

        # Compute the mel-spectrogram
        image = librosa.feature.melspectrogram(
            y=audio_clip,
            sr=self.config['AUDIO_SAMPLE_RATE'],
            n_fft=self.config['AUDIO_NFFT'],
            hop_length=self.config['AUDIO_STRIDE'],
            n_mels=self.config['AUDIO_MELS'],
            fmin=self.config['AUDIO_FMIN'],
            fmax=self.config['AUDIO_FMAX'],
            win_length=self.config['AUDIO_WINDOW'])

        # Optionally convert the mel-spectrogram to decibel scale
        image = librosa.power_to_db(
            image,
            top_db=self.config['AUDIO_TOP_DB'],
            ref=1.0)

        # Calculate the expected number of samples in a clip
        expected_clip_samples = int(self.config['AUDIO_CLIP_DURATION'] * self.config['AUDIO_SAMPLE_RATE'] / self.config['AUDIO_STRIDE'])

        # swap axis and clip to expected samples to avoid rounding errors
        image = np.moveaxis(image, 1, 0)
        image = image[0:expected_clip_samples,:]

        # reshape into standard 3 channels to add the color channel
        image = tf.expand_dims(image, -1)

        # most pre-trained model classifer model expects 3 color channels
        image = tf.repeat(image, self.config['MODEL_INPUT_IMAGE_CHANNELS'], axis=2)

        # calculate the image shape and ensure it is correct
        expected_clip_samples = int(self.config['AUDIO_CLIP_DURATION'] * self.config['AUDIO_SAMPLE_RATE'] / self.config['AUDIO_STRIDE'])
        image = tf.ensure_shape(image, [expected_clip_samples, self.config['AUDIO_MELS'], self.config['MODEL_INPUT_IMAGE_CHANNELS']])

        # note here a high quality LANCZOS5 is applied to resize the image to match model image input size
        image = tf.image.resize(image, (self.config['MODEL_INPUT_IMAGE_WIDTH'],self.config['MODEL_INPUT_IMAGE_HEIGHT']),
                                method=tf.image.ResizeMethod.LANCZOS5)

        # rescale to range [0,1]
        image = image - tf.reduce_min(image)
        image = image / (tf.reduce_max(image)+0.0000001)

