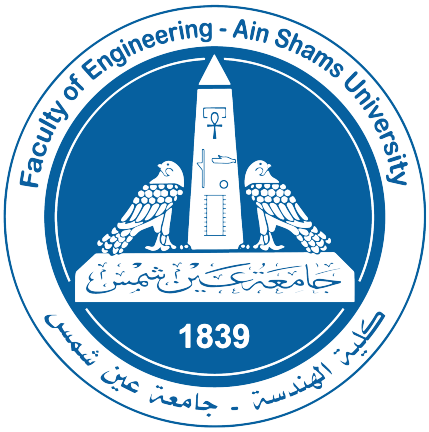

Virtual Mock Interview

Marawan Osama Sherif 18P1416

Ahmed Mohamed Ahmed Aly 18P9313

Ahmed Mohammed Mahmoud Khorkhash 18P7455

Supervisor: Dr. Islam El Maddah

Department of Computer Engineering and Software Systems

Ain Shams University Faculty of Engineering

2023

Abstract

Interviews are quite an important indicator of whether or not a candidate is as competent as their CV might show. From the interviewer's point of view, it is quite a routine procedure; the interviewee just answers some questions and based on their answers, they obtain whether the answers are satisfactory enough to proceed to the next stage. However, for the interviewee, that is not quite the case. In spite of the great strides in technological advancements the last couple of decades provided, there are still no concrete ways for an interviewee to prepare well for an interview other than to memorize or solve multiple recurring questions. Even that does not prepare them in other aspects, such as tone of voice or general temperamental behaviors like facial expressions.

Therefore, we evaluated the best method to use current technology to help these interviewees with these issues. We created a form with the biggest pains an interviewee goes through, and then from there we manifested a web application using AWS, React and MVC that analyzes everything. From their tone of voice and facial expressions to their answers and words used. Interviewees can now use this tool to help hone their skills. The opposite can also occur where human resources in each company can utilize this tool to effectively analyze each candidate to ensure that only the best of each can pass through to the next stage.

Contents

List of Figures

List of Tables

1 Introduction

Interview anxiety is a very common behavior in the world today. Despite how good you are technically, how qualified you are to the job, or even if you already know most of the questions before the interview, people still get interview anxieties. Meeting strangers in a position of authority; talking about yourself; being evaluated and judged on your appearance, demeanor, and ability to sell yourself; these are all triggers for nerves and stress.

Despite the efforts of candidates to overcome their interview anxieties by memorizing behavior tips on what and what not to do during interviews they usually forget them if they did not practice enough and did not mock interviews with a career counsellor. Unfortunately, mock interviews are not always available as it is usually costly, require arranging times with a career counsellor, and might be biased based on the counsellor's own opinion.

The overwhelming number of applicants is another issue that arises when companies are recruiting. This problem is one of the most devastating issues that faces growing companies with small or even medium size human resources department; to solve this issue companies usually tend to use outsourcing HR to interview these applicants and see if they have the required skills for the open job. However, this process costs a lot of money and its efficiency is not the best as outsourcing HR won't have the technical knowledge of the open vacancies, thus, they are used to do non technical interviews as another filtering process to applicants.

Analysing both problems, we can easily identify that the common factor between both problems is the human actor:

- Filtering enormous number of applicants can be very costly to the recruiter company in terms of finance, time and resources.
- Candidates who want to practice for their interviews cannot find a good career counselor that is available anytime, inexpensive, and not biased.

The solution to both problems would be to remove the human actor and replace it with an artificially intelligent model. The AI model is capable to monitor and analyze the candidate behavior during a virtual mock interview; generate an analytical report that matches what would be generated from a nonbiased HR specialist. This is the goal of our graduation project.

Virtual Mock Interview (VMI) is a web application that addresses both issues. it is built on an AI model that consists of two main components, a facial analysis model and a speech analysis model. Both models are used together to analyze the user's behavior during the mock interview. Finally, after the interview is over, a report is generated with a detailed analysis of the user behavior.

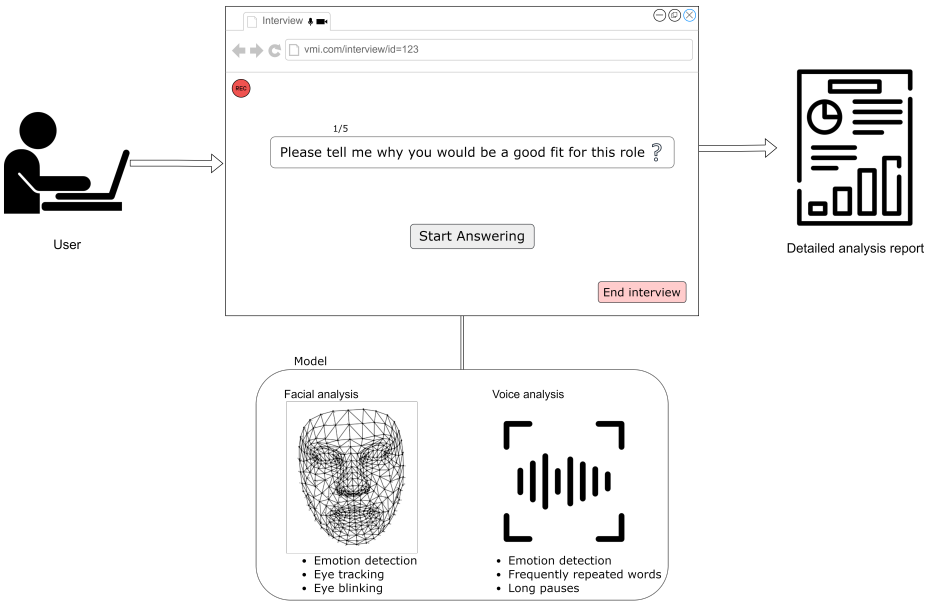

Figure 1.1: program illustration

In this report we will go through all the software project phases such as requirement gathering, market research, system design, implementation, front and back end and the overall general algorithm.

Briefly, chapter 2 shows how we gathered our project requirements, how we analyzed them and transforming requirements to viable designs. Then, in chapter 3 we have all our system designs, such as class diagram, system architecture, component diagrams, sequence diagrams, and others. In chapter 4 we go through how we implemented our program and some snippets of our code and what algorithms we used and their analysis. chapter 5 walks us through the front end and how each page looks and interacts with one another. Adversely, the back-end starts at chapter 6, it shows how the logic and brain of the program work behind the scenes and how everything is interconnected together. In chapter 7, we go through the motions of how the report is created and generated to obtain the information needed for the interviewee and the goal of the project. chapter 8 shows us the general algorithm of the project. Going from beginning to end on the entire user

experience with both the front and back-end working in tandum. Lastly, chapter 9 and chapter 10 talk about the conclusion and references respectively.

2 Requirement Gathering and Analysis

2.1 Domain Analysis

Domain analysis is the process by which we try to learn the background information of our software. This process is required to learn sufficient information; to be able to understand both problems and make good decisions during requirements analysis and other stages of the software engineering process. The more our domain analysis is in depth, the more we can identify our stakeholders better, communicate with users and make intelligent decisions.

We started the domain analysis by publishing a Google survey form for undergraduates, fresh graduates, and HR communities. The results for the form and our comments on them are as follows:

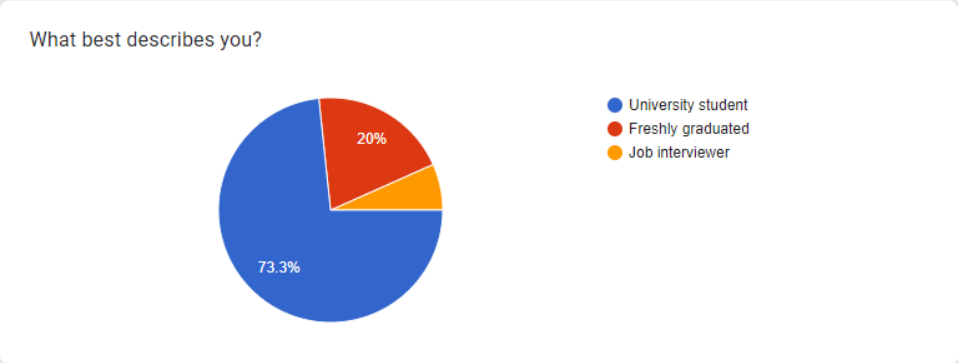

Figure 2.1: Google Form - Potential Stakeholders

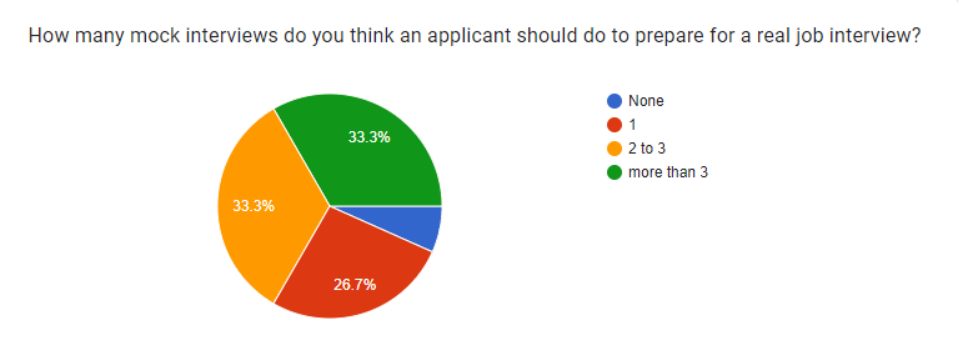

Figure 2.2: Google Form - Number of Mock Interviews

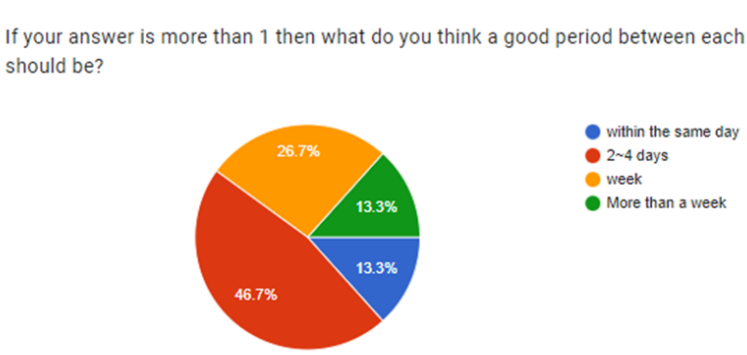

Figure 2.3: Google Form - Period between Mock Interviews

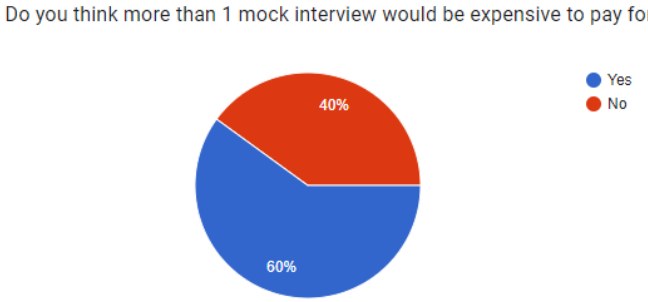

Figure 2.4: Google Form - Mock Interviews Pricing

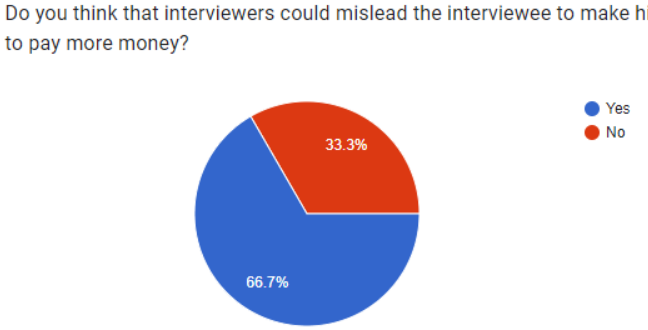

Figure 2.5: Google Form - Biased Career Counsellor

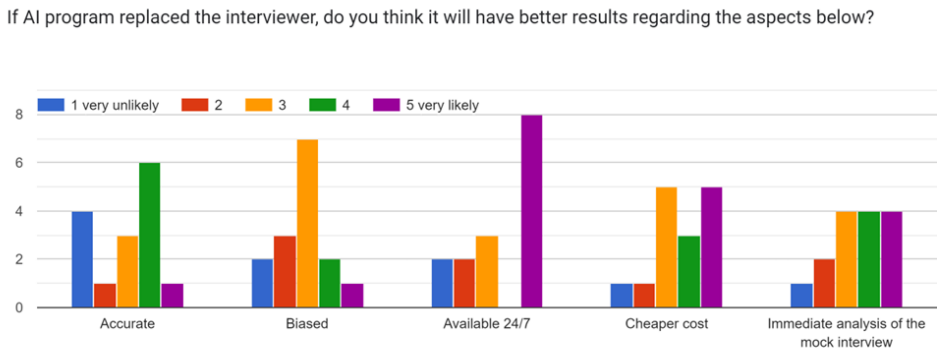

Figure 2.6: Google Form - AI vs Career Counsellor

From the figures above, we can conclude important insights and actions to be taken:

1. **insight:** We can conclude from Figure 2.1 that our insights might be biased towards undergraduates since their communities were the most interactive with the google form, however, we will try to communicate with more HRs throughout the project lifetime.

   action: Adapt agile method to be able to keep in contact with the end users, stakeholders and affected parties.

2. **insight:** We can conclude from Figure 2.2 that the average interviewee would like from 2 to more than 3 interviews to prepare for a real interview.

   action: Focus on the intricacies of the questioning and emotional analysis aspect of the interview to prepare the interviewee for the real interview.

3. **insight:** We can conclude from Figure 2.3 that the average interviewee would like to interview on average each 2 to 4 days.

   action: Ensure the capabilities of our server to handle both stress and longevity. To be able to open and use the app at any point without cutting off or connection issues.

4. **insight:** We can conclude from Figure 2.4 that the average interviewee thinks the traditional mock interviews are expensive.

   action: Ensure a freemium model for our service so that we have an advantage over the traditional interviewing method.

5. **insight:** We can conclude from Figure 2.5 that the average interviewee thinks that the interviewer could mislead the interviewee to make more interviews for monetary gain.

   action: Ensure that the model has the least amount of bias possible with no chance for deceit or misleading the interviewees.

6. **insight:** We can conclude from Figure 2.6 that generally the prospect of an AI interviewer to be in a positive light towards that potential interviewees.

   action: Ensure the accuracy and precision of the model to get the best possible result for the interviewees.

2.2 Use Cases of our System

Generally, we can benefit from UML use case diagrams to get to know more about our system. Use case diagrams can also be used to identify and model the interactions between the system and its users, known as actors. This allows for a more thorough understanding of the system's requirements, including the goals of the actors and the use cases that the system must support.

Additionally, use case diagrams are useful for identifying and modeling the relationships between use cases and actors. This information can be used to define the system's interfaces, as well as to identify any potential conflicts or dependencies between use cases.

Overall, use case diagrams are a valuable tool in the requirements phase because they allow the project team to understand the functionality that the system must provide, the interactions between the system and its users, and the constraints and rules that the system must abide by.

2.2.1 Use Case Tables

Actor: Job seekers

Begin Interview	
Use Case Name	Begin Interview
Description	Allows the job seeker to begin a virtual mock interview
Preconditions	Job Seeker has configured the camera and chose the field
Flow of Events	1. Job Seeker clicks the "Begin Interview" button on the website 2. The website checks the Job Seeker's camera and audio settings 3. The website displays the first interview question and starts the countdown timer 4. Job Seeker answers the question or clicks "Next Question" 5. At question 5, the job seeker clicks the "End Interview" button
Postconditions	Job Seeker has completed the virtual mock interview and their performance report is generated

Table 2.1: Start Interview Use Case table

View Report	
Use Case Name	View Report
Description	Allows the job seeker to view their performance report from a previous interview
Preconditions	Job Seeker has completed a previous virtual mock interview
Flow of Events	1. Job Seeker clicks the "View Performance Report" button on the website 2. The website displays the Job Seeker's performance report, including feedback on their eye contact, pacing, and other performance metrics
Postconditions	Job Seeker has reviewed their performance report

Table 2.2: View Report Use Case table

Re-take Interview	
Use Case Name	Re-take Interview
Description	Allows the job seeker to retake a previous interview
Preconditions	Job Seeker has completed a previous virtual mock interview
Flow of Events	1. Job Seeker clicks the "Re-take Interview" button on the website 2. The website displays the first interview question and starts the countdown timer 3. Job Seeker answers the question or clicks "Next Question" 4. Steps 3-4 repeat until the Job Seeker clicks "Finish Interview" 5. A new performance report is generated for the Job Seeker
Postconditions	Job Seeker has completed a new virtual mock interview and their new performance report is generated

Table 2.3: Re-take Interview Use Case table

Actor: Recruiters

Schedule Interview	
Use Case Name	Schedule Interview
Description	Allows the recruiter to schedule a virtual mock interview for a job candidate
Preconditions	Recruiter has an account on the website and has logged in
Flow of Events	1. Recruiter clicks the "Schedule Interview" button on the website 2. Recruiter enters the job candidate's information and selects the interview questions 3. Recruiter sends the interview link to the job candidate
Postconditions	Job candidate has received the interview link and can begin the virtual mock interview

Table 2.4: Schedule Interview Use Case table

Review Performance Reports	
Use Case Name	Review Performance Reports
Description	Allows the recruiter to review the performance reports of job candidates who have completed virtual mock interviews
Preconditions	Recruiter has job candidates who have completed virtual mock interviews
Flow of Events	1. Recruiter clicks the "Review Performance Reports" button on the website 2. The website displays a list of job candidates who have completed virtual mock interviews 3. Recruiter selects a job candidate's report to review 4. The website displays the job candidate's performance report, including feedback on their eye contact, pacing, and other performance metrics
Postconditions	Job candidate has received the interview link and can begin the virtual mock interview

Table 2.5: Review Performance Reports Use Case table

2.2.2 Use Case Diagram

After illustrating each potential use case in our system for different users categories, we can create UML use case diagram that combines them in one diagram to deliver the general association between the actors and use cases

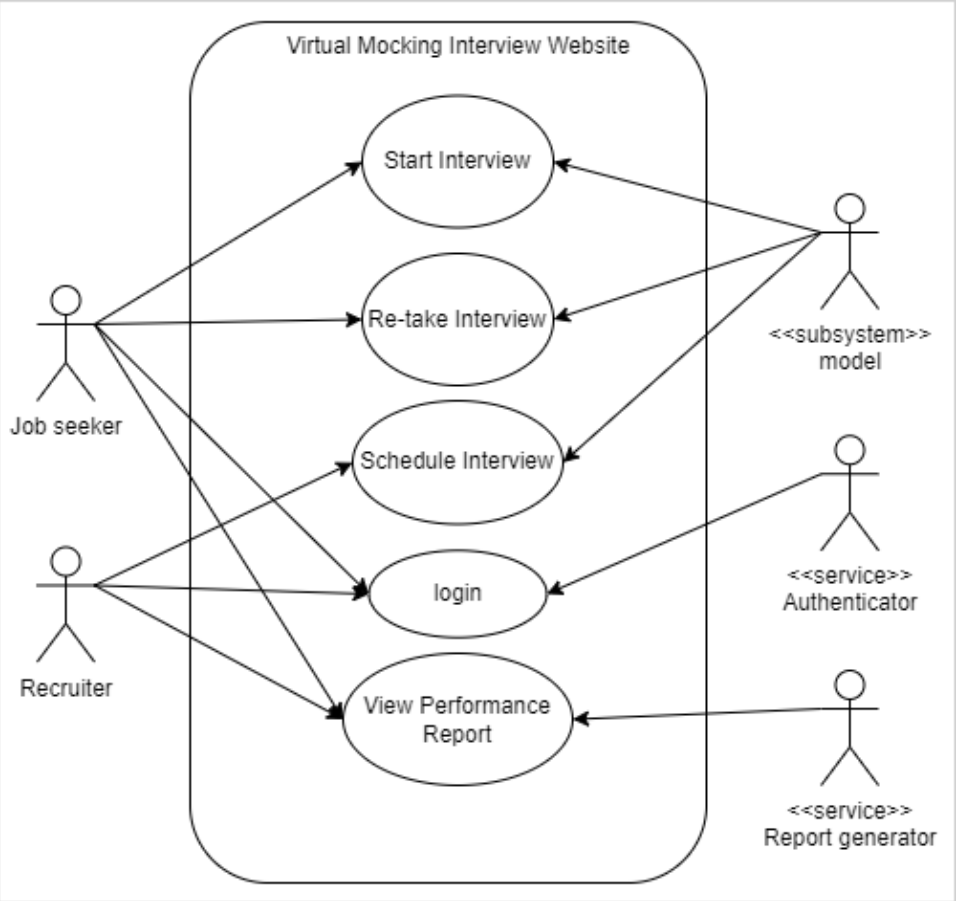

Figure 2.7: Use Case Diagram

3 Software System Design

In order to explain and show our software system design and how exactly it eliminates the human factor part and replaces it with an artificially intelligent model, we need to review what the model is capable of. The AI model is capable of monitoring and analyzing the candidate behavior during the virtual mock interview; generate an analytical report that matches what would be generated from a non biased HR specialist.

So, for an overview of the big picture before going into details where we explain the block diagram and the class diagram, sequence diagram, our web application with its user friendly UI will ask the user to start the configuration process for their camera and microphone, once they entered the website, then the next step is starting an interview in their chosen field.

During the interview, the user behavior and expressions will be recorded and analyzed. Finally, after the interview is over, a feedback report is generated with a detailed analysis of user behavior and how they can improve.

The following sub-sections will include the block diagram with its clarification, the class diagram with its clarification, and the sequence diagram with its clarification.

3.1 Block Diagram

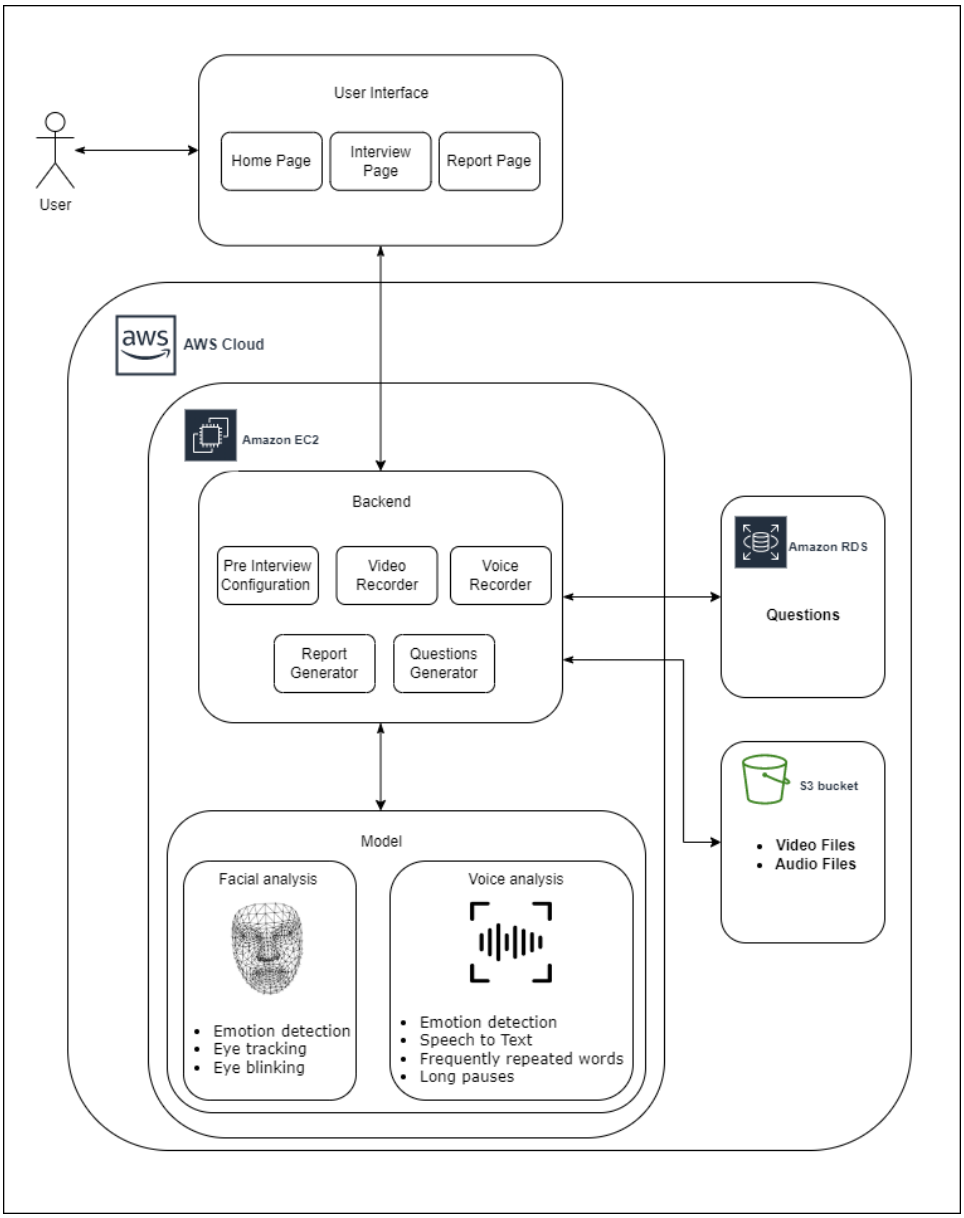

Figure 3.1: Block Diagram

Components of our system starts with the UI component. It is divided to Home page, Interview Page, and Report Page that the user can navigate through.