        return image, audio_clip, sample_rate
```

Figure 9: Combine pipeline.

The function load the data from ByteIO object into audio clip based on input mode. Then, the code check so see whether it should keep only the right channel, followed by converting to float 32 to ensure subsequence process. After extracting random subsection of the audio clip of a specified duration, the function calculate mel spectrogram, adjust the shape before normalizing to range (0, 1).

- **Predict class:** Extracting the predicted class and its probability, round by 2 decimal numbers.
- **Echo Api sends detection event:** Declare the sending format.
- **Sound event detection:**

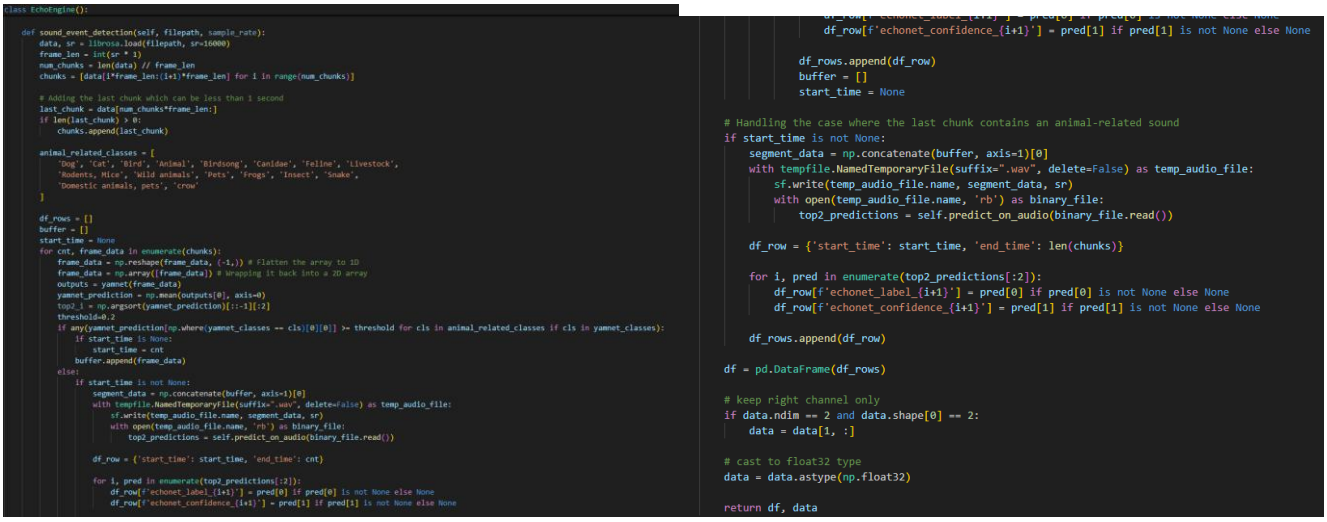


Figure 10: Prediction.

For this function, the input read before resampling to 16 hHz, followed by dividing the audio into 1 second chunks and process if the last chunk is shorter. Using the confidence threshold of 0.2, the function iterates over all chunks, put it into Yamnet model for extracting the top 2 predicted classes. Then, if any class is in the list, the buffering chunks and record time are started. Otherwise, keep processing buffered audio as a detected segment. For each detected segment, it will be written as temporary wav files and store segment data (start and end time, top 2 labels and confidence) in a dictionary. Once the last buffered segment is processed, the function collects all segment data before casting the audio data to float 32 and return the output as df and processed audio data.

- **Load specific subsection**: After calculating the audio duration and adjust end time to be smaller than total duration, the start and end time are converted into sample indices using sample rate. Finally, the function check subsection and pad if needed before returning the subsection.

4) Weather Pipeline:

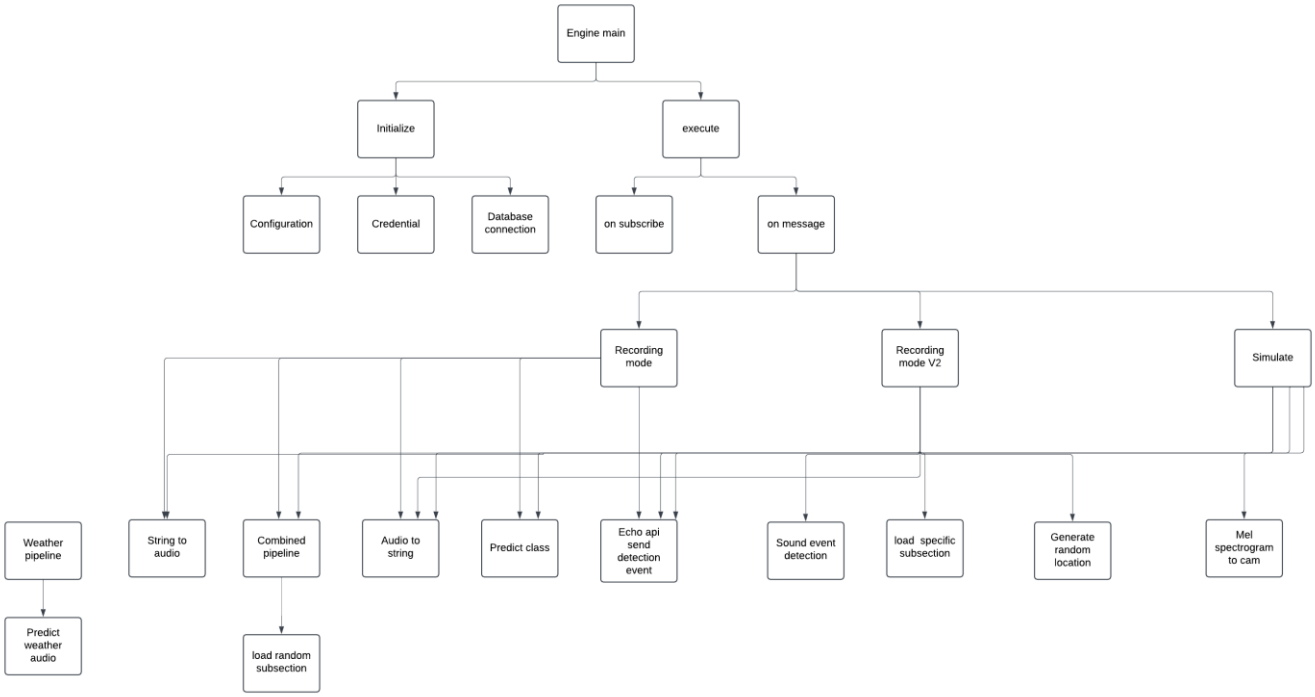
```
class EchoEngine():
    def weather_pipeline(self, audio_clip):
        file = io.BytesIO(audio_clip)
        # Load the audio data with librosa
        audio, sample_rate = librosa.load(file, sr=self.config['WEATHER_SAMPLE_RATE'])
        required_samples = self.config['WEATHER_SAMPLE_RATE'] * self.config['WEATHER_CLIP_DURATION']
        if len(audio) < required_samples:
            audio = np.pad(audio, (0, required_samples - len(audio)), 'constant')
        else:
            audio = audio[:required_samples]