The UI is connected to our backend that consists of multiple components.

Pre-interview Configuration that helps the user to setup his camera and mic to get best results. Questions generator that uses questions stored in our amazon RDS. Video and Voice recorders that are used by our model, Report generator that takes the output of the model and generate a detailed report with visuals.

The Backend is connected to our model which consists of two main components. Facial analysis model that takes video records either in real time or videos stored in S3 bucket, the model utilizes emotion detection, eye tracking, eye blinking, and other facial characteristics. The second model is Voice analysis model, it utilizes emotions through voice, speech to text, frequently repeated words, and long pauses.

The whole backend servers, models, and databases are hosted in AWS cloud services for problem free experience to the user.

3.2 Class Diagram

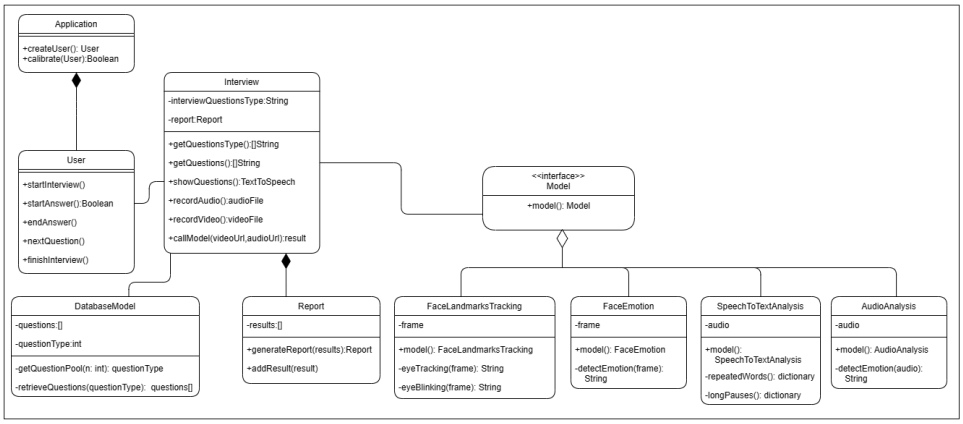

Figure 3.2: Class Diagram

Attached is the initial class diagram for our application, we have an Application class that creates a user and calibrates them. We have User class associated with the Interview class. Our interview class gets the questions from database manager and publish it to the user, it also records the user's video and voice and sends them to their corresponding models. These models implement the model interface, which ensure that the main functionality for each one of them is implemented, and provide the details required to generate the report in the Report class later.

3.3 Sequence Diagram

In this section, we will describe the design of the virtual mock interview website. To ensure that the system meets the functional requirements identified in the previous section, we will use sequence diagrams to describe the interactions between objects and components in the system.

A sequence diagram is a graphical representation of the interactions between objects or components in a system. It shows the sequence of message exchanges between objects or components, the order in which they occur, and how they are related.

The following sequence diagram shows the interactions that occur when a job candidate schedules an interview on the virtual mock interview website:

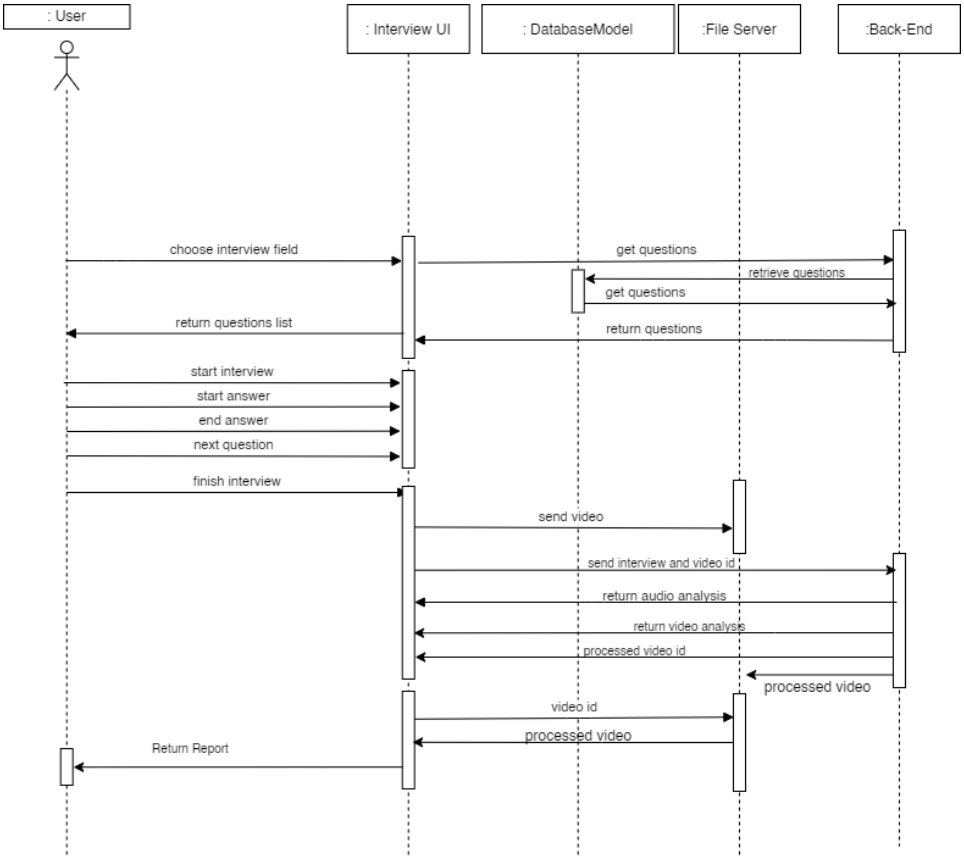

Figure 3.3: Sequence Diagram

This sequence diagram provides a detailed view of the interactions between objects and components in the system, which helps to ensure that the system is implemented correctly and meets the functional requirements.

4 Software Implementation

The virtual interview web app that has been developed utilizes a combination of cutting-edge technologies to analyze and evaluate the performance of the applicant during the interview. The app employs four distinct models, each designed to capture a specific aspect of the applicant's behavior and speech.

The first model, iris tracking using Mediapipe Facemesh uses computer vision algorithms to track the applicant's eye movements and focus during the interview. By analyzing the movement and fixations of the applicant's eyes, the model can provide insights into their level of engagement and attention, which can be an important indicator of their interest and suitability for the role. This model can help fresh graduates who may not have much interview experience to showcase their engagement and focus, providing a more complete picture of their suitability for the role.

The second model, speech to text using Whisper uses advanced speech recognition algorithms to transcribe the applicant's spoken words into written text. This allows for accurate analysis of their responses, including grammar and vocabulary. Additionally, this model supports multiple languages, allowing for a wider range of applicants to be interviewed. This feature can be useful for fresh graduates who may not be fluent in the language of the interview, providing them with an equal opportunity to showcase their skills and abilities.

The third model, speech emotion recognition using MLP classifier uses machine learning algorithms to analyze the applicant's speech and detect the emotions conveyed through their tone and pitch. By analyzing the applicant's speech patterns and vocal cues, the model can provide a deeper understanding of their state of mind during the interview, which can be useful for evaluating their fit for the role and for the company's culture. For fresh graduates who may not have developed the skills to control their emotions during an interview, this model can help to showcase their ability to remain composed and professional under pressure.

Finally, the fourth model, facial emotion detection using Keras uses deep learning algorithms to analyze the applicant's facial expressions and detect the emotions they are feeling during the interview. By analyzing the applicant's facial muscles and movements, the model can provide a complementary perspective to the speech analysis, helping to give a more complete picture of the applicant's emotional state during the interview. This can be especially useful for fresh graduates who may not have developed the ability to conceal their emotions during an interview, providing a more complete picture of their professionalism.

By using these multiple models, the virtual interview web app can provide an in-depth analysis of an applicant's performance during the interview, giving hiring managers a deeper understanding of the candidate's strengths and weaknesses, and enabling them to make more informed hiring decisions. This can be especially beneficial for fresh graduates who may not have the professional experience to make a strong impression during an interview, providing them with the opportunity to showcase their skills and abilities.

4.1 Facial Model

In this subsection, we will be discussing the software implementation of the facial model feature in our virtual interview application. This feature utilizes the combination of the OpenCV library and the MediaPipe FaceMesh solution to capture real-time video footage of the applicant and detect facial landmarks.

The iris position, iris ratio, and nose-to-eye ratio are calculated using a custom function called 'calculateRatios()' and are displayed on the video feed for the user to observe. This feature enables real-time tracking of the applicant's gaze, providing valuable insights into their focus and engagement during the virtual interview. This implementation of iris tracking, now part of our facial model, offers an accurate and efficient approach to gather data on the applicant's engagement throughout the interview.

By leveraging the iris tracking feature, we can enhance the accuracy of the overall evaluation of the applicant's performance. This data-driven approach enables us to make more informed assessments. In this subsection, we will provide a detailed explanation of the code used to implement this feature, including the key functions, algorithms, and libraries employed.

To give you an idea of how the iris tracking appears in action, refer to the example image below:

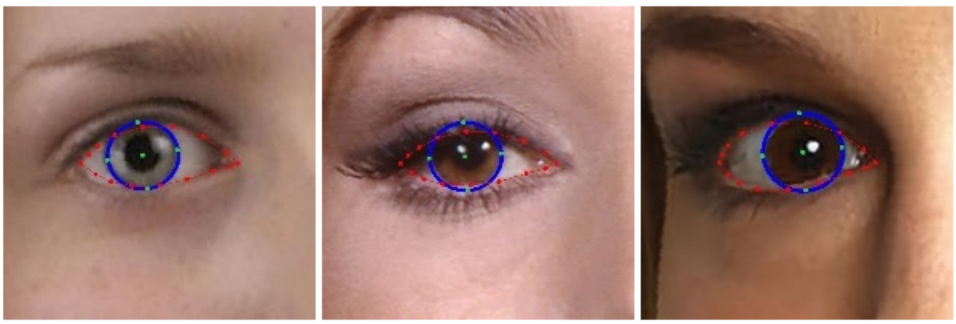

Figure 4.1: Iris tracking example

```
class FacialModel:

    '''
    A class for facial feature extraction and analysis.

    Attributes:
    - config: A ConfigParser object that reads the configuration
        file 'config.ini'
    - LEFT_EYE, RIGHT_EYE, RIGHT_IRIS, LEFT_IRIS, L_H_LEFT,
        L_H_RIGHT, R_H_LEFT,
      R_H_RIGHT, MID_EYES: Indexes for facial landmarks, read from
          the 'FACE_INDEXES'
      section of the configuration file.
    - IMG_H, IMG_W: Height and width of images, read from the
        'IMG_SIZE' section of
      the configuration file.
    - mp_face_mesh: A mediapipe face mesh object.
    - emotion_model: A FacialEmotionAnalysis object for facial
        emotion analysis.
    - DEBUG: A boolean value indicating whether debug mode is
        enabled or not.
    '''

    def __init__(self):
        # read config sections
        self.config = ConfigParser()
        config_path = app.config['FACIAL_CONFIG']
        self.config.read(config_path)
        self.emotion_model = fed.FacialEmotionAnalysis(model_path,
            model_weights)
        # eyes indexes in mediapipe
        self.LEFT_EYE = json.loads(self.config.get('FACE_INDEXES',
            'LEFT_EYE'))
        self.RIGHT_EYE = json.loads(self.config.get('FACE_INDEXES',
            'RIGHT_EYE'))

        self.RIGHT_IRIS = json.loads(self.config.get('FACE_INDEXES',
            'RIGHT_IRIS'))
        self.LEFT_IRIS = json.loads(self.config.get('FACE_INDEXES',
            'LEFT_IRIS'))

        self.L_H_LEFT = json.loads(self.config.get('FACE_INDEXES',
            'L_H_LEFT'))
        self.L_H_RIGHT = json.loads(self.config.get('FACE_INDEXES',
            'L_H_RIGHT'))

        self.R_H_LEFT = json.loads(self.config.get('FACE_INDEXES',
            'R_H_LEFT'))
        self.R_H_RIGHT = json.loads(self.config.get('FACE_INDEXES',
```

```
        'R_H_RIGHT'))
    self.MID_EYES = json.loads(self.config.get('FACE_INDEXES',
        'MID_EYES'))
    # load mediapipe face mesh
    self.mp_face_mesh = mp.solutions.face_mesh
    self.mp_face_detection = mp.solutions.face_detection
    self.DEBUG = False
```

The given code snippet defines a class called **FacialModel** that serves the purpose of facial feature extraction and analysis. This class encapsulates various attributes and methods to enable this functionality.

The class begins with a set of attributes. The **config** attribute is an instance of the **ConfigParser** class, which is responsible for reading the configuration file named 'config.ini'. This file contains important settings and parameters for the facial model. The **LEFT_EYE, RIGHT_EYE, RIGHT_IRIS, LEFT_IRIS, L_H_LEFT, L_H_RIGHT, R_H_LEFT, R_H_RIGHT**, and **MID_EYES** attributes represent indexes for different facial landmarks. These index values are obtained from the **'FACE_INDEXES'** section of the configuration file. Additionally, the **IMG_H** and **IMG_W** attributes store the height and width of the images used for facial analysis, respectively. These values are fetched from the **'IMG_SIZE'** section of the configuration file. The **mp_face_mesh** attribute holds an instance of the **MediaPipe** face mesh solution, which facilitates real-time capture of facial data.

```
def euclideanDistance(self, p1: list, p2: list):
    '''
    Calculate the Euclidean distance between two points in a
        two-dimensional space.

    Args:
    - p1: A list containing the coordinates of the first point.
    - p2: A list containing the coordinates of the second point.

    Returns:
    - dist: A float representing the Euclidean distance between
        the two points.
    '''
    # Unpack the coordinates of the two points from the input
        lists.
    x1, y1 = p1.ravel()
    x2, y2 = p2.ravel()

    # Calculate the Euclidean distance between the two points
        using the formula
    # sqrt((x2-x1)^2 + (y2-y1)^2).
    dist = math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)
```

```
        # Return the calculated distance.
        return dist
```

The given code snippet defines a method named **euclideanDistance** within the **FacialModel** class. This method calculates the Euclidean distance between two points in a two-dimensional space. The method takes two arguments: **p1** and **p2**, which are lists containing the coordinates of the first and second points, respectively.

To begin the calculation, the method first unpacks the coordinates of the two points from the input lists. The **ravel()** function is used to convert the lists into one-dimensional arrays, and then the coordinates are assigned to the variables **x1**, **y1**, **x2**, and **y2**.

Next, the method calculates the Euclidean distance between the two points using the Euclidean distance formula: $\sqrt{(x2-x1)^2+(y2-y1)^2}$. This formula computes the square root of the sum of the squared differences between the x-coordinates and y-coordinates of the two points.

The calculated distance is stored in the variable **dist**, which is of type float. Finally, the method returns the calculated distance as the output.

```
def irisPosition(self, r_iris_center, right_point, left_point,
    mid_eye, l_iris_center):
        '''
        Calculates the position of the iris within the eye (left,
            center, or right) based on the position of the iris
            center
        and the distance ratio between the nose and the eyes.

        Args:
        - r_iris_center (list): Coordinates of the right iris center
            point.
        - right_point (list): Coordinates of the right eye rightmost
            point.
        - left_point (list): Coordinates of the right eye leftmost
            point.
        - mid_eye (list): Coordinates of the midpoint between both
            eyes.
        - l_iris_center (list): Coordinates of the left iris center
            point.

        Returns:
        - A tuple containing:
            - irisPosition (str): Position of the iris within the
                eye (left, center, or right).
            - iris_ratio (float): Ratio between the distance from
```

```
            the right iris center to the rightmost eye point
        and the width of the eye.
        - nose_to_eye_ratio (float): Ratio between the distance
            from the nose to the right iris center and the
        distance between both eyes.
    '''

    # calculate disctance between the two centers of both eyes
    nose_to_right_eye_center =
        self.euclideanDistance(r_iris_center, mid_eye)
    total_eye_distance = self.euclideanDistance(r_iris_center,
        l_iris_center)
    nose_to_eye_ratio =
        nose_to_right_eye_center/total_eye_distance

    # caclulate right eye iris position
    center_to_right_dist = self.euclideanDistance(r_iris_center,
        right_point)
    eye_width = self.euclideanDistance(right_point, left_point)
    iris_ratio = center_to_right_dist/eye_width

    iris_position = ""

    # determine iris position based on the calculated ratios
    if nose_to_eye_ratio <= 0.30:
        iris_position = "right"

    elif nose_to_eye_ratio > 0.30 and nose_to_eye_ratio <= 0.70:

        if iris_ratio <= 0.40:
            iris_position = "right"
        elif iris_ratio > 0.40 and iris_ratio <= 0.60:

            iris_position = "center"
        else:
            iris_position = "left"
    else:
        iris_position = "left"

    return iris_position, iris_ratio, nose_to_eye_ratio
```

The `irisPosition` method calculates the position of the iris within the eye (left, center, or right) based on the position of the iris center and the distance ratio between the nose and the eyes. It takes the following arguments: `r_iris_center` (`list`), `right_point` (`list`), `left_point` (`list`), `mid_eye` (`list`), and `l_iris_center` (`list`).

The method begins by calculating the distance between the right iris center

and the midpoint between both eyes (`nose_to_right_eye_center`) using the `euclideanDistance` method. Similarly, it calculates the total distance between the two iris centers (`total_eye_distance`). The `nose_to_eye_ratio` is obtained by dividing `nose_to_right_eye_center` by `total_eye_distance`.

Next, the method computes the distance between the right iris center and the rightmost point of the right eye (`center_to_right_dist`) using the `euclideanDistance` method. It also calculates the width of the eye by measuring the distance between the rightmost and leftmost points of the right eye (`eye_width`).

The `iris_position` variable is initialized as an empty string. The method then proceeds to determine the position of the iris within the eye based on the calculated ratios. It checks the following conditions:

If `nose_to_eye_ratio` is less than or equal to 0.30, the `iris_position` is set to "right".

If `nose_to_eye_ratio` is greater than 0.30 and less than or equal to 0.70, further conditions are checked based on the `iris_ratio`:

If `iris_ratio` is less than or equal to 0.40, the `iris_position` is set to "right".

If `iris_ratio` is greater than 0.40 and less than or equal to 0.60, the `iris_position` is set to "center".

Otherwise, the `iris_position` is set to "left".

If none of the above conditions are met, the `iris_position` is set to "left".

Finally, the method returns a tuple containing the `iris_position`, `iris_ratio`, and `nose_to_eye_ratio`.

```
def facialAnalysis(self, frame, oldEmotion, oldEmotionProb,
    oldEnergy, oldEnergyProb, frameCounter, fps, DEBUG= False):
        '''
        Detects and analyse person's facial landmarks in real-time
            using the computer's webcam.

        Args:
            - frame: cv2 frame to be analyzed.
            - oldEmotion: previous emotion detected.
            - oldEmotionProb: previous emotion probability.
            - oldEnergy: previous energy detected.
            - oldEnergyProb: previous energy probability.
            - frameCounter: frame counter.
            - fps: frames per second.
```

```
        -DEBUG (bool, optional): A flag indicating whether to
            enable debug mode, which displays additional
            visualizations of the process. Defaults to False.

    Returns:
        tuple: A tuple containing:
            - emotion (str): The emotion detected in the current
                frame.
            - emotion_prob (float): The probability of the
                detected emotion.
            - energy (str): The energy detected in the current
                frame.
            - energy_prob (float): The probability of the
                detected energy.
            - frame (cv2 frame): The frame with the detected
                facial landmarks.
    '''

    # get facial emotion analysis
    frame, facial_emotion, facial_emotion_prob, energitic,
        energy_prob = self.facialEmotionAnalysis(frame,
        oldEmotion, oldEmotionProb, oldEnergy, oldEnergyProb,
        frameCounter, fps, DEBUG)


    # get facial landmarks using mediapipe face mesh model
    with self.mp_face_mesh.FaceMesh(
        max_num_faces= 1,
        refine_landmarks = True,
        min_detection_confidence = 0.5,
        min_tracking_confidence = 0.5
    ) as face_mesh:
            rgb_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
            #get results
            results = face_mesh.process(rgb_frame)
            if results.multi_face_landmarks:

                # get keypoints of face_mesh
                mesh_points = np.array([np.multiply([p.x, p.y],
                    [rgb_frame.shape[1],
                    rgb_frame.shape[0]]).astype(int) \
                                    for p in
                                        results.multi_face_landmarks[0].landmark])
                (l_cx, l_cy), l_rad =
                    cv2.minEnclosingCircle(mesh_points[self.LEFT_IRIS])
                (r_cx, r_cy), r_rad =
                    cv2.minEnclosingCircle(mesh_points[self.RIGHT_IRIS])

                center_left = np.array([l_cx, l_cy], dtype =
```

```
    np.int32)
center_right = np.array([r_cx, r_cy], dtype =
    np.int32)
# draw circles around the iris, and the midpoint
    between both eyes
if DEBUG:
    cv2.circle(frame, center_left, int(l_rad),
        (255, 255, 0), 1, cv2.LINE_AA)
    cv2.circle(frame, center_right, int(r_rad),
        (255, 255, 0), 1, cv2.LINE_AA)
    cv2.circle(frame,
        mesh_points[self.R_H_LEFT][0], 3, (255,
        255, 0), -1, cv2.LINE_AA)
    cv2.circle(frame,
        mesh_points[self.R_H_RIGHT][0], 3, (255,
        255, 0), -1, cv2.LINE_AA)
    cv2.circle(frame,
        mesh_points[self.MID_EYES][0], 3, (255,
        255, 0), -1, cv2.LINE_AA)

# get iris position and ratios for both eyes
iris_pos, iris_ratio, nose_to_eye_ratio =
    self.irisPosition(center_right,
    mesh_points[self.R_H_RIGHT],
    mesh_points[self.R_H_LEFT],
    mesh_points[self.MID_EYES], center_left)

# display the iris position and ratios on the
    frame
if DEBUG:
    cv2.putText(
        frame,
        f"Iris pos: {iris_pos} - eye: {iris_ratio:
            0.2f} - head: {nose_to_eye_ratio:
            0.2f}",
        (30,60),
        cv2.FONT_HERSHEY_SIMPLEX,
        0.5,
         (0, 0, 0),
        6
    )
    cv2.putText(
        frame,
        f"Iris pos: {iris_pos} - eye: {iris_ratio:
            .2f} - head: {nose_to_eye_ratio:
            0.2f}",
        (30,60),
        cv2.FONT_HERSHEY_SIMPLEX,
        0.5,
```

```
                (255, 255, 255),
                1
            )
        else:
            # no face detected
            iris_pos = "Not detected"
            iris_ratio ="Not detected"
            nose_to_eye_ratio = "Not detected"

        # display fps on the frame
        if DEBUG:
            cv2.putText(
                    frame,
                    f"fps: {fps: .2f}",
                    (30, 30),
                    cv2.FONT_HERSHEY_SIMPLEX,
                    0.5,
                    (0, 0, 0),
                    6
            )
            cv2.putText(
                    frame,
                    f"fps: {fps: .2f}",
                    (30, 30),
                    cv2.FONT_HERSHEY_SIMPLEX,
                    0.5,
                    (255, 255, 255),
                    1
            )

    return iris_pos, facial_emotion, facial_emotion_prob,
        energitic, energy_prob, frame
```

The code defines a method called **facialAnalysis** within a class. This method is used to detect and analyze facial landmarks in real-time using the computer's webcam. The method takes several parameters: **frame**, which is the input frame to be analyzed, **oldEmotion** and **oldEmotionProb** which represent the previous emotion detected and its probability, **oldEnergy** and **oldEnergyProb** which represent the previous energy detected and its probability, **frameCounter** which is a counter for the frames processed, **fps** which represents the frames per second, and an optional boolean parameter **DEBUG** which indicates whether to enable debug mode or not.

The method returns a tuple containing various values: **emotion** (a string) represents the emotion detected in the current frame, **emotion_prob** (a float) represents the probability of the detected emotion, **energy** (a string) represents the energy detected in the current frame, **energy_prob** (a float)

represents the probability of the detected energy, and **frame** (a cv2 frame) represents the frame with the detected facial landmarks.

The method starts by calling another method **facialEmotionAnalysis** to perform facial emotion analysis on the input frame. This method returns the modified frame along with the facial emotion, facial emotion probability, energetic state, and energy probability. These values are assigned to variables **frame**, **facial_emotion**, **facial_emotion_prob**, **energetic**, and **energy_prob**, respectively.

Then, the method uses the **mediapipe** library to get the facial landmarks using the **FaceMesh** model. It creates an instance of the **FaceMesh** class with specific parameters such as **max_num_faces**, **refine_landmarks**, **min_detection_confidence**, and **min_tracking_confidence**. The RGB version of the frame is obtained using **cv2.cvtColor**. The **process** method of the **FaceMesh** instance is called on the RGB frame to get the results.

If there are multiple face landmarks detected in the results, the method proceeds to extract the keypoints of the first face (**results.multi_face_landmarks[0]**). The iris positions and ratios are calculated using the **cv2.minEnclosingCircle** function and other specific keypoints obtained from the face landmarks. These values are stored in variables such as **l_cx**, **l_cy**, **l_rad**, **r_cx**, **r_cy**, **r_rad**, **center_left**, and **center_right**.

If the **DEBUG** flag is set to true, circles are drawn around the iris and the midpoint between both eyes using **cv2.circle**, and additional visualizations are added to the frame.

The iris position, iris ratio, and nose-to-eye ratio are calculated using another method called **irisPosition**. If the **DEBUG** flag is set, the iris position and ratios are displayed on the frame using **cv2.putText**.

If no face is detected, the iris position, iris ratio, and nose-to-eye ratio are set to "Not detected".

Finally, if the **DEBUG** flag is set, the frames per second (fps) is displayed on the frame using **cv2.putText**. The method returns the iris position, facial emotion, facial emotion probability, energetic state, energy probability, and the modified frame.

```
def facialEmotionAnalysis(self, frame, oldEmotion, oldEmotionProb,
    oldEnergy, oldEnergyProb, frameCounter, fps, DEBUG = False):
        '''
        Analyzes the facial emotion of the given frame using the
            pre-trained EmotionModel and mediapipe FaceDetection
            model.
```

```
Args:
    -frame: cv2 frame to be analyzed for facial emotion.
    -oldEmotion: The previous emotion detected in the video.
    -oldEmotionProb: The probability of the previous emotion
        detected in the video.
    -oldEnergy: The previous energy detected in the video.
    -oldEnergyProb: The probability of the previous energy
        detected in the video.
    -frameCounter: The current frame number.
    -fps: The current frames per second.
    -DEBUG: Boolean value to determine whether or not to
        display debug information on the frame.

Returns:
    tuple: A tuple containing:
        -frame: The frame with the facial emotion analysis
            displayed on it.
        -emotion: The emotion detected in the frame.
        -emotionProb: The probability of the emotion detected
            in the frame.
        -energy: The energy detected in the frame.
        -energyProb: The probability of the energy detected
            in the frame.
'''
# get a copy of the original frame
orignal_frame = frame.copy()
# convert the frame to grayscale
gray_fr = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

# detect faces in the frame using the mediapipe
    FaceDetection model
with self.mp_face_detection.FaceDetection(model_selection=0,
    min_detection_confidence=0.5) as face_detection:
    # convert the frame to RGB color space for the face
        detection model
    frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
    results = face_detection.process(frame)
    # if no faces were detected, return the original frame
        and the previous emotion and energy
    if(results.detections == None):
        print("ERROR - facial_model.py: No faces detected")
        return orignal_frame, oldEmotion, oldEmotionProb,
            oldEnergy, oldEnergyProb
    # get the bounding box of the detected face
    x =
        int(results.detections[0].location_data.relative_bounding_box.xmin
        * frame.shape[1])
    y =
        int(results.detections[0].location_data.relative_bounding_box.ymin
```

```
        * frame.shape[0])
    w =
        int(results.detections[0].location_data.relative_bounding_box.width
        * frame.shape[1])
    h =
        int(results.detections[0].location_data.relative_bounding_box.height
        * frame.shape[0])

    # crop the face on the grey frame and resize it to 48x48
        --> our region of interest
    fc = gray_fr[y:y+h, x:x+w]
    try:
        roi = cv2.resize(fc, (48, 48))
    except:
        print("ERROR - facial_model.py: resizing the face
            failed")
        return orignal_frame, oldEmotion, oldEmotionProb,
            oldEnergy, oldEnergyProb

    # if the frame is not a multiple of the fps, return the
        original frame and the previous emotion and energy
    if frameCounter % (fps) != 0:
        cv2.putText(orignal_frame, '{}:
            {:0.2f}'.format(oldEnergy, oldEnergyProb), (x-2,
            y-10), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255,
            0), 2)
        cv2.rectangle(orignal_frame,(x,y),(x+w,y+h),(255,
            255, 0), 2)
        return orignal_frame, oldEmotion, oldEmotionProb,
            oldEnergy, oldEnergyProb

    # predict the emotion of the face in the region of
        interest
    try:
        newEmotion, newEmotionProb =
            self.emotion_model.predict_emotion(frame)

    except:
        print("ERROR - facial_model.py: cannot predict
            emotion")
        cv2.putText(orignal_frame, '{}:
            {:0.2f}'.format(oldEnergy, oldEnergyProb), (x-2,
            y-10), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255,
            0), 2)
        cv2.rectangle(orignal_frame,(x,y),(x+w,y+h),(255,
            255, 0), 2)
        return orignal_frame, oldEmotion, oldEmotionProb,
            oldEnergy, oldEnergyProb
```

```
            # To calculate energy probablity:
            # if the new emotion is sad, then the energy probablity
                is the probablity of sad
            # if new emotion == neutral, then the energy probablity
                = 0.5 prob of neutral
            # else: high energy --> 0.5 prob of new emotion + 0.5
                prob of the new emotion
            if newEmotion in ["sad"]:
                isEnergitic = "Not Energetic"
                energyProb = newEmotionProb * 0.3 # new energy
                    probabality is in the range of 0.0 to 0.3 --> sad
            else:
                isEnergitic = "Energetic"
                # energy prob will be 0.3 to 0.6 if the new emotion
                    is neutral, and 0.6 to 1.0 if the new emotion is
                    other than neutral and sad
                energyProb = ((newEmotionProb * 0.3) + 0.3) if
                    newEmotion in ['neutral'] else ((newEmotionProb *
                    0.4) +0.6)

            cv2.putText(orignal_frame, '{}:
                {:0.2f}'.format(isEnergitic, energyProb), (x-2,
                y-10), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 255, 0), 2)
            cv2.rectangle(orignal_frame,(x,y),(x+w,y+h),(255, 255,
                0), 2)
        return orignal_frame, newEmotion, newEmotionProb,
            isEnergitic, energyProb
```

The given code defines a method called **facialEmotionAnalysis** within a class. This method is used to analyze the facial emotion of a given frame using the pre-trained **EmotionModel** and **mediapipe FaceDetection** model. It takes several parameters: **frame**, which is the input frame to be analyzed for facial emotion, **oldEmotion** and **oldEmotionProb** representing the previous emotion detected in the video and its probability, **oldEnergy** and **oldEnergyProb** representing the previous energy detected in the video and its probability, **frameCounter** representing the current frame number, **fps** representing the current frames per second, and an optional boolean parameter **DEBUG** which determines whether or not to display debug information on the frame.

The method returns a tuple containing various values: **frame**, which is the modified frame with the facial emotion analysis displayed on it, **emotion** representing the emotion detected in the frame, **emotionProb** representing the probability of the emotion detected in the frame, **energy** representing the energy detected in the frame, and **energyProb** representing the probability of the energy detected in the frame.

The method begins by making a copy of the original frame and converting it to grayscale using **cv2.cvtColor**. Then, it utilizes the **mediapipe FaceDetection** model to detect faces in the frame. The frame is converted to the RGB color space, and the **process** method of the **mediapipe FaceDetection** instance is called on the frame to obtain the detection results.

If no faces are detected in the results, the method prints an error message and returns the original frame along with the previous emotion and energy. Otherwise, it extracts the bounding box coordinates of the detected face from the results and crops the face region from the grayscale frame. The cropped region is resized to 48x48 pixels as the region of interest.

The method then checks if the frame number is a multiple of the frames per second (fps). If not, it adds the previous energy and its probability to the original frame and returns it along with the previous emotion and energy.

Next, the method uses the **emotion_model** to predict the emotion of the face within the region of interest. If the prediction fails, an error message is printed, and the original frame with the previous emotion and energy is returned.

To calculate the energy probability, the method checks the predicted emotion. If the emotion is "sad," the energetic state is set as "Not Energetic," and the energy probability is calculated as 30

Finally, the energetic state and energy probability are added to the original frame, along with the bounding box around the detected face. The method returns the modified frame, the new emotion, the new emotion's probability, the energetic state, and the energy probability.

4.2 Audio Pre-processing

The code is focused on audio processing and segmentation. It provides functions to slice an audio file into smaller segments based on silence and speech intervals. This segmentation allows for further analysis and processing of individual speech segments within the audio file.

The `sliceAudioFile` function takes an audio file path as input and performs the following steps:

1. Loads the audio file as a numpy ndarray using the `librosa` library.
2. Splits the audio file into segments based on a specified silence threshold.
3. Converts the frame-based segmentation into a list of time-based stamps.

4. Identifies and collects the silent time stamps and speech time stamps.

5. Returns the original audio data, sample rate, silent time stamps, and speech time stamps.

Additionally, the code provides the `slicingForEmotionDetection` function, which takes an audio file path and speech time stamps as input. It further processes the speech segments identified in the audio file and divides them into smaller segments of a fixed duration (5 seconds in this case). This function returns the divided audio segments and the sample rate.

These functions can be used as building blocks for various audio analysis tasks, such as emotion detection, feature extraction, or further processing of specific speech segments within an audio file.

```
TIME_THRESHOLD = 5
SILENCE_THRESHOLD = 20

def sliceAudioFile(AudioFilePath):

    silentStamps = []
    speechStamps = []
    # load audio file as an numpy ndarray using librosa
    try:
        x, sr = librosa.load(AudioFilePath, mono = True)
    except:
        print("ERROR: failed to load audio file")
        return [], []
    # split the audio file into segments of audio
    xt = librosa.effects.split(y= x,top_db = SILENCE_THRESHOLD)
    # add [0, 0] to the beginging of the list
    xt = np.insert(xt, 0, [0, 0], axis=0)
    # create a list of time instead of frames
    timeStamps = []
    for i in range(len(xt)):
        timeStamps.append([xt[i][0]/sr, xt[i][1]/sr])
    # if no time stamps then return the whole audio file
    if len(timeStamps) == 0:
        return x, sr, [], [[0.0, timeStamps[-1][1]]]

    # get the silent time stamps
    for i in range(len(timeStamps)-1):
            # add found silent time stamp to the silent time stamps
                list
            if timeStamps[i+1][0] - timeStamps[i][1] >
                TIME_THRESHOLD:
                silentStamps.append([timeStamps[i][1],
                    timeStamps[i+1][0]])
```

```
    # if no silent time stamps then return the whole audio file
    if len(silentStamps) == 0:
        return x, sr, [], [[0.0, timeStamps[-1][1]]]

    # check if the first frame is silent
    if not silentStamps[0][0] == 0.0:
            speechStamps.append([0.0, silentStamps[0][0]])

    for i in range(len(silentStamps) - 1):
        speechStamps.append([silentStamps[i][1],
            silentStamps[i+1][0]])
    # check if the last frame is silent
    if not silentStamps[-1][1] == timeStamps[-1][1]:
        speechStamps.append([silentStamps[-1][1], timeStamps[-1][1]])

    return x, sr, silentStamps, speechStamps
```

`sliceAudioFile` function performs audio preprocessing by slicing an audio file into silent and speech segments based on specified thresholds. It takes the following argument: `AudioFilePath` (a string representing the file path of the audio file).

The function initializes two empty lists, `silentStamps` and `speechStamps`, to store the timestamps of silent and speech segments, respectively.

Using the `librosa` library, the audio file is loaded as a numpy ndarray (`x`) with a specified sampling rate (`sr`). If an error occurs during audio file loading, an error message is printed, and empty lists are returned.

Next, the audio file is split into segments of audio based on the silence threshold (`SILENCE_THRESHOLD`) using the `librosa.effects.split` function. The resulting segments are stored in the `xt` variable.

To ensure proper segmentation, a `[0, 0]` segment is added at the beginning of the list of segments using `np.insert`.

A list of time stamps is created to represent the start and end times of each segment. Each frame in `xt` is converted from frame index to time by dividing the frame index by the sampling rate (`sr`).

If no time stamps are obtained, indicating no segments were found, the original audio file (`x`) and its sampling rate (`sr`) are returned along with empty lists for silent and speech segments.

Next, the function identifies silent time stamps by comparing the end time of one segment with the start time of the next segment. If the time difference

exceeds the time threshold (`TIME_THRESHOLD`), indicating a silent period, the time stamp is added to the `silentStamps` list.

If no silent time stamps are obtained, the function returns the original audio file (`x`) and its sampling rate (`sr`) along with empty lists for silent and speech segments.

The function then checks if the first frame is silent. If it is not, a speech segment from the start of the audio file to the beginning of the first silent segment is added to the `speechStamps` list.

For each pair of consecutive silent segments, a speech segment between them is added to the `speechStamps` list.

Finally, the function checks if the last frame is silent. If it is not, a speech segment from the end of the last silent segment to the end of the audio file is added to the `speechStamps` list.

The function returns the original audio file (`x`), its sampling rate (`sr`), the list of silent time stamps (`silentStamps`), and the list of speech time stamps (`speechStamps`).

```
def slicingForEmotionDetection(audio_path, speechTimeStamps):
        myfile = sf.SoundFile(audio_path)
        samplerate = myfile.samplerate
        audioFiles = []
        isAtSegmentEnd = False
        for i in range(len(speechTimeStamps)):
            # divide segment to 5 seconds small segments
            start = speechTimeStamps[i][0]
            end = speechTimeStamps[i][1]
            if end - start > TIME_THRESHOLD:
                while end - start > TIME_THRESHOLD:
                    myfile.seek(int(start))
                    data = myfile.read(int(TIME_THRESHOLD*
                        samplerate), dtype = "float32")
                    audioFiles.append(data)
                    start += TIME_THRESHOLD
            # if speech is bigger than 2 seconds then it is worth to
                create a segment for it, else ignore it
            if end - start > 2:
                myfile.seek(int(start))
                data = myfile.read(int((end - start)*samplerate),
                    dtype = "float32")
                audioFiles.append(data)
        return audioFiles, samplerate
```

The `slicingForEmotionDetection` function is responsible for further slicing the audio segments into smaller segments of 5 seconds each, based on the specified time threshold. It takes two arguments: `audio_path` (a string representing the path to the audio file) and `speechTimeStamps` (a list containing the start and end times of speech segments).

The function begins by creating a `SoundFile` object (`myfile`) using the `sf.SoundFile` function from the `soundfile` library. It retrieves the sampling rate (`samplerate`) from the `myfile` object.

An empty list, `audioFiles`, is initialized to store the smaller audio segments.

The function then iterates through each speech segment in `speechTimeStamps`. For each segment, it calculates the start and end times (`start` and `end`) of the segment.

If the duration of the segment is longer than the specified time threshold (`TIME_THRESHOLD`), the segment is divided into smaller segments of 5 seconds each. This is achieved by repeatedly seeking to the appropriate position in the audio file (`myfile.seek`), reading the corresponding data (`myfile.read`), and appending it to the `audioFiles` list. The `start` time is incremented by the time threshold until the remaining duration is less than or equal to the time threshold.

If the duration of the segment is greater than 2 seconds, it is considered worth creating a separate segment for it. In this case, the function seeks to the start position, reads the data corresponding to the remaining duration (`myfile.read`), and appends it to the `audioFiles` list.

Finally, the function returns the list of smaller audio segments (`audioFiles`) and the sampling rate (`samplerate`).

4.3 Voice Model

This section focuses on utilizing a speech emotion recognition model to classify emotions present in speech audio. The provided code makes use of the soundfile and librosa libraries for audio feature extraction, along with the glob library to facilitate access to audio files. The code specifically utilizes the RAVDESS dataset, which includes labeled emotions in the filenames.

Initially, the code filters out emotions that are not included in the predefined set of available emotions. Subsequently, the train-test split function from the sklearn library is employed to divide the data into separate training and testing sets. The code proceeds to train a multi-layer perceptron model using the training data and evaluates its performance on the testing data, printing the accuracy of the model. Finally, the trained model is saved using

the pickle library for future use.

Overall, the code follows a clear workflow, involving data preprocessing, model training, and evaluation. It leverages established libraries and datasets to facilitate the speech emotion recognition task, and the use of pickle enables the preservation of the trained model for later utilization.

```
class VoiceModel:
    '''
    A class that uses the Whisper library to perform speech-to-text
        conversion and
    performs speech emotion analysis using a pre-trained machine
        learning model.
    '''

    def __init__(self, model="small.en"):
        '''
        Initializes the SpeechToText object.

        Args:
            model (str): language model to be used for
                speech-to-text conversion.
        '''

        # list of fillers to be checked in the model
        self.fillers = ['well', 'like', 'actually', 'basically',
            'seriously', 'literally', 'totally','right', 'umm',
            'um', 'uh', 'hmm', 'okay', 'ok', 'honestly', 'yeah',
            'yep' ]
        self.complex_fillers = ['i guess','i suppose','believe
            me','you know what i mean','i mean','you see','you
            know','at the end of the day']

        self.samplerate_soundFile = 0
        self.model_type = model
        # load the emotion recognition model
        pkl_model_path = app.config['VOICE_MODEL_PKL']

        self.emotion_model = pickle.load(open(pkl_model_path, 'rb'))
        print("INFO: voice emotion model loaded!")
```

The VoiceModel class is designed to utilize the Whisper library for performing speech-to-text conversion and conducting speech emotion analysis using a pre-trained machine learning model.

The class includes an `__init__` method that initializes the `VoiceModel` object. It takes an optional argument `model` (default value: "tiny.en"), which specifies the language model to be used for speech-to-text conversion.

The class has the following attributes:

`fillers`: A list of common fillers that are checked in the model for speech analysis. `complex_fillers`: A list of more complex fillers or phrases that are considered during the analysis. `samplerate_librosa`: The samplerate used by the Librosa library for audio processing. `samplerate_soundFile`: The samplerate of the audio file being analyzed. `model_type`: The type of language model being used for speech-to-text conversion. In the `__init__` method, the fillers and complex fillers lists are initialized. The `samplerate_librosa` and `samplerate_soundFile` attributes are set to 0 initially. The `model_type` attribute is set based on the provided `model` argument.

Additionally, the pre-trained machine learning model for speech emotion analysis is loaded using `pickle`. The path to the model is obtained from the `VOICE_MODEL_PKL` configuration variable. A message is printed to indicate that the model has been loaded successfully.

```
def audio_prepocessing(self, audio_path, DEBUG=False):
        '''
        Calculates the total duration of silences in an audio
            segment and returns the maximum duration of a single
            silence segment.

        Args:
            audio_path (str): The path to the audio file to be
                analyzed.
            DEBUG (bool, optional): If True, additional debug
                information will be printed. Defaults to False.
        Returns:
            silentTimeStamps (List): A list of timestamps of silent
                segments in the audio file.
            speechTimeStamps (List): A list of timestamps of speech
                segments in the audio file.
            audio_librosa (np.ndarray): The audio file as a numpy n
                dimensional array
            audioFiles (list of np.ndarray): The audio files list
                where each item is a numpy n dimensional array but
                with different sampling rate than librosa sampling.
        '''
        try:
            # call the audio_prepocessing slicing functions
            audio_librosa, sr_librosa, silentTimeStamps,
                speechTimeStamps = ap.sliceAudioFile(audio_path)
            self.samplerate_librosa = sr_librosa
            audioFiles, sr_soundFile =
                ap.slicingForEmotionDetection(audio_path,
                speechTimeStamps)
            self.samplerate_soundFile = sr_soundFile
```

```
        except:
            print("ERROR: audio_prepocessing, failed to slice audio
                file")
            return None, None, None, None
        if DEBUG:
            print("DEBUG: silentTimeStamps",silentTimeStamps)
            print("DEBUG: speechTimeStamps",speechTimeStamps)
            print("DEBUG: sr_librosa",sr_librosa)
            print("DEBUG: audio_librosa type ", type(audio_librosa))
            print("DEBUG: audio_librosa shape ", audio_librosa.shape)
            print("_____________________________________________")
            print("DEBUG: sr_soundFile",sr_soundFile)
            print("DEBUG: audioFiles[0] type ", type(audioFiles[0]))
            print("DEBUG: audioFiles[0] shape ", audioFiles[0].shape)
        return silentTimeStamps, speechTimeStamps, audio_librosa,
            audioFiles
```

The `audio_preprocessing` method is responsible for performing audio preprocessing tasks on an audio file. It calculates the total duration of silences in the audio segment and returns the maximum duration of a single silence segment.

The method takes two arguments: `audio_path` (a string representing the path to the audio file to be analyzed) and `DEBUG` (an optional boolean flag indicating whether additional debug information should be printed, defaulting to `False`).

The method returns the following values:

`silentTimeStamps`: A list of timestamps representing the silent segments in the audio file. `speechTimeStamps`: A list of timestamps representing the speech segments in the audio file. `audio_librosa`: The audio file represented as a NumPy n-dimensional array using the Librosa library. `audioFiles`: A list of audio files, where each item is a NumPy n-dimensional array representing a segment of the audio with a different sampling rate than Librosa. Within the method, the audio file is sliced and processed using the `sliceAudioFile` and `slicingForEmotionDetection` functions from the `ap` module. The resulting silent timestamps, speech timestamps, Librosa samplerate (`sr_librosa`), and Librosa audio array (`audio_librosa`) are assigned to the corresponding attributes of the class.

If the audio preprocessing fails at any point, an error message is printed, and `None` values are returned for all the output variables.

If the `DEBUG` flag is set to `True`, additional debug information is printed, including the silent timestamps, speech timestamps, Librosa samplerate,

Librosa audio array shape, sound file samplerate (`sr_soundFile`), and shape of the audio files in the `audioFiles` list.