        mel_spectrogram = librosa.feature.melspectrogram(
            y=audio, sr=self.config['WEATHER_SAMPLE_RATE'],
            n_fft=self.config['AUDIO_NFFT'],
            hop_length=self.config['AUDIO_STRIDE'],
            n_mels=self.config['AUDIO_MELS'],
            fmin=self.config['AUDIO_FMIN'],
            fmax=self.config['AUDIO_FMAX']
        )
        log_mel_spectrogram = librosa.power_to_db(mel_spectrogram, top_db=self.config['AUDIO_TOP_DB'])
        spectrogram_resized = tf.image.resize(log_mel_spectrogram[np.newaxis, :, :, np.newaxis], [260, 260])
        spectrogram_resized = np.repeat(spectrogram_resized, 3, axis=-1)
        return spectrogram_resized, audio, self.config['WEATHER_SAMPLE_RATE']

    def predict_weather_audio(self, audio_clip):
        """
        Call with audio clip, it will call weather detection model running on model container
        """
        image, audio, sample_rate = self.weather_pipeline(audio_clip)

        image = tf.expand_dims(image, 0)

        image_list = image.numpy().tolist()

        data = json.dumps({"signature_name": "serving_default", "inputs": image_list})
        url = self.config['WEATHER_SERVER']
        headers = {"content-type": "application/json"}
        json_response = requests.post(url, data=data, headers=headers)
        model_result   = json.loads(json_response.text)
        predictions = model_result['outputs'][0]
        print("Weather Prediciton", predictions)
        #TODO: Map prediciton of appropriate class label
        return predictions
```

Figure 11: Weather Pipeline.

In this pipeline, the functions are created to predict the weather data. Using audio clip in bytes format as input, the files are converted as object and adjust the file by either padding or truncating to the required length. Then, the mel spectrogram is calculated and change into log-mel spectrogram, followed by resizing and converting to RGB like format. Once the processing steps are completed, the function sends request to the server to retrieve the prediction classes and probability.

5) Recap my observation:

Below is my Recap for the flow of code based on my observation. This will help you identify the place to implement changes. Note that the above functions are the main flow of the code to avoid unnecessary long explanation.



D) Conclusion:

With this guide, I hope that I can help you understand more about the project. This will help you easier to implement changes to the code and improve the performance. In this report, I only focus mostly on the main Engine function. There are more parts of the code that needs to be digested deeper. If is there any errors that I might made in this report, feel free to adjust if need. Thank you.