Finally, the method returns the computed values: silent timestamps, speech timestamps, Librosa audio array, and the list of audio files.

```
def analyze_text(self, text, DEBUG=False):
        '''
        Analyzes the given text for the occurrence of simple and
            complex fillers and returns the counts and the most
            frequent
        simple filler.

        Args:
            text (str): The text to be analyzed.
            DEBUG (bool, optional): If True, additional debug
                information will be printed. Defaults to False.

        Returns:
            tuple: A tuple containing:
                - simple_filler_counts (Counter): A Counter object
                    containing the count of each simple filler in the
                    text.
                - complex_filler_counts (dict): A dictionary
                    containing the count of each complex filler in
                    the text.
                - most_common_simple_filler (str): The most frequent
                    simple filler in the text.
        '''
        # Pre-process text
        text = text.lower()
        words = re.findall(r'\w+', text)

        # Count occurrences of simple fillers
        simple_filler_counts = Counter(word for word in words if
            word in self.fillers)

        # Count occurrences of complex fillers
        complex_filler_counts = {}
        for filler in self.complex_fillers:
            count = text.count(filler)
            if count > 0:
                complex_filler_counts[filler] = count

        # Determine most frequent simple filler
        if simple_filler_counts:
            most_common_simple_filler =
                simple_filler_counts.most_common(1)[0][0]
        else:
```

```
        most_common_simple_filler = None
    # Print debug information
    if DEBUG:
        print("DEBUG: simple_filler_counts",simple_filler_counts)
        print("DEBUG:
            complex_filler_counts",complex_filler_counts)
        print("DEBUG:
            most_common_simple_filler",most_common_simple_filler)

    return simple_filler_counts, complex_filler_counts,
        most_common_simple_filler
```

The `analyze_text` method analyzes a given text for the occurrence of simple and complex fillers. It returns the counts of simple fillers, counts of complex fillers, and the most frequent simple filler.

The method takes two arguments: `text` (a string representing the text to be analyzed) and `DEBUG` (an optional boolean flag indicating whether additional debug information should be printed, defaulting to `False`).

The method returns a tuple containing the following values:

`simple_filler_counts`: A `Counter` object that contains the count of each simple filler in the text. `complex_filler_counts`: A dictionary that contains the count of each complex filler in the text. `most_common_simple_filler`: The most frequent simple filler in the text. In the method, the text is preprocessed by converting it to lowercase and extracting words using regular expressions. Then, the occurrences of simple fillers are counted using a `Counter` object, where each word in the text is checked against the list of simple fillers.

The occurrences of complex fillers are counted by iterating through the list of complex fillers. For each complex filler, the count is obtained by counting its occurrences in the text, and if the count is greater than zero, it is added to the `complex_filler_counts` dictionary.

Next, the most frequent simple filler is determined. If there are any simple filler counts available, the most common filler is obtained from the `simple_filler_counts` `Counter` object using the `most_common` method. Otherwise, `None` is assigned to `most_common_simple_filler`.

If the `DEBUG` flag is set to `True`, additional debug information is printed, including the simple filler counts, complex filler counts, and the most frequent simple filler.

Finally, the method returns the computed values: the counts of simple fillers,

counts of complex fillers, and the most frequent simple filler.

```
def speech_emotion_analysis(self, audioFiles, DEBUG=False):
        '''
        Analyzes the emotions conveyed in a list of audio files
            using a pre-trained emotion detection model.

        Args:
            audioFiles (list): A list of audio files of type numpy
                ndarrays.
            DEBUG (bool, optional): If True, print debug
                information. Defaults to False.

        Returns:
            emotionResults (list): A list of the predicted emotions
                for each audio file in audioFiles.
        '''
        emotionResults = []
        # Iterate through audio files
        for file in audioFiles:
            try:
                # Extract audio features
                features = self.extract_feature(file, mfcc=True,
                    chroma=True, mel=True)
                # Predict emotion using pre-trained model
                result =
                    self.emotion_model.predict(features.reshape(1,
                    -1))
                emotionResults.append(result[0])

            except:
                print('ERROR: failed to extract features and predict
                    emotion at time stamp', speechTimeStamp)
                continue
        if DEBUG:
            print("DEBUG: emotionResults",emotionResults)
        return emotionResults
```

The `speech_emotion_analysis` method performs emotion analysis on a list of audio files using a pre-trained emotion detection model.

The method takes two arguments: `audioFiles` (a list of audio files represented as numpy ndarrays) and `DEBUG` (an optional boolean flag indicating whether additional debug information should be printed, defaulting to `False`).

The method returns a list `emotionResults` that contains the predicted emotions for each audio file in `audioFiles`.

Within the method, the audio files are iterated through using a loop. For each audio file, audio features are extracted using the `extract_feature` method. These features include MFCC (Mel-frequency cepstral coefficients), chroma features, and mel features.

The extracted features are then passed to the pre-trained emotion detection model for emotion prediction. The model predicts the emotion based on the provided features, and the resulting emotion is appended to the `emotionResults` list.

If there is an error during feature extraction or emotion prediction for a particular audio file, an error message is printed, and the iteration continues to the next file.

If the `DEBUG` flag is set to `True`, additional debug information is printed, including the `emotionResults` list.

Finally, the method returns the list of predicted emotions, `emotionResults`.

```
def extract_feature(self, audio, **kwargs):
        '''
        Extract feature from audio file `file_name`
            Features supported:
                - MFCC (mfcc)
                - Chroma (chroma)
                - MEL Spectrogram Frequency (mel)
                - Contrast (contrast)
                - Tonnetz (tonnetz)
        e.g:
            - `features = extract_feature(path, mel=True, mfcc=True)`
        Args:
            - audio (numpy.ndarray): Audio data in numpy array format
            - **kwargs (bool): Keyword arguments for different
                feature types

        Returns:
            - numpy.ndarray: Extracted features in numpy array format
        '''
        sample_rate = self.samplerate_soundFile
        # Get keyword arguments for different feature types
        mfcc = kwargs.get("mfcc")
        chroma = kwargs.get("chroma")
        mel = kwargs.get("mel")
        contrast = kwargs.get("contrast")
        tonnetz = kwargs.get("tonnetz")
        # setting audio to mono channel
        if audio.ndim > 1:
            X = np.mean(audio, axis = 1)
```

```
        else:
            X = audio

        if chroma or contrast:
            stft = np.abs(librosa.stft(X, n_fft = 512))
            features = np.array([])
        if mfcc:
            mfccs = np.mean(librosa.feature.mfcc(y=X,
                sr=sample_rate, n_mfcc=40).T, axis=0)
            features = np.hstack((features, mfccs))
        if chroma:
            chroma = np.mean(librosa.feature.chroma_stft(S=stft,
                sr=sample_rate).T,axis=0)
            features = np.hstack((features, chroma))
        if mel:
            mel = np.mean(librosa.feature.melspectrogram(y=X,
                sr=sample_rate).T,axis=0)
            features = np.hstack((features, mel))
        if contrast:
            contrast =
                np.mean(librosa.feature.spectral_contrast(S=stft,
                sr=sample_rate).T,axis=0)
            features = np.hstack((features, contrast))
        if tonnetz:
            tonnetz =
                np.mean(librosa.feature.tonnetz(y=librosa.effects.harmonic(X),
                sr=sample_rate).T,axis=0)
            features = np.hstack((features, tonnetz))
        return features
```

The `extract_feature` method is responsible for extracting audio features from an audio file. It supports various types of features, including MFCC (Mel-frequency cepstral coefficients), chroma features, MEL spectrogram frequency, contrast, and tonnetz.

The method takes an `audio` parameter, which represents the audio data in numpy array format, and keyword arguments (`**kwargs`) for specifying the desired feature types. For example, `extract_feature(audio, mel=True, mfcc=True)` extracts both MEL and MFCC features from the audio.

The sample rate for the audio is obtained from the `samplerate_soundFile` attribute of the class.

The method starts by converting the audio to mono channel if it has more than one dimension. This is done by taking the mean across channels.

Then, depending on the specified feature types, the method calculates and

concatenates the corresponding features. For features like chroma and contrast, the Short-Time Fourier Transform (STFT) of the audio is calculated using the librosa library. The features are computed using functions provided by librosa, such as `librosa.feature.mfcc()` for MFCC features, `librosa.feature.chroma_stft()` for chroma features, and `librosa.feature.melspectrogram()` for MEL features.

The extracted features are stored in a numpy array, and at the end of the method, this array is returned as the extracted features.

Each feature type is conditioned by its corresponding keyword argument (`mfcc`, `chroma`, `mel`, `contrast`, `tonnetz`). If a feature type is specified, the corresponding features are calculated and concatenated to the feature array. If a feature type is not specified, it is skipped.

The method utilizes librosa library functions and operations on numpy arrays to extract the desired audio features efficiently.

```
def voiceModel(self, video_path, DEBUG=False):

        #create parallel subprocess to execute whisper model
        # go to user directory and run whisper model
        try:
            process = subprocess.call('whisper {}.wav --model {}
                --language en --output_dir {} '.format(video_path,









        except Exception as e:
            print("ERROR: failed to run whisper model")
            print(e)
            return

        silentTimeStamps, speechTimeStamps, audioFiles =
            self.audio_prepocessing("{}.wav".format(video_path),

        # emotion analysis
        voice_emotions, voice_tone =
            self.speech_emotion_analysis(audioFiles, DEBUG)

        # read text file
        with open('{}.txt'.format(video_path), 'r') as f:
            text = f.read()
        # analyze text
```

```
simpleFillerDictionary, complexFillerDictionary,
    mostCommonSimpleFiller = self.analyze_text(text, DEBUG)
highlightedText = text
# lower all text
highlightedText = highlightedText.lower()
# highlight the filler words in the text
for word in self.fillers:
    pattern = r"\b{}\b".format(word)
    matches = re.findall(pattern, highlightedText)
    if matches:
        for match in matches:
            highlightedText = re.sub(pattern,
                '<span>{}</span>'.format(match),
                highlightedText)
for cmplx in self.complex_fillers:
    pattern = r"\b{}\b".format(word)
    matches = re.findall(pattern, highlightedText)
    if matches:
        for match in matches:
            highlightedText = re.sub(pattern,
                '<span>{}</span>'.format(match),
                highlightedText)



# <span style="color: green; text-decoration:
    underline;">{}</span>
# Open the VTT file and read its contents
with open('{}.vtt'.format(video_path), 'r') as f:
    vtt_contents = f.read()
# lower all text
vtt_contents = vtt_contents.lower()
for word in self.fillers:
    pattern = r"\b{}\b".format(word)
    matches = re.findall(pattern, vtt_contents)
    if matches:
        for match in matches:
            vtt_contents = re.sub(pattern, '<span
                style="color: green; text-decoration:
                underline;">{}</span>'.format(match),
                vtt_contents)
for cmplx in self.complex_fillers:
    pattern = r"\b{}\b".format(word)
    matches = re.findall(pattern, vtt_contents)
    if matches:
        for match in matches:
            vtt_contents = re.sub(pattern, '<span
                style="color: green; text-decoration:
                underline;">{}</span>'.format(match),
```

```
                vtt_contents)
    # Write the modified contents to the VTT file
    with open('{}.vtt'.format(video_path), 'w') as f:
        f.write(vtt_contents)
    # return all the analyzed information
    return {   "silentTimeStamps":silentTimeStamps,
               "speechTimeStamps":speechTimeStamps,
               "text":text,
               "highlightedText": highlightedText,
               "simpleFillerDictionary":simpleFillerDictionary,
               "complexFillerDictionary":complexFillerDictionary,
               "mostCommonSimpleFiller":mostCommonSimpleFiller,
               "voice_emotions":voice_emotions,
               "voice_tone":voice_tone
           }
```

The `voiceModel` method represents the main functionality of the VoiceModel class. It performs various operations on audio and text data to analyze speech patterns, emotions, and fillers in a given video.

The method takes the video_id as input and an optional DEBUG flag for enabling debug information.

First, the method executes the Whisper model using a subprocess to perform speech-to-text conversion on the audio of the video. The Whisper model is called with the appropriate arguments, including the audio file path, the selected model type, and the language.

Next, the method calls the `audio_preprocessing` function to process the audio file. It retrieves the timestamps of silent and speech segments, as well as the audio data itself.

After that, the method performs speech emotion analysis using the `speech_emotion_analysis` function. It passes the extracted audio files as input and obtains a list of predicted emotions for each audio segment.

The method reads the text file associated with the video and analyzes it for the occurrence of simple and complex fillers using the `analyze_text` function. It returns a dictionary with the counts of simple and complex fillers, as well as the most frequent simple filler.

Next, the method reads the contents of the VTT file associated with the video and highlights the filler words by modifying the HTML tags. It replaces the filler words and complex fillers with HTML span tags that apply red color and underline to the text.

Finally, the modified VTT contents are written back to the VTT file, and the method returns the analyzed information, including the silent timestamps, speech timestamps, text, simple filler dictionary, complex filler dictionary, most common simple filler, and the list of predicted emotions.

The method combines various functionalities, such as speech-to-text conversion, audio preprocessing, emotion analysis, filler analysis, and VTT file modification, to provide a comprehensive analysis of the given video's speech characteristics.

4.4 Facial Emotional Detection

This code snippet presents an implementation of facial emotion analysis using a Convolutional Neural Network (CNN) in the **Keras** framework.

The code starts by importing necessary libraries, including **Pandas**, **NumPy**, and **Keras**. These libraries provide functionalities for data manipulation, numerical operations, and building neural network models, respectively.

Next, the code introduces a class called **FacialEmotionAnalysis**. This class encapsulates the functionality for facial emotion prediction. It defines a list of emotion labels, such as "Angry," "Disgust," "Fear," "Happy," "Neutral," "Sad," and "Surprise," stored in the **EMOTIONS_LIST** attribute.

The constructor method **__init__()** initializes an instance of **FacialEmotionAnalysis** by loading the pre-trained model architecture from a JSON file specified by **model_json_file**. The JSON file contains the model's architecture configuration. The code reads the file, retrieves the architecture details, and creates the model using **model_from_json()** from Keras. Then, the model's weights are loaded from the file specified by **model_weights_file** using **load_weights()** function.

The **predict_emotion()** method takes an image (**img**) as input and performs emotion prediction using the loaded model. The image is passed to the model's **predict()** function, which returns a probability distribution over the different emotion classes. The predicted emotion is determined by selecting the class with the highest probability using **np.argmax()**.

```
from tensorflow.keras.models import model_from_json
import numpy as np

import tensorflow as tf


class FacialEmotionAnalysis(object):

    EMOTIONS_LIST = ["Angry", "Disgust",
```

```
                "Fear", "Happy",
                "Neutral", "Sad",
                "Surprise"]

    def __init__(self, model_json_file, model_weights_file):
        # load model from JSON file
        with open(model_json_file, "r") as json_file:
            loaded_model_json = json_file.read()
            self.loaded_model = model_from_json(loaded_model_json)

        # load weights into the new model
        self.loaded_model.load_weights(model_weights_file)
        self.loaded_model.make_predict_function()

    def predict_emotion(self, img):
        self.preds = self.loaded_model.predict(img)
        return
            FacialEmotionAnalysis.EMOTIONS_LIST[np.argmax(self.preds)]
```

The code snippet provided defines a class called **FacialEmotionAnalysis** for performing facial emotion analysis using a pre-trained deep learning model.

The class imports the necessary libraries, including **model_from_json** from **tensorflow.keras.models** and **numpy**. It also imports **tensorflow** for compatibility reasons.

The class has a class attribute called **EMOTIONS_LIST**, which is a list of emotion labels representing different facial expressions.

The **__init__** method serves as the constructor for the class. It takes in the paths to a model's JSON file (**model_json_file**) and its corresponding weights file (**model_weights_file**). Within this method, the model architecture is loaded from the JSON file using **model_from_json**, and the weights are loaded into the model using **load_weights**. The **make_predict_function** is called to prepare the model for predictions.

The **predict_emotion** method takes an image (**img**) as input and performs emotion prediction on it using the loaded model. The image is passed through the model's **predict** function, and the predicted probabilities for each emotion are stored in **self.preds**. The predicted emotion label is determined by finding the index of the highest probability using **np.argmax**, and retrieving the corresponding label from the **EMOTIONS_LIST**.

Overall, the **FacialEmotionAnalysis** class encapsulates the functionality of loading a pre-trained facial emotion analysis model, making predictions

on input images, and returning the predicted emotion label.

5 Front-End

The "Front-End" component of the **Virtual Mock Interview** (VMI) project consists of a collection of 10 main JavaScript React component files that work together to create a seamless user interface and manage the interview process. These files collaborate to ensure a smooth and intuitive experience for the interviewees, allowing them to undergo interviews in their chosen field with ease.

The first file, **Config.js**, enables users to configure their microphone and camera settings. It also allows the user to see themselves using their camera to be able to modify or alter their positioning for the best possible output.

Next, **Example.js** serves as an interactive demonstration of the interview interface. It offers the interviewee a visual representation of how the interface will function during the actual interview. This helps familiarize them with the features and functionalities they will encounter, ensuring a smoother experience when the real interview begins.

In the context of field selection, the **Field.js** file plays a crucial role. It presents the interviewee with a list of available fields or topics from which they can choose. This ensures that the interviewee selects a specific field in which they want to be questioned and interviewed. They can't proceed to the next page unless they have chosen the interview field.

The **Footer.js** file is responsible for the footer section of the user interface. It provides additional information, links, or navigation options that enhance the overall user experience. The footer adds value to the interface by offering relevant details about VMI and offering VMI's email so that Interviewees feel free to drop us a line at any time.

Complementing the footer, the **Header.js** file handles the header section of the user interface. It typically displays the project or platform logo which serves as a navigation hub, allowing users to access the home page of the platform easily.

As users land on the website, they encounter the **Landing.js** file, which serves as the main landing page. This page typically features a prominent button that initiates the interview process of beginning and takes us to the configuration page to start configuring the microphone and camera. It provides a clear call-to-action, encouraging users to proceed with the interview.

In the **HomeMain.js** file, two images are accompanied by descriptions that explain how the website works and provide an overview of the project's aims. This section serves as the main content area of the homepage, providing introductory information to users about the platform and its objectives.

The most critical component of the front-end architecture is the **Interview.js** file. This file manages the core aspects of the interview process. It retrieves the interview questions, sends relevant information to the back-end, records video responses for each question, and forwards them to the server. Additionally, it handles session or interview IDs to ensure proper video submission for generating the interview report. The Interview.js file encapsulates the heart of the interview experience.

To handle cases where users navigate to non-existent or invalid pages, the **NotFound.js** file comes into play. It serves as a fallback page, displaying a user-friendly message indicating that the requested page was not found. This ensures a better user experience by gracefully handling unexpected page navigation scenarios.

Lastly, the **Report.js** file generates the interview report. It collects all the necessary information from the interview and compiles it into a comprehensive report. This report is then presented to the interviewee, providing them with valuable insights and feedback regarding their interview performance.

Together, these JavaScript files form the front-end infrastructure of the VMI project. They work in harmony to deliver a user-friendly interview experience, enabling field selection, question retrieval, video recording, and report generation.

5.1 Homepage

In this subsection, we will focus specifically on the homepage of the VMI project. The homepage serves as the initial point of interaction for users and provides essential information about the platform. We will discuss the key files that contribute to the homepage's functionality and appearance.

Firstly, the **Landing.js** file plays a crucial role in the homepage. It represents the main landing page that users encounter when they access the website. This file sets the tone for the entire interview process, presenting users with a visually appealing and informative introduction. It typically includes a prominent button or call-to-action that initiates the interview process, encouraging users to take the next steps.

Secondly, the **HomeMain.js** file is responsible for the primary content section of the homepage. It showcases two images accompanied by descriptions that explain the functionality and purpose of the website. This section provides users with a clear understanding of how the platform works and outlines its overall objectives. The HomeMain.js file ensures that the homepage effectively communicates the key features and benefits of the VMI project.

Additionally, the **Header.js** file contributes to the homepage's structure and user interface. Located at the top of the page, the header serves as a central navigation hub. It includes the project logo.

Lastly, the **Footer.js** file handles the footer section of the homepage. Positioned at the bottom of the page, the footer complements the overall design and enhances the user experience. It contains additional information, links, or further guidance. The Footer.js file adds a professional touch to the homepage, ensuring that users have a further guidance access to important information of the VMI project.

5.1.1 Landing.js

```
import { Link } from "react-router-dom"

const Landing = () => {

    return (
        <div className="landing" >
            <div className="intro-text">
                <h1 className="websit-title" >Virtual Mock
                    Interview</h1>
                <p>Empowering job seekers with an intelligent and
                    unbiased interview preparation platforms that
                    leverages advanced technology
                    and user-driven insights to increase their
                        chances of landing their dream job.
                </p>
                <Link to="/configuration" ><button
                    className="button-start landing-button" >Begin
                    and Explore <span
                    className="triangle"></span></button></Link>
            </div>
        </div>
    )
}

export default Landing
```

The provided code snippet represents a functional react component called **Landing** in the VMI project. This component is responsible for rendering the landing page of the website.

In the code, the component imports the **Link** component from the **react-router-dom** library, which enables navigation between different pages within the application.

Inside the **return** statement, the component's JSX (JavaScript XML) code defines the structure and content of the landing page.

The **div** element with the className **"landing"** represents the main container for the landing page. Within this container, there is another **div** element with the className **"intro-text"**.

Inside the **intro-text** div, there are three elements: an **h1** element with the className **"websit-title"** displaying the title **"Virtual Mock Interview"**, a **p** element containing a description of the platform's purpose and features, and a **Link** component wrapped around a **button** element.

The **Link** component enables users to navigate to the **"/configuration"** page when the button is clicked. This allows users to proceed to the configuration stage of the interview process.

The button itself has the className **"button-start landing-button"**, which applies styling to give it a specific appearance. Additionally, there is a **span** element with the className **"triangle"** inside the button, which serve as a decorative element/icon.

Finally, the component is exported using the **export default** statement, making it available for use in other parts of the application.

5.1.2 HomeMain.js

```
const HomeMain = () => {

    return (

        <div className="main" >
            <h2 className="main-title" >How it works:</h2>
            <div className="main-content" >
                <div>
                    <img src="/unnamed-removebg-preview.png"
                        alt="facial img" ></img>
                    <h4>Facial Analysis Model</h4>
                    <p>Analyze users reactions to each question</p>
                    <p>and his overall emotions through the
                        interview</p>
                </div>
                <div>
                    <img src="/unnamed2-removebg-preview.png"
                        alt="voice img" ></img>
                    <h4>Voice Analysis Model</h4>
                    <p>Analyze users answers, finding repetitive</p>
                    <p>words, emotion detection, and silent
                        moments</p>
```

```
            </div>
          </div>
        </div>

    )

}

export default HomeMain
```

The provided code snippet represents a functional component called **HomeMain**. This component is responsible for rendering the main content of the homepage, specifically explaining how the platform works.

Within the **return** statement, the component's JSX (JavaScript XML) code defines the structure and content of the main section of the homepage.

The main content is enclosed in a **div** element with the className **"main"**. Inside this container, there is an **h2** element with the className **"main-title"** displaying the title **"How it works:"**, which introduces the explanation of the platform's functionality.

The core content of the explanation is contained within another **div** element with the className **"main-content"**. Inside this div, there are two similar sections, each represented by a separate **div** element.

Each section includes an **img** element displaying an image related to the analysis model being described. The source of the image is specified using the **src** attribute, pointing to the respective image file.

Below each image, there is an **h4** element displaying the title of the analysis model, such as **"Facial Analysis Model"** and **"Voice Analysis Model"**. These titles provide a brief description of the analysis performed by each model.

Following the title, there are two **p** elements containing additional details about the analysis process. These paragraphs explain the specific functionalities of each analysis model, such as analyzing the user's reactions to questions, detecting emotions throughout the interview, identifying repetitive words in the user's answers, and detecting silent moments.

Finally, the component is exported using the **export default** statement, making it available for use in other parts of the application.

5.1.3 Header.js

```
const Header = () => {
    return (
        <div className="logo-container" id="page-top" >
            <div className="top-logo-menu" >
                <a className="logo-name" href="/" >VMI</a>
            </div>
        </div>
    )
}
export default Header
```

The provided code snippet represents a functional component called **Header**. This component is responsible for rendering the header section of the website.

Within the **return** statement, the component's JSX code defines the structure and content of the header section.

The header is contained within a **div** element with the className **"logo-container"**. This container serves as a wrapper for the header content.

Inside the **logo-container**, there is another **div** element with the className **"top-logo-menu"**. This div represents the top section of the header, where the logo can be placed.

Within the **top-logo-menu**, there is an **a** element with the className **"logo-name"**. This element serves as the logo for the website and displays the text "VMI". The **href** attribute is set to "/" to make the logo clickable and navigate the user to the homepage when clicked.

5.1.4 Footer.js

```
const Footer = () => {
    return (
        <div className="footer-container" >
            <div className="footer">
                <p>Feel free to drop us a line at: <a
                    className="link"
                    href="mailto:vmi@gmail.com?subject=Contact"
                    >vmi@gmail.com</a></p>
                &copy; 2023 <span>VMI</span> All Rights Reserved
            </div>
        </div>
    )
}
```

```
export default Footer
```

The provided code snippet represents a functional component called **Footer**. This component is responsible for rendering the footer section of the website.

Within the **return** statement, the component's JSX code defines the structure and content of the footer section.

The footer is contained within a **div** element with the className **"footer-container"**. This container serves as a wrapper for the footer content.

Inside the **footer-container**, there is another **div** element with the className **"footer"**. This div represents the main content of the footer.

Within the **footer**, there are two elements. The first element is a **p** element that contains text with a link. The text prompts users to "Feel free to drop us a line at:", followed by an email address "vmi@gmail.com". The email address is wrapped in an **a** element with the className **"link"**. Clicking on the email address link will open the user's default email client with a pre-filled subject of "Contact".

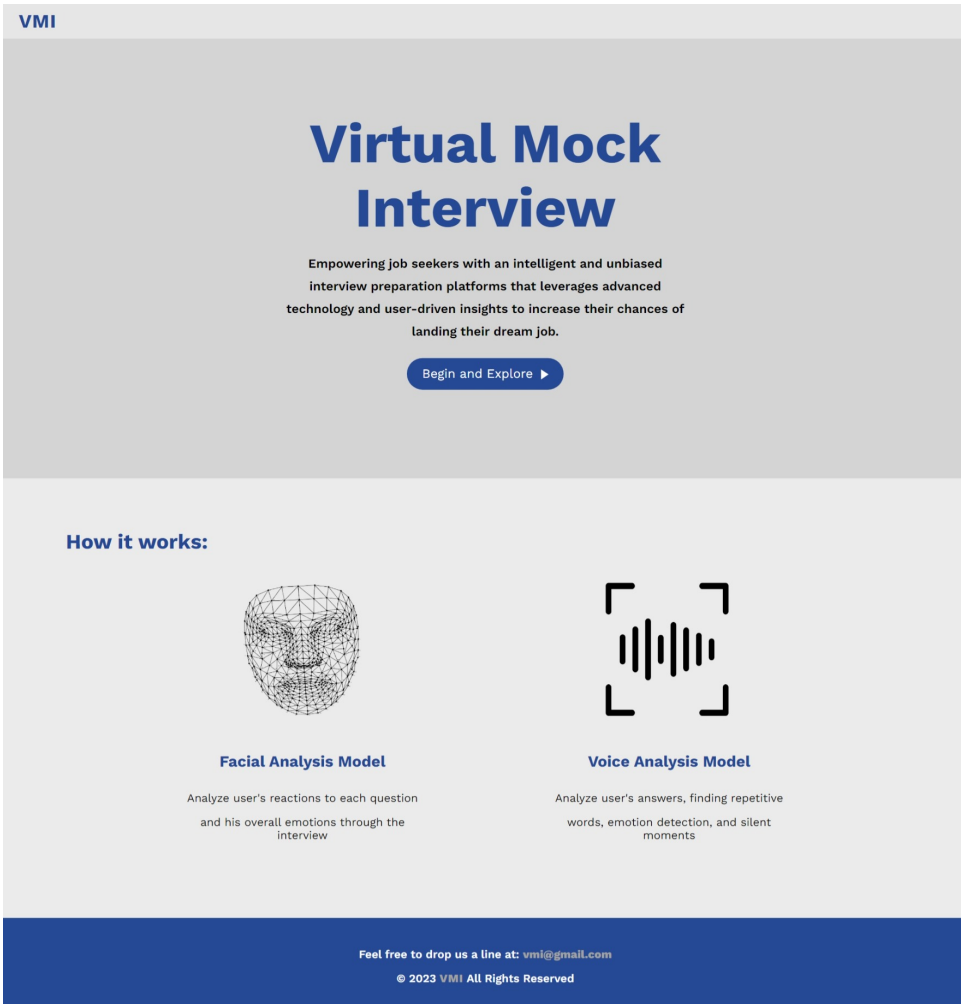

Figure 5.1: Sample Homepage

5.2 Configuration Page

As was illustrated before, the configuration page's main functionality is to allow the user or interviewee to be able to activate both their mic and video camera, so that their image may be recorded and analyzed during the interview.

They can also open their camera and see a preview of how their recording will look like. During that time, they can alter or change or adjust their positioning so that the main focus of their camera would be themselves without any outside obstructions.

5.2.1 Config.js

```
import React, { useState, useEffect, useRef} from 'react';
import { Link } from "react-router-dom"
import {useNavigate} from 'react-router-dom';
import Webcam from "react-webcam";

const Config = () => {
    const navigate = useNavigate();
    useEffect(() => {
        window.scrollTo(0, 0);
    }, []);

    const webcamRef = useRef(null);
    const audioFlag = false

    const [hasUserMediaError, setHasUserMediaError] =
        useState(false);
    const onUserMediaError = () => {
        setHasUserMediaError(true);
    }

    function checkVideo() {
        if (hasUserMediaError){
            const alertBox = document.createElement('div');
            alertBox.classList.add('custom-alert');
            alertBox.textContent = 'Please allow access for camera
                and mic permission to continue';
            document.body.appendChild(alertBox);

            setTimeout(() => {
            alertBox.remove();
            }, 2000);
        }
        else {
            navigate('/field');
        }
    }

    return (
        <>
            <div className="instructions-parent" >
                <div className="instructions-container" >
                    <h2 className="main-title-2" >How to start the
                        interview:</h2>
                    <h2 className="main-title-2" >1- Please allow
                        access for camera and mic permission:</h2>
                </div>
            </div>
```

```
            <div className="config-container">
                <div className="config-box">
                {hasUserMediaError ? (
                    <img src="/video-logo-removebg-preview.png"
                        alt="no video" className="config-img" />
                ) : (
                    <Webcam className= "web-cam" audio={audioFlag}
                        ref={webcamRef} muted={true}
                        onUserMediaError={onUserMediaError} />
                )}
                </div>
                <button className="button-start"
                    onClick={checkVideo}>Next choose your field <span
                    className="triangle"></span></button>
            </div>

        </>
    );
}
export default Config
```

The component begins by importing necessary dependencies such as React, useState, useEffect, useRef, useNavigate, and Webcam. The Webcam component is used to capture video from the user's camera.

Within the component, an **useEffect** hook is used to scroll the window to the top when the component is mounted. This ensures that the user starts at the top of the page.

the **useNavigate** hook is used to get the navigate function, which is used for programmatically navigating to a different route.

The **webcamRef** and **audioFlag** variables are declared using the **useRef** and **useState** hooks, respectively. The **webcamRef** is used to reference the Webcam component, while the **audioFlag** is set to **false** indicating that audio is not enabled.

The component also utilizes the **useState** hook to manage the **hasUserMediaError** state variable, which is initially set to **false**. The **onUserMediaError** function is defined to handle any errors that occur when accessing the user's media devices.

The **checkVideo** function is defined to handle the logic for checking the **hasUserMediaError** state variable. If **hasUserMediaError** is true, an **alertBox** element is created and appended to the body of the document, displaying a message requesting camera and microphone permissions. After

a timeout of 2000 milliseconds, the **alertBox** is removed. If **hasUserMediaError** is false, the navigate function is called to navigate to the '/field' route.

The return statement consists of JSX code that defines the structure and content of the configuration page. It includes an **instructions-parent** container that holds an **instructions-container** with headings providing instructions on how to start the interview.

Following the instructions, a div element with the className **"config-container"** is rendered. Inside the **"config-container"**, there is a div element with the className **"config-box"**. Depending on the value of **hasUserMediaError**, either an image or the Webcam component is rendered. If **hasUserMediaError** is true, an image with the source "/video-logo-removebg-preview.png" and the alt text "no video" is displayed, with the className **"config-img"**. Otherwise, the Webcam component is rendered with the className **"web-cam"**, **audioFlag** set to false, **webcamRef** assigned as the reference, and muted set to true. In case of an error while accessing the user's media, the **onUserMediaError** function is called.

Finally, a button element is rendered with the className **"button-start"**. It has an onClick event handler set to the **checkVideo** function and displays the text "Next choose your field". Additionally, there is a span element with the className **"triangle"** to represent a triangular icon.

Overall, the **Config** component sets up a configuration page where users are prompted to grant camera and microphone permissions. It handles any errors that may occur during the media access process and provides a smooth transition to the next step of choosing a field for the interview. The component is exported as the default export, making it available for use in other parts of the application.

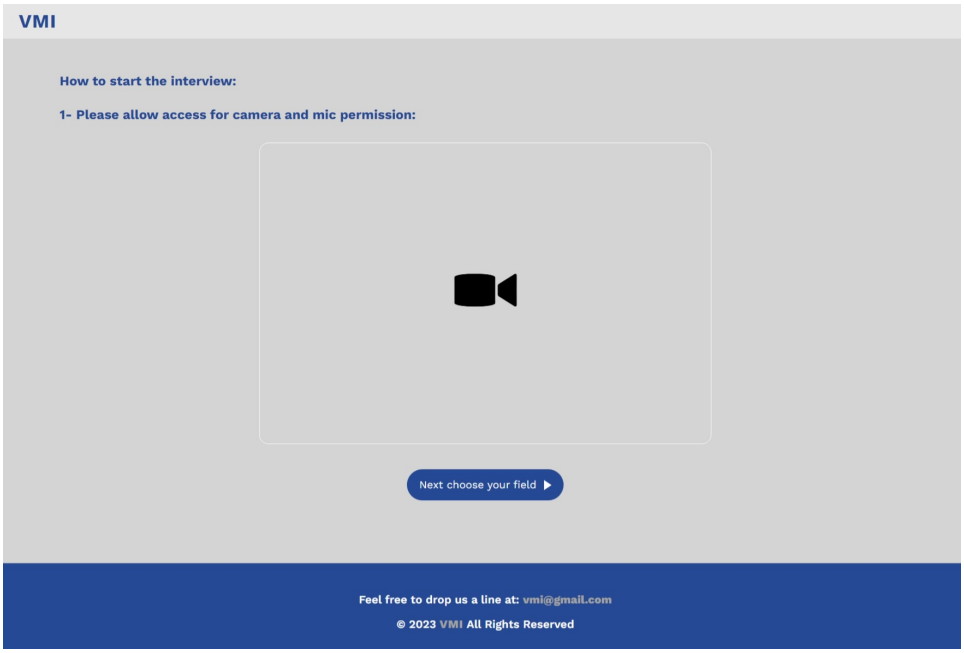

Figure 5.2: Sample Config

5.3 Field Page

5.3.1 Field.js

```
import React, { useEffect } from 'react';
import { useState } from 'react';
import {useNavigate} from 'react-router-dom';
import { useCookies } from 'react-cookie';

const Field = () => {
    const [cookies, setCookie] = useCookies(['job-field']);
    useEffect(() => {
        window.scrollTo(0, 0);
        }, []);

    const [choosenField, setChoosenField] = useState('');

    const handleButtonClick = (event) => {
        setChoosenField(event.target.id);
        console.log(event.target.id);
        };

    const navigate = useNavigate();

    const handleFieldSelect = () => {
        setCookie('job-field', choosenField, { path: '/' });
```

```
    if (choosenField === '') {
        const alertBox = document.createElement('div');
        alertBox.classList.add('custom-alert');
        alertBox.textContent = 'Please choose a field first';
        document.body.appendChild(alertBox);

        setTimeout(() => {
        alertBox.remove();
        }, 2000);
    }
    else {
        navigate('/example-case');
    }

    }
return (
    <>
        <div className='FCPP' >
            <div className="instructions-parent" >
                <div className="instructions-container" >
                    <h2 className="main-title-2" >How to start
                        the interview:</h2>
                    <h2 className="main-title-2" >2- Please
                        choose the field you want to make an
                        interview in:</h2>
                </div>
            </div>

            <div className="config-container FCP">
                <div className="field-container" >

                    <button id='Junior-Software-Engineer'
                        onClick={handleButtonClick}
                        className={choosenField ===
                        'Junior-Software-Engineer' ? 'activeField'
                        : ''}>Junior Software Engineer</button>
                    <button id='Electrical-Engineer'
                        onClick={handleButtonClick}
                        className={choosenField ===
                        'Electrical-Engineer' ? 'activeField' :
                        ''}>Electrical Engineering</button>
                    <button id='Junior-Automation-Engineer'
                        onClick={handleButtonClick}
                        className={choosenField ===
                        'Junior-Automation-Engineer' ?
                        'activeField' : ''}>Junior Automation
                        Engineer</button>
```

```
            <button id='Junior-Game-Developer'
                onClick={handleButtonClick}
                className={choosenField ===
                'Junior-Game-Developer' ? 'activeField' :
                ''}>Junior Game Developer</button>
            <button id='Junior-Frontend-Developer'
                onClick={handleButtonClick}
                className={choosenField ===
                'Junior-Frontend-Developer' ?
                'activeField' : ''}>Junior Frontend
                Developer</button>
            <button id='Junior-Mobile-Development'
                onClick={handleButtonClick}
                className={choosenField ===
                'Junior-Mobile-Development' ?
                'activeField' : ''}>Junior Mobile
                Development</button>
            <button id='Junior-Control-Engineer'
                onClick={handleButtonClick}
                className={choosenField ===
                'Junior-Control-Engineer' ? 'activeField'
                : ''}>Junior Control Engineer</button>
            <button id='Junior-Embedded-Engineer'
                onClick={handleButtonClick}
                className={choosenField ===
                'Junior-Embedded-Engineer' ? 'activeField'
                : ''}>Junior Embedded Engineer</button>
            <button id='Junior-Human-Resources'
                onClick={handleButtonClick}
                className={choosenField ===
                'Junior-Human-Resources' ? 'activeField' :
                ''}>Junior Human Resources</button>
            <button id='Junior-Project-Manager'
                onClick={handleButtonClick}
                className={choosenField ===
                'Junior-Project-Manager' ? 'activeField' :
                ''}>Junior-Project-Manager</button>
            <button id='Junior-Content-Creator'
                onClick={handleButtonClick}
                className={choosenField ===
                'Junior-Content-Creator' ? 'activeField' :
                ''}>Junior-Content-Creator</button>
            <button id='accountant'
                onClick={handleButtonClick}
                className={choosenField === 'accountant' ?
                'activeField' : ''}>accountant</button>

        </div>
```

```
                    <button className="button-start"
                        onClick={handleFieldSelect}>Next see an
                        example <span
                        className="triangle"></span></button>
                </div>
            </div>

        </>
    )
}
export default Field
```

The component starts by importing necessary dependencies such as React, useEffect, useState, useNavigate, and useCookies. These dependencies enable various functionalities within the component.

The component utilizes the **useEffect** hook to scroll the window to the top when the component is mounted, ensuring that the user starts at the top of the page.

The **Field** component uses the **useCookies** hook to manage the **cookies** state variable, which is responsible for storing the selected job field. The **setCookie** function is used to update the cookie value.

The component defines the **choosenField** state variable using the **useState** hook. This variable keeps track of the currently selected field.

The **handleButtonClick** function is defined to handle the selection of a field. It updates the **choosenField** state variable with the ID of the clicked button and logs the ID to the console.

The **navigate** function, obtained from the **useNavigate** hook, allows for programmatic navigation to different routes within the application.

The **handleFieldSelect** function is called when the user clicks the "Next see an example" button. It sets the cookie with the selected field and performs some validation. If no field is selected, it displays an alert box for a short duration to notify the interviewee to select a field first. Otherwise, it navigates to the "/example-case" route.

The JSX code within the return statement defines the structure and content of the field selection page. It includes an **instructions-parent** container that holds an **instructions-container** with headings providing instructions on how to start the interview and choose a field.

Next, a **config-container** is displayed, containing a **field-container** with

multiple buttons representing different job fields. The **choosenField** state variable is used to determine if a button should have the **activeField** class, indicating the currently selected field.

Finally, there is a button labeled "Next see an example" that triggers the **handleFieldSelect** function when clicked. It includes a triangular icon for visual enhancement.

The **Field** component provides an interface for users to select their desired job field for the interview. It manages the selected field using cookies and allows for smooth navigation to the next step, which is viewing an example interview case. The component is exported using the **export default** statement, making it available for use in other parts of the application.

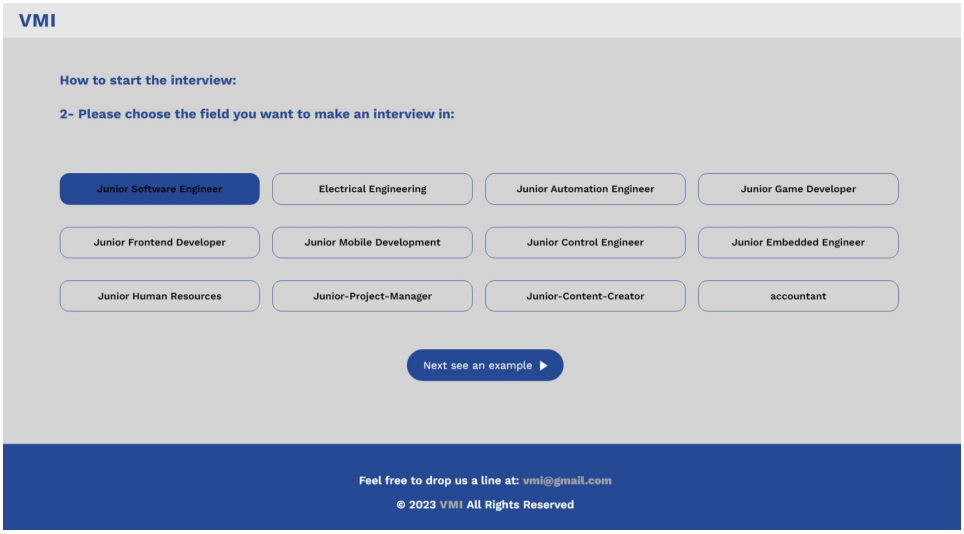

Figure 5.3: Sample Field

5.4 Example Case

5.4.1 ExampleCase.js

```
import React, { useState } from 'react'
import { useEffect } from 'react';
import { Link } from "react-router-dom"
import axios from "axios";

const ExampleCase = () => {

    useEffect(() => {
        window.scrollTo(0, 0);
        }, []);
```

```
    const [isClicked, setIsClicked] = useState(false);
    const handleClick = () => {
        setIsClicked(true);
    }

    return (
        <>
            <div className="instructions-parent" >
                <div className="instructions-container" >
                    <h2 className="main-title-2" >Example on how to
                        start the interview:</h2>
                    <h2 className="main-title-2" >3- Answer questions
                        as shown below:</h2>
                </div>

                <div className="question-container">
                    <div className="question-box">
                        <div className={`icons-holder ${isClicked ?
                            'clicked' : ''}`}>
                            <i className="fas fa-video"></i>
                            <i className="fas fa-microphone"></i>
                        </div>
                        <div className="question">Example: What is
                            your biggest strength?</div>
                        <div className='ex-start-holder' ><button
                            className="ex-start" onClick={handleClick}
                            >Start Answer </button></div>

                    </div>
                </div>

                <div className='link-container-for-example-page' >
                    <Link to="/interview" ><button
                        className="button-start" >Go to Interview
                        page <span
                        className="triangle"></span></button></Link>
                </div>
            </div>

        </>
    )
}
export default ExampleCase
```

The component starts by importing necessary dependencies such as React, useState, useEffect, Link, and axios.

The **useEffect** hook is used to scroll the window to the top when the com-

ponent is mounted, ensuring that the user starts at the top of the page.

The **ExampleCase** component defines the **isClicked** state variable using the **useState** hook. This variable keeps track of whether the user has clicked the "Start Answer" button.

The **handleClick** function is defined to handle the click event of the "Start Answer" button. It sets the **isClicked** state variable to **true**.

The JSX code within the return statement defines the structure and content of the example case page. It includes an **instructions-parent** container that holds an **instructions-container** with headings providing instructions on how to start the interview and answer questions.

Next, a **question-container** is displayed, containing a **question-box** that represents the example question. The **isClicked** state variable is used to conditionally add the **clicked** class to the **icons-holder** div, which displays video and microphone icons in the color red as an indication feedback of what will happen when the interview start.

The example question is displayed in the **question** div, and there are one button labeled "Start Answer" for the user to interact with. The **handleClick** function is called when the "Start Answer" button is clicked.

Finally, there is a **link-container-for-example-page** that includes a **Link** component, allowing the user to navigate to the "/interview" route by clicking the "Go to Interview page" button.

The **ExampleCase** component provides an example question and guides the user on how to start answering questions. It incorporates interactivity by tracking the user's click on the "Start Answer" button. The component is exported using the **export default** statement, making it available for use in other parts of the application.

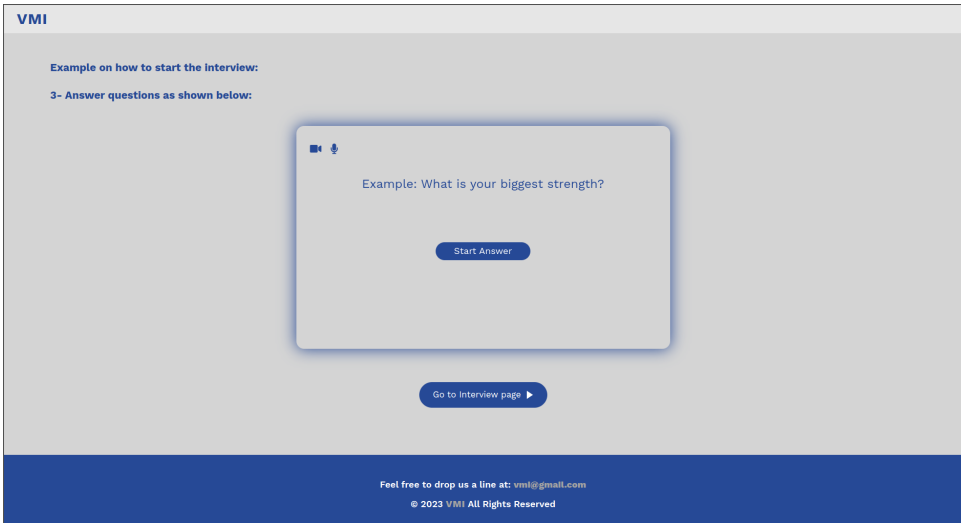

Figure 5.4: Sample ExampleCase

5.5 Interview

5.5.1 Interview.js

```
import React, { useEffect } from 'react';
import { useState } from 'react';
import {useNavigate} from 'react-router-dom';
import axios from "axios";
import Webcam from "react-webcam";
import { useCookies } from 'react-cookie';
import { v4 as uuidv4 } from 'uuid';

const Interview = () => {
    const [cameraViewToggle, setCameraViewToggle] = useState(false);

    const [interviewId, setInterviewId] = useState('');

    useEffect(() => {
        const generatedInterviewId = uuidv4(); // Generate a random
            user ID
        setInterviewId(generatedInterviewId);
        setCookie('interview_id', generatedInterviewId, { path: '/'
            });

    }, []);

    const [cookies, setCookie] = useCookies(['job-field',
        'interview_id']);
    const jobField = cookies['job-field'];
    const [questions, setQuestions] = useState([]);
```

```
const client = axios.create({
    baseURL: "http://127.0.0.1:5000"
});

useEffect(() => {
    const fetchData = async () => {
    try {
        const endpoint = "/questions";
        const response = await client.get(endpoint, {
        headers: {
            'Content-Type': 'application/json',
            Accept: 'application/json'
        },
        params: {
            job_field: jobField
        },
        data: {}
        });

        const extractedQuestions = response.data.slice(0,
            5).map(question => question.question_text);
        setQuestions(extractedQuestions);
    } catch (error) {
        console.error('Error:', error);
        // Handle error response here, such as displaying an
            error message to the user
    }
    };

    fetchData();
}, [jobField]);

const navigate = useNavigate();

const [currentQuestionIndex, setCurrentQuestionIndex] =
    useState(0);
const [showFirstQuestion, setShowFirstQuestion] =
    useState(false);
const currentQuestion = showFirstQuestion ?
    questions[currentQuestionIndex] : '';

const sendAndChange = () => {
    setCurrentQuestionIndex(currentQuestionIndex + 1);
}

const sendAndChange2 = () => {
    setShowFirstQuestion(true);
}
```

```
const [counter, setCounter] = useState(0);
const [buttonText, setButtonText] = useState('Start Interview');

const [isClicked, setIsClicked] = useState(false);
const handleClick = () => {
    setIsClicked(true);
}

const webcamRef = React.useRef(null);
const mediaRecorderRef = React.useRef(null);
const [capturing, setCapturing] = React.useState(false);
const [recordedChunks, setRecordedChunks] = React.useState([]);

const handleDataAvailable = React.useCallback(({ data }) => {
    if (data.size > 0) {
        setRecordedChunks((prev) => prev.concat(data));
    }
    }, []);

const handleStartCaptureClick = React.useCallback(() => {
    setCapturing(true);
    mediaRecorderRef.current = new
        MediaRecorder(webcamRef.current.stream, {
    mimeType: "video/webm;codecs=vp9,opus"

});
    mediaRecorderRef.current.addEventListener(
        "dataavailable",
        handleDataAvailable
    );
    mediaRecorderRef.current.start();
}, [webcamRef, setCapturing, mediaRecorderRef,
    handleDataAvailable]);

const handleStopCaptureClick = React.useCallback(() => {
    mediaRecorderRef.current.stop();
    setCapturing(false);
}, [mediaRecorderRef, setCapturing]);

const handleSendToBackend = React.useCallback(() => {
    if (recordedChunks.length) {
    const blob = new Blob(recordedChunks, {
        type: "video/webm"
    });
    const videoFilename = `${interviewId}_${counter -
        1}_video.webm`;
    const formData = new FormData();
```

```
    formData.append("file", blob, videoFilename);
    formData.append("interview_id", interviewId);
    formData.append("question", questions[counter - 2]);
    formData.append("video_filename", videoFilename);
    axios.post(`http://127.0.0.1:8000/file/${interviewId}`,
        formData)
        .then(response => {
        console.log(response.data);
        })
        .catch(error => {
        // Handle the error
        });
    setRecordedChunks([]);
    }
}, [recordedChunks]);

const firstStartCapturing = () => {
    if (capturing) {
        handleStopCaptureClick();
        console.log("capturing is set to true");
    }
    handleStartCaptureClick();

    console.log("First recording started");
};

const nextQuestionCapturing = async () => {
    if (capturing) {
        handleStopCaptureClick();
        console.log("Recording stopped");
        handleSendToBackend();
        console.log("Recording sent to backend");
    }
    handleStartCaptureClick();
    console.log("new question Recording started");
};

const lastQuestionCapturing = async () => {
    if (capturing) {
        handleStopCaptureClick();
        console.log("last question Recording stopped");
        handleSendToBackend();
        console.log("last question Recording sent to backend");
        navigate('/report');
    }
};

const [startFlag, setStartFlag] = useState(false);
const [timeRemaining, setTimeRemaining] = useState(60*2);
```

```
const [displayTimer, setDisplayTimer] = useState(false);

const [displayRunningLate, setDisplayRunningLate] =
    useState(false);

const formatTime = (seconds) => {
    const minutes = Math.floor(seconds / 60);
    const remainingSeconds = seconds % 60;
    return `${minutes.toString().padStart(2,
        '0')}:${remainingSeconds.toString().padStart(2, '0')}`;
};

const handleResetClick = () => {
    setTimeRemaining(60*2);
    setDisplayTimer(true);
    console.log(displayTimer);
};

const handleTimerEnd = () => {
    console.log('Timer reached zero!');
    MajorFunction();
};

const handleTimerLate = () => {
    console.log('Last ten seconds!');
    setDisplayRunningLate(true);
};

useEffect(() => {
    let interval = null;

    if (timeRemaining > 0 && displayTimer) {
        interval = setInterval(() => {
        setTimeRemaining((prevTime) => prevTime - 1);
        }, 1000);
    }

    if (timeRemaining === 0) {
        handleTimerEnd();
    }else if (timeRemaining === 10) {
        handleTimerLate();
    }

    return () => {
    clearInterval(interval);
    };
}, [timeRemaining, displayTimer]);

useEffect(() => {
```

```
    console.log("we are inside reload effect", counter);

    const handleBeforeUnload = (event) => {
    if (counter !== 0) {
        event.preventDefault();
        event.returnValue = '';
        return 'Can not perform this action now';
    }
    };

    window.addEventListener('beforeunload', handleBeforeUnload);

    return () => {
    window.removeEventListener('beforeunload',
        handleBeforeUnload);
    };
}, [counter]);

function MajorFunction() {

        if (counter === 0) {
            setStartFlag(true);
            firstStartCapturing();
            handleClick();
            sendAndChange2();
            setCounter(counter + 1);
            setButtonText('Next Question');
            console.log(counter);
            setDisplayRunningLate(false);
            handleResetClick();

        } else if (counter === 4) {
            nextQuestionCapturing();
            sendAndChange();
            setButtonText('End Interview');
            setCounter(counter + 1);
            console.log(counter);
            setDisplayRunningLate(false);
            handleResetClick();

        }
        else if (counter === 5) {
            nextQuestionCapturing();
            setIsClicked(false);
            sendAndChange();
            setButtonText('To Report Page');
            setDisplayRunningLate(false);
            setCounter(counter + 1);
            setDisplayTimer(false);
```

```
            setCameraViewToggle(false);
            console.log(counter);

        }
        else if (counter === 6) {
            console.log("i am in counter 6")
            lastQuestionCapturing();
            sendAndChange();

        }
        else {
            nextQuestionCapturing();
            sendAndChange();
            setCounter(counter + 1);
            console.log(counter);
            setDisplayRunningLate(false);
            handleResetClick();
        }
}

useEffect(() => {
    window.scrollTo(0, 0);
    }, []);

function getQuestion(){
    if (startFlag === false){
        return 'Start whenever you are ready, \n Good Luck!'
    }
    if(currentQuestionIndex < 5){
    return `Q${currentQuestionIndex + 1}: ${currentQuestion}`
    }
    else{
        return 'Congratulations! You have completed the
            interview.'
    }
}
function disableButton(){
    if(currentQuestionIndex < 5){
        return false;
    }
    else{
        return true;
    }
}

return (
    <>
```

```
<div className="instructions-parent
    instructions-parent-for-interview" >
    <div className="instructions-container" >
    </div>
    <div className="question-container">
        <div className="question-box
            question-box-for-interview">
            <div className={`icons-holder-interview
                icons-for-interview ${isClicked ?
                'clicked' : ''}`}>
                <div>
                    <i className="fas fa-video"></i>
                    <i className="fas fa-microphone"></i>
                </div>
                <div style={{ display: disableButton() ?
                    'none' : 'block' }}>
                    {/* <button
                        onClick={toggleVideoButton} >show
                        cam</button> */}
                    {/* <button
                        onClick={toggleVideoButtonOff} >No
                        cam</button> */}
                    <label className="switch" >
                        <input
                            type="checkbox"

                            onChange={()
                                =>{setCameraViewToggle(!cameraViewToggle);}}
                        />
                        <span className="slider round" />
                    </label>
                    <span
                        className="toggle-text">{cameraViewToggle
                        ? 'Video On' : 'Video Off'}</span>
                    <div className={`timer ${displayTimer
                        ? 'timer-update' : ''}`}
                        >{formatTime(timeRemaining)}</div>
                </div>

            </div>
            <div className= "running-container" >
                <p className={`time-running
                    ${displayRunningLate ?
                    'time-running-update' : ''}`} >Time
                    Low</p>
            </div>
            <div className="question"
                id={`${cameraViewToggle ? 'questionId' :
                'questionId-without-video'}`}>
```

```
                        {getQuestion()}
                    </div>

                    <Webcam className='interview-vid'
                        audio={true} ref={webcamRef} muted={true}
                        style={{ display: cameraViewToggle ?
                        'block' : 'none' }} />
                </div>
            </div>
            <div className='link-container-for-example-page
                link-container-for-interview-page' >
                <button className="button-start"
                    onClick={MajorFunction} >{buttonText} <span
                    className="triangle"></span></button>
            </div>
        </div>

    </>
  )
}
export default Interview
```

This code is a React component called **Interview**. It imports necessary dependencies such as **React**, **useEffect**, **useState**, **useNavigate**, **axios**, **Webcam**, **useCookies**, and **uuidv4**. The component defines several state variables using the **useState** hook, such as **cameraViewToggle**, **interviewId**, **questions**, **currentQuestionIndex**, **showFirstQuestion**, **counter**, **buttonText**, and **isClicked**. It also utilizes the **useEffect** hook to perform side effects when certain dependencies change.

The component sets up an effect to generate a unique interview ID using **uuidv4** and stores it in the **interviewId** state variable. It also sets a cookie with the interview ID using **setCookie** from the **useCookies** hook. Another effect is used to fetch questions from a server based on the **jobField** cookie value. The fetched questions are stored in the **questions** state variable.

The component defines functions to handle capturing video using the webcam, sending the recorded video to the backend, and controlling the flow of the interview process. These functions are triggered by button clicks and state changes.

The component also defines additional state variables for managing the timer and displaying timer-related messages. It also includes helper functions such as **formatTime** to convert seconds to a formatted time string.

An effect is set up to decrement the **timeRemaining** state variable every second if the **displayTimer** flag is set. When the timer reaches zero, the **handleTimerEnd** function is called. Additionally, when the timer reaches the last ten seconds remaining, the **handleTimerLate** function is called to display a "Time Low" message.

Another effect is set up to handle browser reload or close events. If the **counter** state variable is not zero, it prevents the default action and displays a confirmation message to the user.

The **MajorFunction** function is called in response to button clicks and state changes. It controls the flow of the interview process based on the **counter** value. It starts capturing video, sends it to the backend, and updates the state variables accordingly.

The component also includes rendering logic to display the question, timer, webcam view, and buttons. The **getQuestion** function determines the displayed question based on the **startFlag** and **currentQuestionIndex** state variables. The **disableButton** function disables the button when all questions have been answered.

Overall, the **Interview** component manages the interview process, including generating an interview ID, fetching questions, capturing and sending video, handling the timer, and controlling the flow of the interview.

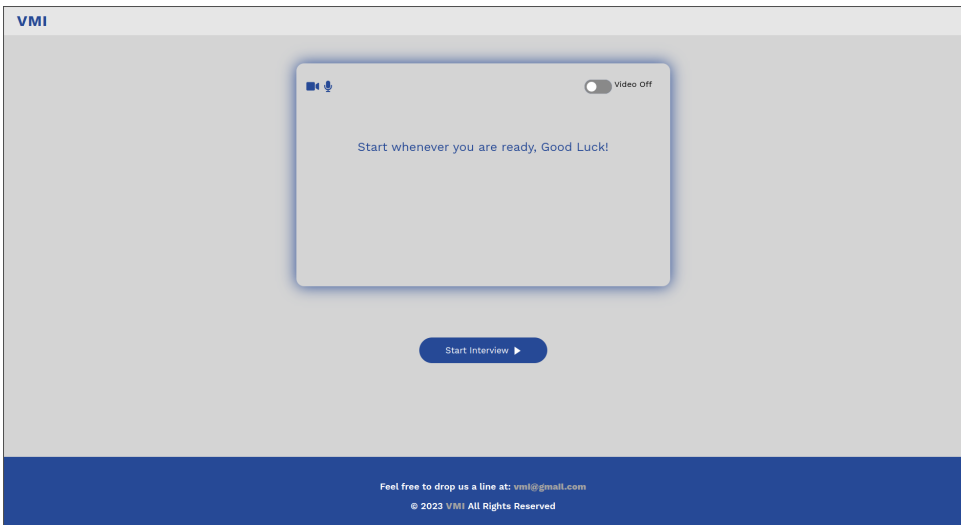

Figure 5.5: Sample Interview: 1

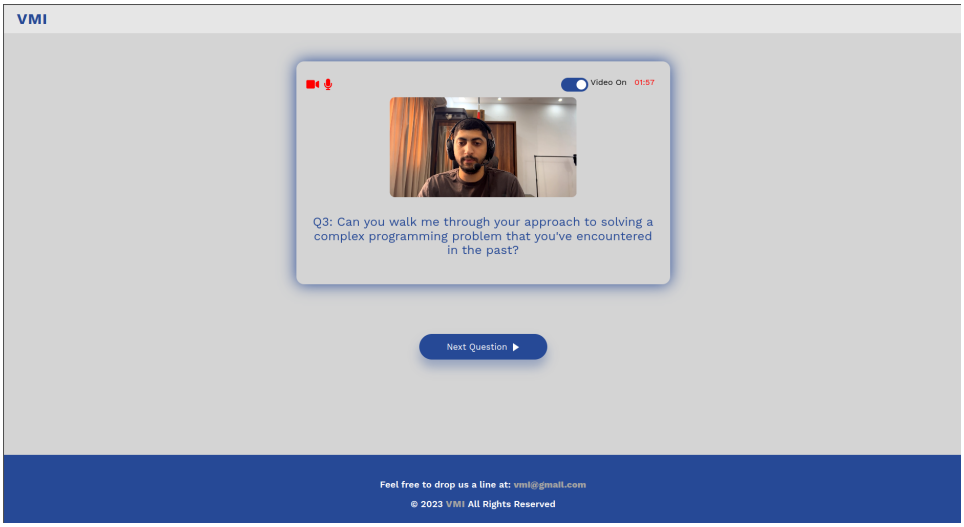

Figure 5.6: Sample Interview: 2

5.6 Report

5.6.1 Report.js

```
import React from 'react';
import { useEffect, useState } from 'react';
import axios from "axios";
import { useCookies } from 'react-cookie';
import ReactPlayer from 'react-player'

const Report = () => {
    const [cookies] = useCookies(['interview_id']);
    const [results, setResults] = useState([]);
    useEffect(() => {
      window.scrollTo(0, 0);
    }, []);

    function getVideoAnalysis(videoId) {
      return new Promise((resolve, reject) => {
        const retryDelay = 30000; // 30 seconds
        const retry = () => {
          axios
            .get(`http://127.0.0.1:5000/report?interview_id=${cookies['interview_id']
            }&video_id=${videoId}`)
            .then(response => {
              increaseProgress();
              resolve(response.data);
            })
            .catch(error => {
              // Retry after the specified delay
```

```
          setTimeout(retry, retryDelay);

        });
    };

    retry(); // Start the initial request
  });
}

useEffect(() => {
  async function fetchData() {
    const videoIds = ['_1_video', '_2_video', '_3_video',
        '_4_video', '_5_video'];
    for (const videoId of videoIds) {
      const data = await getVideoAnalysis(videoId);
      setResults(prevResults => [...prevResults, data]);
    }
  }
  fetchData();
}, []);

const [progress, setProgress] = useState(0);

useEffect(() => {
          const fill =
              document.querySelector('.progress-bar-fill');
          const text = document.querySelector('.progress-text');

          fill.style.width = progress + '%';
          text.textContent = progress + '%';
        }, [progress]);

const increaseProgress = () => {
          setProgress(prevProgress => prevProgress + 20);
        };

const fill = document.querySelector('.progress-bar-fill');
const text = document.querySelector('.progress-text');
const wrapper = document.querySelector('.wrapper');

function api_call(file){
  return
      `http://127.0.0.1:8000/file/${cookies['interview_id']}/${file}`
}
function disableProgressBar(){
  console.log(progress)
  if(progress === 100){
    return true;
  }
```

```
  return false;
}
return (
    <>
        <div className='vis-container' >
            <div className="instructions-parent report-container"
                >

                <div className="instructions-container" >
                    <h2 className="main-title-2 report-title"
                        >Report:</h2>
                </div>
            </div>
            <div className="wrapper" style={{ display:
                disableProgressBar() ? 'none' : 'block' }}>
                <div className="progress-bar" >
                <span className="progress-bar-fill"></span>
                </div>
                <span className="progress-text">Download
                    starting...</span>
            </div>

          <div className='vis-img-cont-par'>
            {results.length !== 5 ? (
              <div className='vis-img-cont'></div>
            ) : (
              <>
                {results.map((obj, index) => {
                  const figures = obj.figures;
                  return (
                    <React.Fragment key={index}>
                        <h3 className='the-Q' >Question {index+1}
                            (Video and Text extracted)</h3>
                        <div className="visu-container_Holder">
                            <div id="video-containerrrr">
                                {/* <video className='visu-video'
                                    controls>
                                    <source
                                        src={require(`../${obj.video}`)}
                                        type="video/mp4" />
                                    <track
                                        src={require(`../${obj.vtt}`)}
                                        kind="captions"
                                        srcLang="en"
                                        label="English" />
                                </video> */}
                                <video className='visu-video'
                                    controls>
```

```
                        <source
                            src={api_call(obj.video)}
                            type="video/mp4" />
                        <track src={ api_call(obj.vtt)}
                            kind="captions"
                            srcLang="en"
                            label="English" />
                    </video>
                </div>
                {console.log(obj.highlighted_text)}
                <p className="hamadassa"
                    dangerouslySetInnerHTML={{ __html:
                    obj.highlighted_text[index+1]
                    }}></p>
            </div>

            <h3 className='the-Q' >Question {index+1}
                Results</h3>

            <div className="feedback-Holder">
                <h3>Feedback</h3>
                <p>{obj.gpt_response[index+1]}</p>
            </div>

            <div className="feedback-Holder">
                <h3>Complete Analysis</h3>
                <div className="visu-container_Holder
                    fig-Holder">
                    {figures.map((photo) => (
                    <img
                        className='vis-img'
                        src={api_call(photo)}
                        alt='just wait'
                        key={photo}
                    />
                    ))}
                </div>
            </div>

            {index !== results.length - 1 && <hr
                className='visu-hr' />} {/* Add the
                horizontal line except for the last
                figure */}
        </React.Fragment>
        );
    })}
  </>
 )}
</div>
```

```
            </div>
        </>
    )
}
export default Report
```

In Report component, we start by importing the necessary dependencies and libraries such as **React**, **useEffect**, **useState**, **axios**, and **ReactPlayer**. We also utilize the **useCookies** hook from **'react-cookie'** for handling cookies. The **Report** component is defined as a functional component.

Inside the component, we define some state variables using the **useState** hook, including **cookies** and **results**. The **useEffect** hook is used to scroll to the top of the page when the component is rendered. We also define a function called **getVideoAnalysis** that returns a promise. It makes an HTTP request using **axios** to fetch video analysis data asynchronously. If there is an error, it retries the request after a specified delay. Another **useEffect** hook is used to fetch the video analysis data for multiple **videoIds** asynchronously. The data is then stored in the **results** state.

We also define the **progress** state variable to track the progress of the video analysis. Another **useEffect** hook is used to update the progress bar based on the **progress** state. The **increaseProgress** function is defined to increment the **progress** state. Finally, the **api-call** function is defined to construct the API endpoint for fetching files, and the **disableProgressBar** function is defined to determine whether the progress bar should be disabled based on the **progress** state.

We define the JSX markup for the **Report** component. It starts with a **div** container with the className **'vis-container'**. Inside this container, we have an **instructions-parent** div with the className **'report-container'**, and an **instructions-container** div containing the main title **'Report:'**.

Next, we have a **wrapper** div that displays a progress bar if the **progress** is not 100. The progress bar is represented by a **div** with the className **'progress-bar'** and a child **span** element with the className **'progress-bar-fill'** that represents the filled portion of the progress bar. The progress percentage is displayed using a **span** element with the className **'progress-text'**.

After that, we have a **div** with the className **'vis-img-cont-par'** that conditionally renders different components based on the length of the **results** array. If the length is not 5, it renders a placeholder div with the className **'vis-img-cont'**. Otherwise, it maps through the **results** array and renders

the video, extracted text, feedback, and analysis for each question.

For each question, it displays the question number, the video player with controls, and the extracted text using the **dangerouslySetInnerHTML** prop to render HTML content so that we turn the span tags that represent the fillers to color red, bold and underline using CSS to show the fillers to the interviewee. It also displays the feedback and complete analysis, which includes images representing figures related to the question.

Finally, we have an **hr** element with the className **'visu-hr'** that is rendered for each question except the last one to add a horizontal line between the figures. The JSX markup is wrapped inside the main **div** with the className **'vis-container'**.

That's the overview of the code in Report.js. It handles fetching video analysis data, displaying progress, and rendering the report content based on the received data.

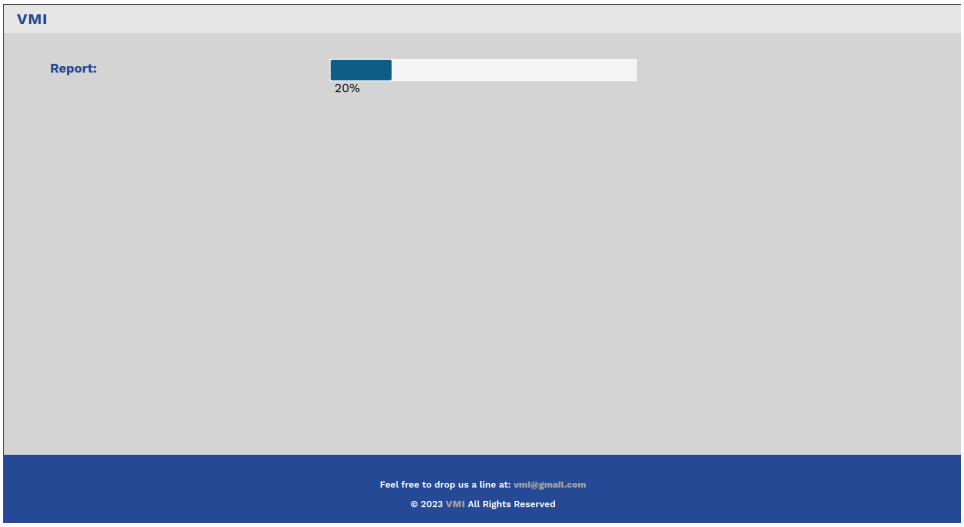

Figure 5.7: Sample Report: 1

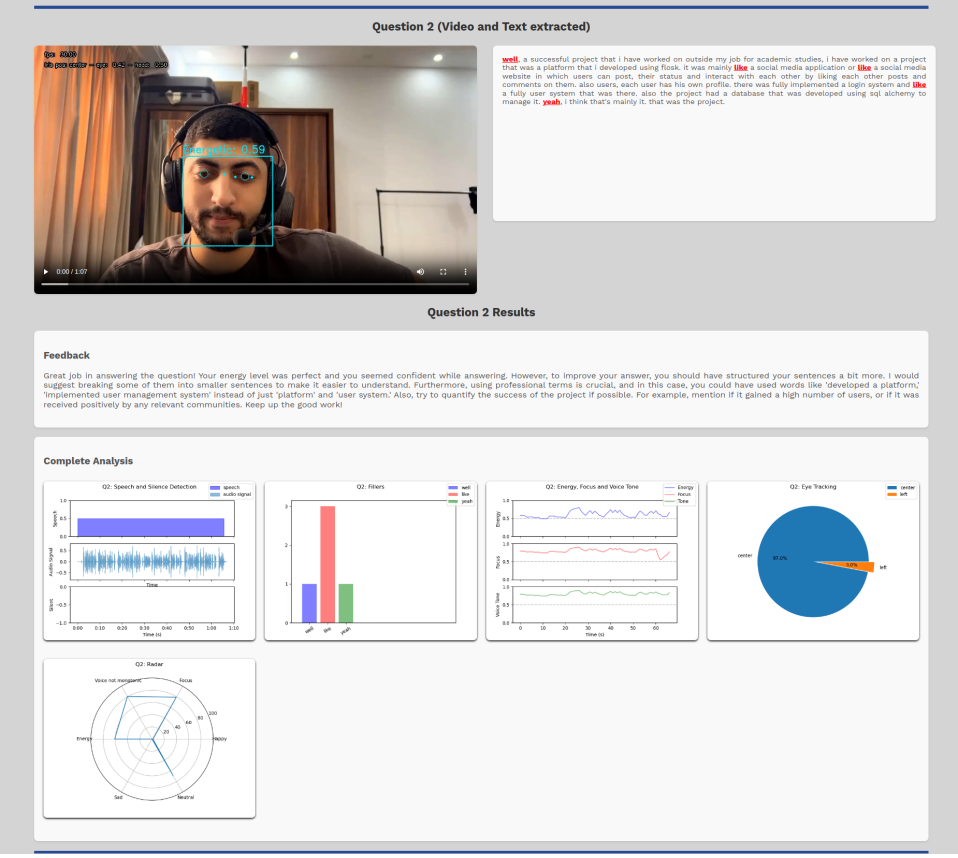

Figure 5.8: Sample Report: 2

5.7 Not Found

5.7.1 NotFound.js

```
const NotFound = () => {
    return (
        <div className="landing" >
            <div className="intro-text">
                <h1 className="websit-title" >This page is not
                    found</h1>
            </div>
        </div>
    )
}
export default NotFound
```

The **NotFound** component is responsible for rendering a page indicating that the requested page is not found or does not exist. It provides a simple

message to inform the user about the situation.

The `NotFound` component renders a container with the class `landing`. Inside this container, there is an `intro-text` div that contains a heading element (`h1`) with the class `websit-title`. The heading displays the message "This page is not found" to inform the user that the requested page could not be found.

Overall, the `NotFound` component provides a clear and concise message to the user when a page is not found, helping them understand the situation and navigate accordingly.

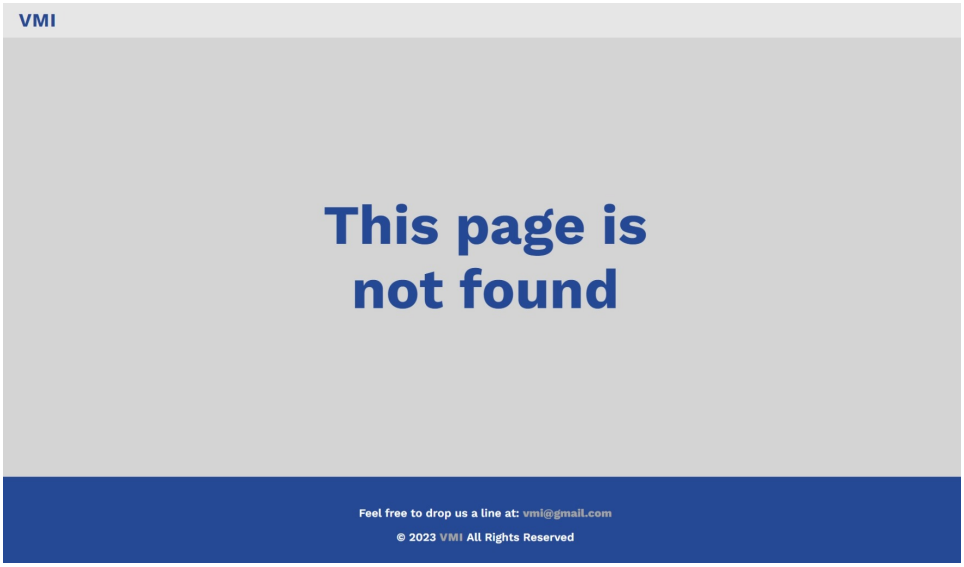

Figure 5.9: Sample NotFound

6 Back-end

The backend of the project plays a crucial role in providing the necessary data and processing capabilities to the frontend. It facilitates the communication between the user interface and the underlying algorithms and databases. In this section, we will explore the key functionalities and components of the backend system.

One of the primary responsibilities of the backend is to supply the frontend with the questions required for the interview process. Through a well-defined API, the frontend communicates with the backend to retrieve the necessary data. The backend serves as a reliable source of interview questions, allowing the frontend to present them to the user in a structured and organized manner.

The backend also handles the processing of videos submitted by the user through the frontend. These videos contain valuable information that needs to be extracted and analyzed to generate the required insights for creating the interview report. The backend utilizes advanced algorithms and models to perform both visual and audio analysis on the submitted videos. This analysis enables the extraction of relevant data points, such as facial expressions, speech patterns, and other behavioral cues, which are crucial for evaluating the candidate's performance during the interview.

To support the video analysis process, the backend seamlessly integrates with machine learning models. These models are trained to recognize and interpret various aspects of the video data, including facial expressions, sentiment analysis, and speech recognition. By leveraging these models, the backend can extract meaningful insights from the videos, enriching the evaluation process and contributing to the generation of comprehensive interview reports.

Once the necessary data has been collected and analyzed, the backend is responsible for generating the final interview report. It combines the retrieved interview questions, the analyzed video data, and any other relevant information to create a comprehensive and informative report. The backend ensures that the report is structured in a clear and concise manner, providing a detailed assessment of the candidate's performance and highlighting areas of strength and improvement.

6.1 init

```
from .video import Video
from .questions import Questions
from .report import Report
from app import api
```

```
api.add_resource(Questions, '/questions')
api.add_resource(Video, '/video')
api.add_resource(Report, '/report')
```

In the given code snippet, several resources and routes are defined using the Flask-RESTful framework. These resources are essential components of the backend system, responsible for handling various API requests and performing specific actions.

Firstly, the code imports three modules, namely Video, Questions, and Report. These modules encapsulate the logic and functionalities related to video processing, interview questions, and report generation, respectively. These modules are defined within the project's internal file structure.

Next, the code registers these modules as resources using the api.add_resource() function provided by Flask-RESTful. This function takes two parameters: the resource class and the route URL. The resource class represents a specific functionality or endpoint that the API can interact with. The route URL defines the path at which the resource can be accessed.

For instance, the Questions resource is registered with the route URL '/questions', which means that when an HTTP request is made to '/questions', the corresponding functionality defined in the Questions class will be executed. This resource is responsible for handling API requests related to retrieving interview questions.

Similarly, the Video resource is registered with the route URL '/video'. When an HTTP request is made to this URL, the associated functionality in the Video class is executed. This resource manages the processing and analysis of video data submitted by the frontend.

6.2 questions

```
class Questions(Resource):
    def __init__(self):
        self.parser = reqparse.RequestParser()
        self.parser.add_argument('job_field', type=str,
            location='args', required=True)
        self.field = None
    def get(self):
        # Get the job field from the request.
        # print request.args
        job_field = self.parser.parse_args().get('job_field')
        introduction_questions =
            Question.get_questions_by_field('Introductory', 1)
```

```
        non_tech_questions =
            Question.get_questions_by_field('non-technical', 1)
        tech_questions = Question.get_questions_by_field(job_field,
            3)
        questions = introduction_questions + non_tech_questions +
            tech_questions
        # Return the list of questions in json format.
        return jsonify([question.serialize() for question in
            questions])
```

In the provided code snippet, a Flask resource class named Questions is defined, which inherits from the Resource class provided by Flask-RESTful. This class represents an endpoint in the API that handles HTTP requests related to retrieving interview questions based on a specific job field.

The Questions class contains a constructor method __init__() that initializes the class attributes. Within the constructor, an instance of reqparse.RequestParser() is created and assigned to the attribute parser. This parser is responsible for parsing and validating the incoming HTTP request arguments. In this case, it expects a 'job_field' argument of type string, which is required and should be located in the URL query parameters ('args').

The Questions class also has a class attribute named field initialized with a value of None. This attribute is used to store the job field extracted from the request for further processing.

The Questions class defines a get() method, which is invoked when an HTTP GET request is made to the corresponding endpoint. Within this method, the job_field argument is extracted from the request using the parser.parse_args().get('job_field') statement. This retrieves the value of the 'job_field' argument from the URL query parameters.

After obtaining the job field, the code retrieves three types of interview questions based on the field. It calls the Question.get_questions_by_field() method, passing the job field and the desired number of questions for each type. The method returns a list of Question objects that match the specified criteria. The questions are categorized as 'Introductory', 'non-technical', and the provided job field.

The retrieved questions from each category are concatenated into a single list named questions. Finally, this list of questions is serialized using a list comprehension and converted into JSON format using the jsonify() function provided by Flask. The resulting JSON response contains the serialized questions.

Do note that the retrieved questions come from another model in the backend used to retrieve the questions. This can be considered one of the parts for storage that exist in our system, and it goes as follow

```
class Question(db.Model):
    __tablename__ = 'questions'
    id = db.Column(db.Integer, primary_key=True)
    question_text = db.Column(db.String(255), unique=False)
    field = db.Column(db.String(125), unique=False)
    category = db.Column(db.String(125), unique=False)
    difficulty = db.Column(db.String(125), unique=False)

    # get question by id
    @classmethod
    def get_question_by_id(cls, id):
        return cls.query.filter_by(id=id).first()

    # query questions by field
    @classmethod
    def get_questions_by_field(cls, field, limit):
        # query random questions by the given field
        return
            cls.query.filter_by(field=field).order_by(func.random()).limit(limit).all()

    # serialize question object
    def serialize(self):
        return {
            'id': self.id,
            'question_text': self.question_text,
            'field': self.field,
            'category': self.category,
            'difficulty': self.difficulty
        }


    def __repr__(self):
        return '<Question:{}:{}, Field: {}, category: {},
            difficulty: {}>'.format(self.id, self.question_text,
            self.field, self.category, self.difficulty)
```

The provided code defines a SQLAlchemy model class named Question that represents a table in the database called "questions". This class inherits from the db.Model class, which is likely a SQLAlchemy base model class associated with the application's database.

The Question class specifies the table name using the __tablename__ attribute and includes several column definitions representing the attributes

of a question: id, question_text, field, category, and difficulty. These attributes correspond to the columns in the "questions" table and are defined with their respective data types.

The class includes several class methods for querying and retrieving question data from the database. The get_question_by_id() method accepts an id parameter and queries the database to retrieve the question with the matching ID.

The get_questions_by_field() method accepts a field parameter and a limit parameter. It queries the database to retrieve a specified number of random questions based on the given field.

The serialize() method serializes a question object into a dictionary, representing the question's attributes. This method can be used to convert a Question object into a JSON-like structure for easy serialization and transmission.

The __repr__() method defines a string representation of the Question object, which is used for debugging and logging purposes. It returns a formatted string that includes the question's ID, question text, field, category, and difficulty.

6.3 video

```
ALLOWED_EXTENSIONS = {'mp4', 'mov', 'avi', 'wmv', 'flv', 'mkv',
    'webm'}

class Video(Resource):

    def __init__(self):
        self.parser = reqparse.RequestParser()
        self.parser.add_argument('question', type=str,
            location='form')
        self.parser.add_argument('interview_id', type=str,
            location='form')
        self.parser.add_argument('video_filename', type=str,
            location='form')

    def post(self):

        args = self.parser.parse_args()
        print(args)
        interview_id = args['interview_id']

        video_filename = args['video_filename']
        question = args['question']
```

```
        if secure_filename(video_filename) and
            self.allowed_file(video_filename):
            # pass video to the queue to be processed.
            video_id = video_filename.split('.')[0]
            app.config['video_queue_manager'].add_video(interview_id,
                video_id, question)

            # Return video is being processed .
            return ' video {} added to queue'.format(video_id), 200
        else:
            return "Invalid file name.", 400


    def allowed_file(self, filename):
        return '.' in filename and filename.rsplit('.',
            1)[1].lower() in ALLOWED_EXTENSIONS
```

The provided code defines a Flask resource class named Video, which handles HTTP requests related to video processing in the API. This class inherits from the Resource class provided by Flask-RESTful.

The Video class includes a constructor method __init__() that initializes the class attributes. Within the constructor, an instance of reqparse.RequestParser() is created and assigned to the attribute parser. This parser is responsible for parsing and validating the incoming HTTP request arguments. In this case, it expects a 'video' argument of type FileStorage, which should be located in the request's files.

An attribute named ALLOWED_EXTENSIONS is defined as a set of allowed file extensions for video files.

The Video class defines a post() method, which is invoked when an HTTP POST request is made to the corresponding endpoint. Within this method, it first checks if a 'video' file is present in the request's files. If not, it returns a response with a status code of 400 and a message indicating that no video file was found.

If a video file is present, the code proceeds to check if the file extension is allowed by calling the allowed_file() method. This method checks if the file extension is in the set of allowed extensions defined by ALLOWED_EXTENSIONS. If the file extension is not allowed, a response with a status code of 400 and a message indicating an invalid file format is returned.

If the video file passes the validation, further processing is performed. The secure filename is obtained using the secure_filename() function from the werkzeug.utils module. The video filename is then split to extract the in-

terview ID.

If the video filename contains '1_video', indicating it is the first video of the interview, a directory with the interview ID is created using os.mkdir().

The video file is saved to the appropriate directory based on the interview ID. If the current working directory matches the interview ID, the video file is saved directly. Otherwise, the video file is saved to the directory specified in the UPLOAD_FOLDER configuration variable.

After saving the video file, the video ID is extracted from the filename. The video_queue_manager (defined elsewhere) is then used to add the video to the processing queue asynchronously for further analysis.

Finally, a JSON response is returned, indicating that the video has been added to the queue for processing.

6.4 Report

The API includes a report endpoint that generates a report based on the provided interview ID and video ID. The generated report is returned as a JSON response.

```
class Report(Resource):
    def __init__(self):
        self.parser = reqparse.RequestParser()
        self.parser.add_argument('interview_id', type=str,
            location='args', required=True)
        self.parser.add_argument('video_id', type=str,
            location='args', required=True)
    def get(self):
        # Get the interview id from the request.
        interview_id = self.parser.parse_args().get('interview_id')
        video_id = self.parser.parse_args().get('video_id')
        # Get the report.
        video_result = ReportGenerator.generate_report(interview_id,
            video_id)

        if video_result['msg'] != 'success':
            return {'status':video_result['msg']}, 404


        return {
            'status': 'success',
            'highlighted_text': video_result['highlighted_text'],
            'text': video_result['text'],
            'gpt_response': video_result['gpt_response'],
            'figures': ['{}_video_0.png'.format(video_id[1]), #
```

```
            1_video_0.png
                    '{}_video_1.png'.format(video_id[1]),
                    '{}_video_2.png'.format(video_id[1]),
                    '{}_video_3.png'.format(video_id[1]),
                    '{}_video_4.png'.format(video_id[1])
                    ],
        'video': '{}{}.mp4'.format(interview_id, video_id),
        'vtt': '{}{}.vtt'.format(interview_id,video_id),
    }, 200
```

HTTP Method GET
Request Parameters The following parameters are required in the query string:

- **interview_id** (string): The ID of the interview.
- **video_id** (string): The ID of the video.

Preconditions Job Seeker has completed a previous virtual mock interview
Flow of Events

1. Job Seeker clicks the "Re-take Interview" button on the website
2. The website displays the first interview question and starts the countdown timer
3. Job Seeker answers the question or clicks "Next Question"
4. Steps 3-4 repeat until the Job Seeker clicks "Finish Interview"
5. A new performance report is generated for the Job Seeker

Request Example

```
GET /report?interview_id=123&video_id=456
```

Response The API response is a JSON object with the following fields:

- **status** (string): The status of the operation. Possible values: "success" or an error message.
- **highlighted_text** (string): The highlighted text from the report.
- **text** (string): The complete text of the report.
- **figures** (list): A list of file paths to image files associated with the report.

- **video** (string): The file path to the video file associated with the report.
- **vtt** (string): The file path to the VTT (WebVTT) file associated with the report.

Response Example

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "status": "success",
    "highlighted_text": "Lorem ipsum dolor sit amet.",
    "text": "Lorem ipsum dolor sit amet, consectetur adipiscing elit.",
    "figures": [
        "456_video_0.png",
        "456_video_1.png",
        "456_video_2.png"
    ],
    "video": "123456.mp4",
    "vtt": "123456.vtt"
}
```

6.5 Video Queue Manager

```
class VideoQueueManager:
    def __init__(self, video_analyzer):
        self.video_queue = Queue()
        self.video_analyzer = video_analyzer
        self.processing_lock = threading.Lock()
        self.worker_thread =
            threading.Thread(target=self.process_video_queue)
        self.worker_thread.start()

    def add_video(self, interview_id, video_filename):
        self.video_queue.put((interview_id, video_filename))

    def process_video_queue(self):
        while True:
            try:
                # Get video information from the queue
                interview_id, video_id = self.video_queue.get()
                # Acquire the lock to ensure exclusive access to the
                    processing function
```

```
                self.processing_lock.acquire()
                # Analyze the video.
                self.video_analyzer.analyze_video(interview_id,
                    video_id)

                # Release the lock to allow other videos to be
                    processed
                self.processing_lock.release()
                # Mark the video as done.
                self.video_queue.task_done()
            except Exception as e:
                logging.exception("Error processing video:
                    {}".format(e))

    def __del__(self):
        # Wait for the worker thread to finish processing the
            remaining videos
        self.video_queue.join()
        # Terminate the worker thread
        self.worker_thread.join()
```

The provided code defines a class named VideoQueueManager, which manages a queue of videos to be processed by the video_analyzer.

The class initializes with a video queue, a reference to the video_analyzer, a processing lock, and a worker thread. The worker thread is started, and its target is set to the process_video_queue() method.

The add_video() method is used to add a video to the video queue. It takes an interview_id and a video_filename as parameters and puts them into the video queue.

The process_video_queue() method is the target function for the worker thread. It continuously loops and retrieves videos from the video queue. Once a video is retrieved, it acquires the processing lock to ensure exclusive access to the analyze_video() method of the video_analyzer. It then calls the analyze_video() method, passing the interview_id and video_id as arguments. After the video analysis is complete, it releases the processing lock and marks the video as done by calling task_done() on the video queue. If an exception occurs during video processing, it logs the exception.

The __del__() method is a destructor that waits for the worker thread to finish processing the remaining videos in the video queue by calling join() on the video queue. It also terminates the worker thread by calling join() on the worker thread.

6.6 Video Analyzer

```
class VideoAnalyzer:
    @staticmethod
    def analyze_video(interview_id,video_id, question, DEBUG=False):

        openai.api_key = app.config['OPENAI_API_KEY']
        voiceModel = VoiceModel()
        result_dict = {}
        original_video_path =
            '{}/{}/{}'.format(app.config["UPLOAD_FOLDER"],
            interview_id, video_id)
        output_video_path =
            '{}/{}/{}'.format(app.config["DOWNLOAD_FOLDER"],
            interview_id, video_id)

        # using ffmpeg to convert the video to mp4 and seperate the
            audio from the video
        process1 = subprocess.call('ffmpeg -fflags +genpts -i
            {}.webm -r 30
            {}-input.mp4'.format(original_video_path,output_video_path),shell=True)
        process2 = subprocess.call('ffmpeg -i {}-input.mp4 -q:a 0
            -map a
            {}.wav'.format(output_video_path,output_video_path),shell=True)

        # using the voice model to analyze the audio
        result_dict.update(voiceModel.voiceModel(output_video_path ,
            DEBUG))
        # delete the voice model object to free up memory
        del voiceModel

        # create a process to run the facial model
        ctx = mp.get_context('spawn')
        queue = ctx.Queue()
        p1 = ctx.Process(target=VideoAnalyzer.analyze_vid_process,
            args=(queue ,output_video_path, DEBUG))
        p1.start()
        # get the result from the facial model process
        result_dict.update(queue.get())
        # terminate the facial model process and free up memory
        p1.terminate()
        # get the gpt response from openai and add it to the
            result_dict
        try:
            gpt_response = openai.ChatCompletion.create(
                model = "gpt-3.5-turbo",
                messages = [
                    {"role": "system", "content": "You are a helpful
                        interviewer that provides feedback on the
                        interviewee's answer directly to the
```

```
                    interviewee. Mention the interviewee's
                    sentences structure , also mention whether
                    the words they used are professional or not,
                    also comment on the energy of the interviewee
                    during answering the question, lastly,
                    provide examples whenever possible whenever
                    there is a window for improvment in the
                    interviewee's speech, sentences structure and
                    words. You must not ask questions."},
                {"role": "user", "content":'I got asked question:
                    '{}', and I answered '{}', mostly my energy
                    was {} during the question.'.format(question,
                    result_dict['text'],
                    result_dict['most_energy'])},
            ]
        )
        print(gpt_response['choices'][0]['message']['content'])
        result_dict['gpt_response'] =
            gpt_response['choices'][0]['message']['content']
    except:
        result_dict['gpt_response'] = "Sorry, OpenAI is not
            working right now, please try again later."
    # merge the audio and video together
    process3 = subprocess.call('ffmpeg -i {}-temp.mp4 -i {}.wav
        -c:v libx264 -c:a aac -strict -2 -map 0:v:0 -map 1:a:0
        {}-temp-audio.mp4'.format(output_video_path,
        output_video_path, output_video_path),shell=True)
    # delete the temp files
    #os.remove('{}-temp.mp4'.format(path_to_video_id))
    os.rename('{}-temp-audio.mp4'.format(output_video_path),
        '{}.mp4'.format(output_video_path))

    # create a pickle file to store the result dictionary
    with open('{}.pkl'.format(output_video_path), 'wb+') as f:
        pickle.dump(result_dict, f, pickle.HIGHEST_PROTOCOL)
    return

@staticmethod
def analyze_vid_process(queue,video_path, DEBUG):
    # load facial model here instead of init to utilize gpu usage
    facialModel = FacialModel()
    # using opencv library to process the video and send the
        frames to the facial model
    iris_pos_per_frame = []
    facial_emotion_per_frame = []
    facial_emotion_prob_per_frame = []
    energy_per_frame = []
    energy_prob_per_frame = []
```

```
iris= ""
emotion = ""
emotion_prob = 0
energy = ""
energy_prob = 0
frameCounter = 0

# load the video
cap = cv2.VideoCapture('{}-input.mp4'.format(video_path))
# get the fps of the video
fps = cap.get(cv2.CAP_PROP_FPS)
print("DEBUG: fps: {}".format(fps))
#creating video writer to write the output video
fourcc = cv2.VideoWriter_fourcc(*'mp4v')
output_video =
    cv2.VideoWriter('{}-temp.mp4'.format(video_path),
    fourcc, fps , frameSize = (int(cap.get(3)),
    int(cap.get(4))))

while cap.isOpened():
    ret, frame = cap.read()
    if not ret:

        print("video ended or failed to grab frame")
        break

    # get the iris position and facial emotion for the
        current frame
    iris, emotion, emotion_prob, energy, energy_prob, frame
        = facialModel.facialAnalysis(frame, emotion,
        emotion_prob, energy, energy_prob, frameCounter,
        fps, True)
    frameCounter += 1
    iris_pos_per_frame.append(iris)
    facial_emotion_per_frame.append(emotion)
    facial_emotion_prob_per_frame.append(emotion_prob)
    energy_per_frame.append(energy)
    energy_prob_per_frame.append(energy_prob)
    # write the frame to the output video
    output_video.write(frame)
    if cv2.waitKey(1) & 0xFF == ord('q'):
         break
cap.release()
# release the video writer and close all windows
cv2.destroyAllWindows()
# calculate what is the most frequent energy
energy = max(set(energy_per_frame),
    key=energy_per_frame.count)
# delete the facial model object to free up memory
```

```
del facialModel
result_dict = {
    'iris_pos_per_frame': iris_pos_per_frame,
    'facial_emotion_per_frame': facial_emotion_per_frame,
    'facial_emotion_prob_per_frame':
        facial_emotion_prob_per_frame,
    'energy_per_frame': energy_per_frame,
    'energy_prob_per_frame': energy_prob_per_frame,
    'most_energy': energy
}
# put the result dictionary in the queue to be used by the
    main process
queue.put(result_dict)
return
```

The given code defines a class **VideoAnalyzer** with a static method **analyze_video** and another static method **analyze_vid_process**. This class is responsible for analyzing a video by processing its frames and audio. The main purpose of the code is to perform facial analysis and audio analysis on the video, generate a response using OpenAI's GPT model, and merge the audio and video together.

In the **analyze_video** method, several parameters are passed, including **interview_id**, **video_id**, **question**, and **DEBUG** (which is set to **False** by default). The method starts by setting the OpenAI API key and initializing a **VoiceModel** object. It also defines a dictionary called **result_dict** to store the analysis results. The paths for the original video and the output video are set based on the provided **interview_id** and **video_id**.

Next, the code uses the FFmpeg library to convert the video to MP4 format and extract the audio from the video. The FFmpeg commands are executed using the **subprocess** module. After that, the audio is analyzed using the **voiceModel** object, and the results are added to the **result_dict**. The **voiceModel** object is then deleted to free up memory.

A separate process is created to run the facial analysis using the **analyze_vid_process** method. This is done using the **multiprocessing** module. The **analyze_vid_process** method loads a **FacialModel** object and processes the frames of the video using OpenCV. It extracts the iris position, facial emotion, and energy from each frame using the **facialAnalysis** method of the **FacialModel** class. The results are stored in lists for each frame. The processed frames are also written to an output video file. Finally, the most frequent energy value is calculated, the **facialModel** object is deleted, and the result dictionary is created with all the collected data.

Back in the **analyze_video** method, the GPT response from OpenAI is obtained by sending a message to the GPT-3.5 Turbo model. The message includes a system role message with instructions for providing feedback on the interviewee's answer, and a user role message containing the question and the interviewee's answer. The response from GPT is extracted and added to the **result_dict**.

The code then merges the audio and video together using FFmpeg and deletes the temporary files. A pickle file is created to store the **result_dict** as binary data. Finally, the method returns.

The **analyze_vid_process** method is a static method that receives a queue, a video path, and a **DEBUG** flag as arguments. It loads a **FacialModel** object and processes the video frames using OpenCV. It analyzes each frame using the **facialAnalysis** method of the **FacialModel** class to extract iris position, facial emotion, and energy. The results are stored in lists. The processed frames are written to an output video file. After processing all frames, the most frequent energy value is calculated, the **facialModel** object is deleted, and the result dictionary is created. The result dictionary is put into the queue to be used by the main process. Finally, the method returns.

Both methods utilize external libraries such as OpenCV, FFmpeg, and OpenAI's API for video processing, audio analysis, and natural language processing tasks, respectively.

7 Report Creation

The data obtained from each of the four models integrated within our system (iris tracking, speech to text, speech emotion recognition, and facial emotion detection) is utilized to generate a comprehensive and objective report for the end user. This report aims to aid the user in becoming a more effective interviewee by highlighting their strengths and areas for improvement.

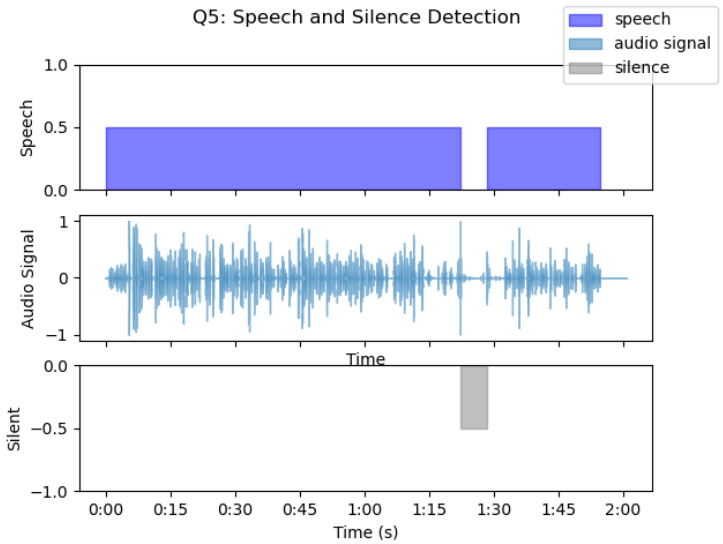

Figure 7.1: Speech and Silence Detection

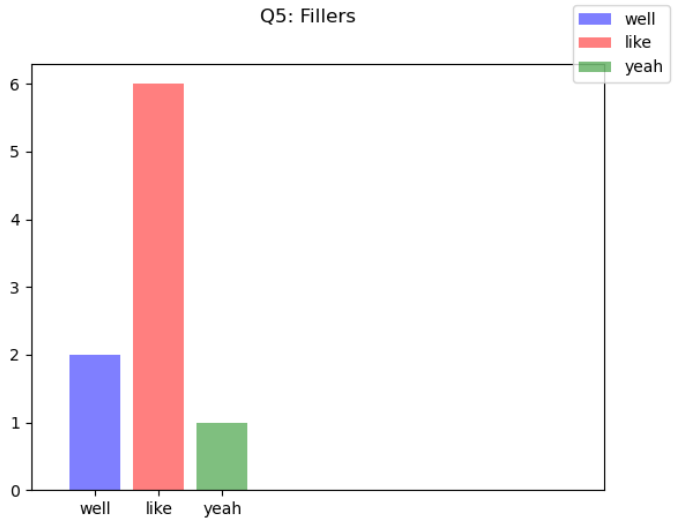

Figure 7.2: Fillers

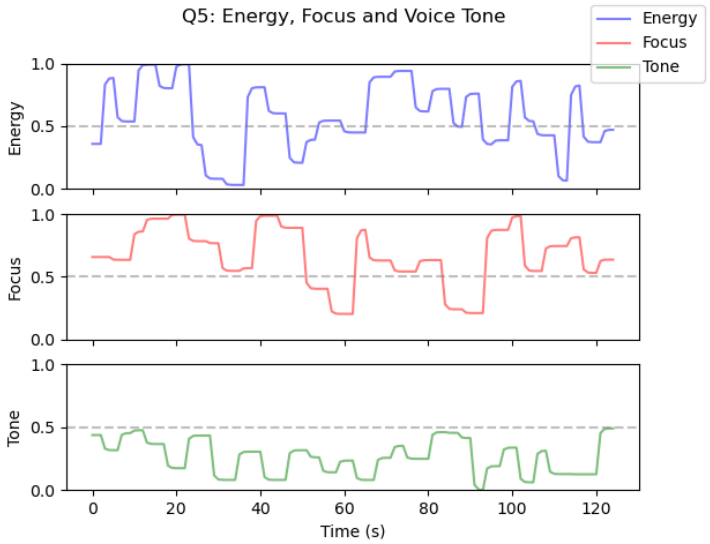

Figure 7.3: Energy, Focus and Voice Tone

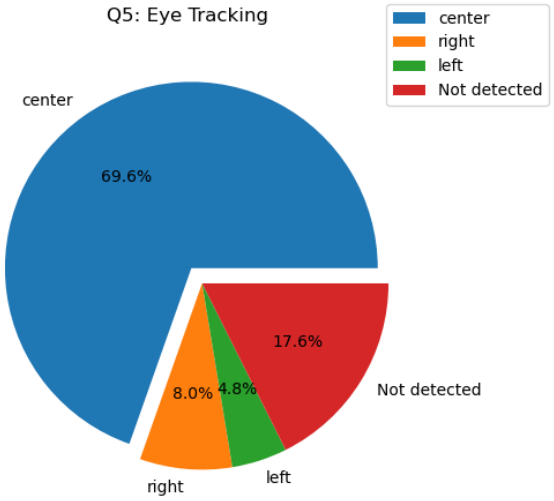

Figure 7.4: Eye Tracking

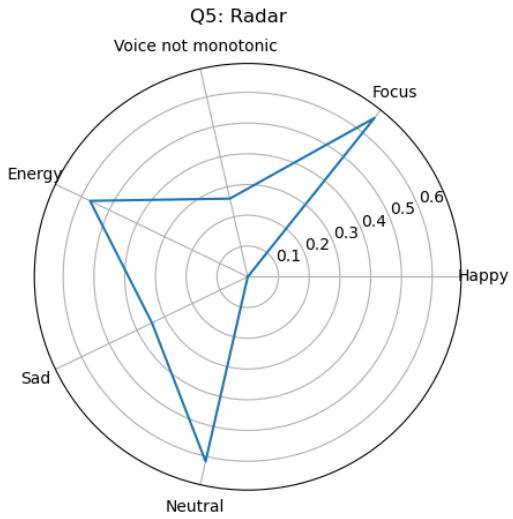

Figure 7.5: Radar

```
import matplotlib
matplotlib.use('agg') # to use matplotlib without gui support

class ReportGenerator:
    @staticmethod
    def generate_report(interviewId: str, videoId: str):
        # wait for video to be processed
        text = {}
        highlighted_text = {}
        path_to_file = app.config['UPLOAD_FOLDER'] + '/' +
            interviewId + '/'
        if not os.path.isfile(path_to_file + interviewId + videoId +
            '.pkl'):
            return {'msg': 'Video ID not found.'}
        else:
            i = int(videoId[1])
            df = pd.read_pickle(path_to_file + interviewId +
                videoId+'.pkl')
            y, sr = librosa.load(path_to_file + interviewId +
                videoId + '.wav')
            seconds = librosa.get_duration(y=y, sr=sr)
            framerate = int(len(df['iris_pos_per_frame']) / seconds)
            # lower the length of iris_pos_per_frame,
                facial_emotion_per_frame, energy_per_frame to be per
                second instead of per frame
            lite_df = {}
            lite_df.update(ReportGenerator.seconds_converter(df,
                framerate))
```

```
        text.update({i:df['text']})
        highlighted_text.update({i:df['highlightedText']})

        # plot the speech and silence of the question
        ReportGenerator.plot_speech(y, sr,
            df['silentTimeStamps'], df['speechTimeStamps'],
            path_to_file,i,0)
        #plot fillers of the question
        ReportGenerator.plot_fillers(
            df['simpleFillerDictionary'],
            df['complexFillerDictionary'],
            path_to_file,
            i,
            1)

        # add energy_val, focus_val, voice_tone_val to lite_df
        lite_df.update(ReportGenerator.category_to_number(
            lite_df['energy_per_second'],
            lite_df['focus_per_second'],
            lite_df['voice_tone_per_second']))
        # plot the energy, focus, voice_tone of the question
        ReportGenerator.plot_plot(
            ReportGenerator.lpf(lite_df['energy_val'], 0.9),
            ReportGenerator.lpf(lite_df['focus_val'],0.9),
            ReportGenerator.lpf(lite_df['tone_val'],0.9),
            path_to_file,
            i,
            2)
        # plot the iris tracking of the question
        ReportGenerator.plot_iris(
            lite_df['iris_pos_per_second'],
            path_to_file,
            i,
            3)
        # plot the radar of the question
        ReportGenerator.plot_radar(
            lite_df['facial_emotion_per_second'],
            lite_df['energy_per_second'],
            lite_df['focus_per_second'],
            lite_df['voice_tone_per_second'],
            ReportGenerator.lpf(lite_df['energy_val'], 0.9),
            ReportGenerator.lpf(lite_df['focus_val'],0.9),
            ReportGenerator.lpf(lite_df['tone_val'],0.9),
            path_to_file,
            i,
            4)
        print('DEBUG: done with ' + videoId)
    return {'msg': 'success',
```

```
            'text': text,
            'highlighted_text': highlighted_text
            }


    def plot_speech(y, sr, silentTimeStamps, speechTimeStamps,
        path_to_file, questionId, plotId):
        ...


    def plot_fillers(simpleFillerDictionary,
        complexFillerDictionary,path_to_file, questionId, plotId):
        ...

    def plot_plot(energy_val,focus_val, tone_val,path_to_file,
        questionId, plotId):
        ...

    def plot_iris(iris_pos_per_second,path_to_file, questionId,
        plotId):
        ...


    def plot_radar(facial_emotion_per_second, energy_per_second,
        focus_per_second,
        voice_tone_per_second,energy_val,focus_val,tone_val,
        path_to_file, questionId, plotId):
       ...
```

The ReportGenerator class is responsible for generating a report based on the provided interview and video IDs.

1. The `generate_report` method is the main entry point for report generation. It takes two parameters: `interviewId` (the ID of the interview) and `videoId` (the ID of the video associated with the interview).

2. Initially, the method waits for the video to be processed. This could involve any necessary preprocessing or analysis steps that need to be completed before generating the report.

3. The `text` and `highlighted_text` dictionaries are created to store text and highlighted text information for each video in the report.

4. The `path_to_file` variable is set to the path where the video files are stored, based on the provided `interviewId`.

5. The method checks if the video file exists by constructing the file path

using the `interviewId` and `videoId`. If the file does not exist, it returns a message indicating that the video ID was not found.

6. If the video file exists, the code proceeds with report generation. It starts by extracting relevant data from the video file and loading it into a DataFrame (`df`). Additionally, it loads the audio data (`y`) and sample rate (`sr`) using the Librosa library.

7. The duration of the audio file is calculated using the `librosa.get_duration` function, based on the audio data and sample rate. This information is used to calculate the frame rate (`framerate`) by dividing the number of frames in the DataFrame by the audio duration.

8. The `seconds_converter` function is called to convert frame-based data in the DataFrame to per-second data. The resulting per-second data is stored in the `lite_df` dictionary.

9. The `text` and `highlighted_text` dictionaries are updated with the text and highlighted text information extracted from the DataFrame.

10. Various plotting functions are called to generate visualizations for different aspects of the interview question. These visualizations include speech and silence plot, fillers plot, energy/focus/voice tone plot, iris tracking plot, and radar plot. Each plotting function takes relevant data from the `lite_df`, audio data (`y`), and other parameters, and saves the generated plots to the specified file path.

11. Finally, a success message along with the `text` and `highlighted_text` dictionaries is returned as the output of the `generate_report` method.

The generated report is expected to provide insights and visualizations that help analyze and understand various aspects of the interview question, including speech patterns, non-verbal cues, and emotional expressions.

8 General Algorithm

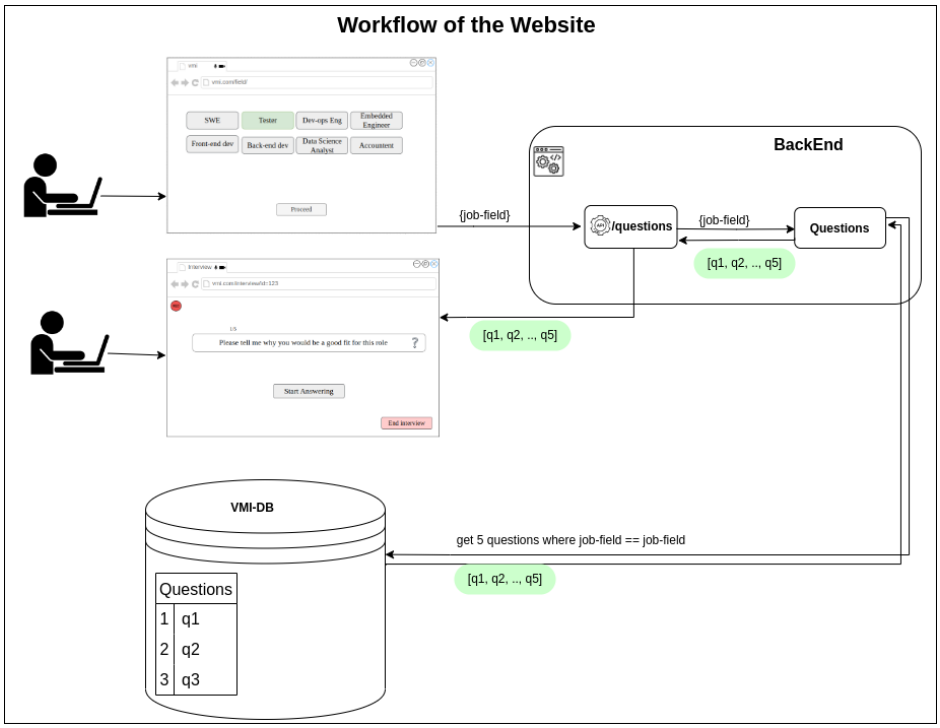

Figure 8.1: Workflow of the Website: part 1

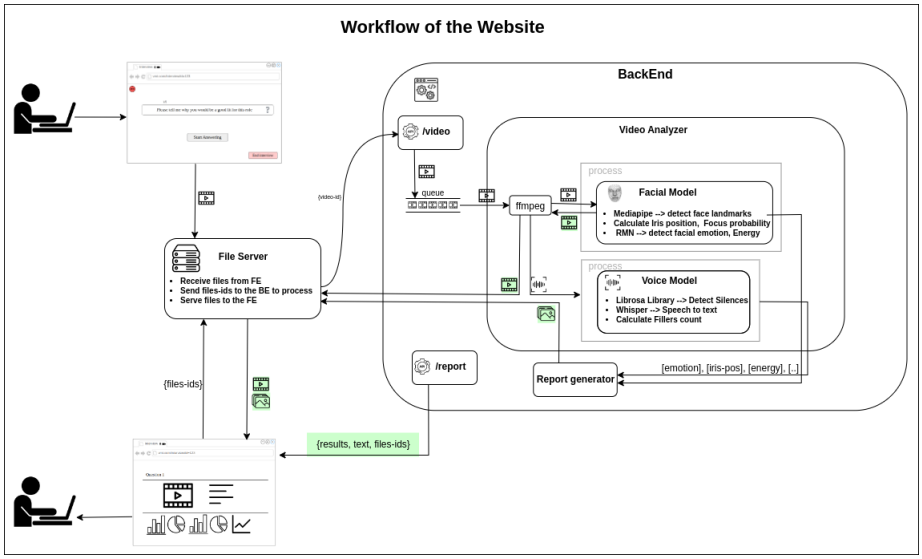

Figure 8.2: Workflow of the Website: part 2

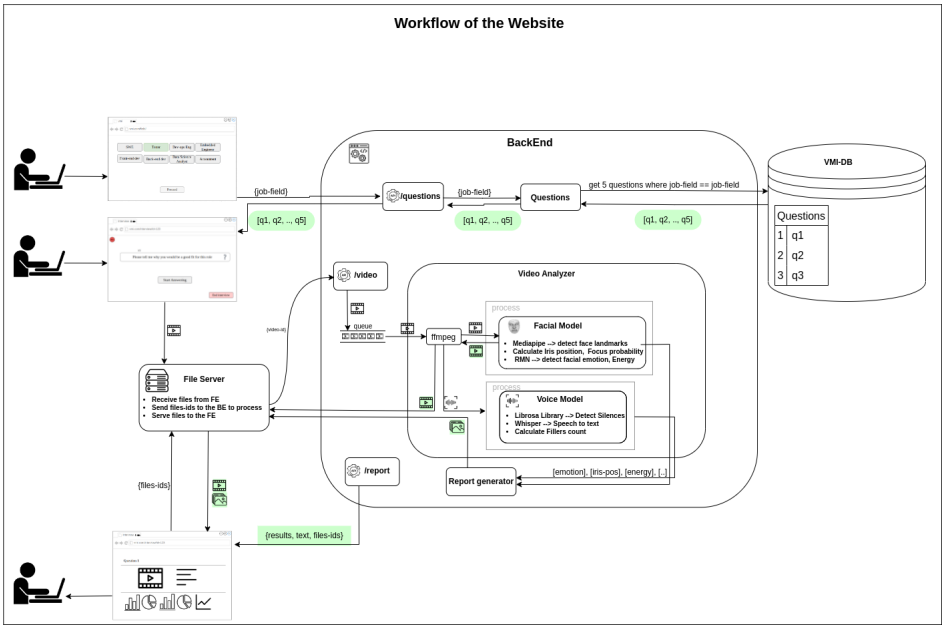

Figure 8.3: Workflow of the Website: together

First, the user enters the website and is presented with a user interface that includes a "Begin and Explore" button. Upon clicking this button, the user is redirected to the configuration page. At this stage, the website requests access to the user's camera and microphone, enabling the use of video recording and audio capabilities. Once the necessary permissions are granted, the user proceeds to the field selection phase.

In the field selection phase, the user is provided with options to choose a specific field or domain of expertise. After making a selection, this information is sent to the backend of the website for further processing. Subsequently, the backend retrieves five questions from the questions database. These questions are specifically tailored to the chosen field, consisting of three technical questions, one non-technical question, and one "breaking the ice" question aimed at creating a comfortable and engaging atmosphere.

Before proceeding to the interview page, the user is given the opportunity to view an example of an interview for better understanding and preparation. This serves as a reference to showcase how the interview process will be conducted and what is expected from the user.

Once the user feels ready to start the interview, they can begin by clicking the "Start Interview" button. At this point, an interview ID is generated by the system and stored in the user's cookies or session. This ID serves as a unique identifier for the ongoing interview.

As the user starts answering the questions, the video recording begins simultaneously. Each time the user proceeds to the next question, the video recording is automatically paused, and the recorded video is sent to a file server for storage and further processing. The local file server receives the videos and assigns a video ID to each recorded video. This video ID is then sent back to the backend and added to the video processing queue.

The video processing queue operates on a first-in, first-out basis. When the video ID reaches the top of the queue, the video analyzer component is triggered. The video analyzer performs various analysis tasks on the recorded video, including facial analysis, audio analysis, and potentially other forms of video analysis. The results of this analysis will be used for generating insights, providing feedback, and further evaluation of the interview performance.

Once the video reaches the video analyzer component, it undergoes a series of processes that can be effectively parallelized. Firstly, the video is split into its audio component, which is then extracted and passed to the voice model for further analysis. At the same time, the facial model is initialized to begin analyzing the visual aspects of the video. These processes can run concurrently, maximizing efficiency and reducing processing time.

Simultaneously, a new video file is created to store the final output. As the video analysis progresses, the results are incorporated onto each frame of the original video, enhancing it with additional information. Each modified frame is then added to the newly created video file. This merging of the analysis results with the original video ensures that the final output contains the desired visual enhancements and overlays.

Once both the audio and video analysis processes are complete, the two components are merged. The audio, which has been processed by the voice model, is synchronized with the modified video. This synchronized audio-video combination is then sent to the file server for storage and future access.

Additionally, the results of the video analysis, including the analysis data, the filename of the modified video, and any associated figures or visual representations, are compiled into a JSON format. This JSON file is then sent back to the user, providing them with a comprehensive report of the interview analysis.

After the user has finished the interview, they are redirected to the "/report" page. This page specifically requests the analysis data, video filename, and associated figure filenames. An API is triggered, initiating the report generator. The report generator utilizes the analysis data from both the facial and voice models to generate visual representations and figures that

encapsulate the interview performance and analysis.

During the report generation process, a progress bar is displayed to the user, providing real-time feedback on the progress of report generation. This progress bar helps the user understand the time it will take for the report to be fully generated.

Once the report is generated, it is visualized to the user, allowing them to access and review the comprehensive analysis of their interview. The report includes visual figures, insights, and observations derived from the facial and voice models, offering a detailed assessment of the interview performance and providing valuable feedback to the user.

9 Discussion & Conclusion

In conclusion, this project aimed to develop a comprehensive system for real-time emotion recognition using a combination of iris tracking, speech to text, speech emotion recognition, and facial emotion detection. The system was implemented using a variety of cutting-edge technologies such as MediaPipe Facemesh, Whisper, MLP classifier, and Keras. Throughout the development process, various techniques such as data preprocessing, model training, and integration testing were utilized to ensure the system's accuracy and robustness.

There were also multiple techniques for the front-end and the back-end to collaborate together for the ultimate final product such as the creation, modification, and usage of APIs for smooth communication and a higher degree of efficiency.

There was also the issue of scalability that got handled in the project using multiple techniques such as the utilization of queues and the modification process of GPU VRAM usage so as not to overload whatever system the webapp is running on.

It is worth noting that the project was not without its challenges. One of the main challenges faced was the integration of the various models, which required a significant amount of testing and debugging. As well as the perfecting of the queues as not to overload the system. In addition the creation of unique interview IDs without the need for tracking a unique user ID so as to avoid the creation of a login and user database system.

Overall, this project has demonstrated the feasibility of a real-time emotion recognition system and highlights the importance of interdisciplinary collaboration and the utilization of modern technologies in achieving such a goal. As the field of emotion recognition continues to evolve, it is expected that this system will serve as a valuable tool for various applications such as human-computer interaction, mental health, and market research.

10 References

Radford, A., Kim, J. W., Xu, T., Brockman, G., McLeavey, C., & Sutskever, I. (2022). Robust speech recognition via large-scale weak supervision. arXiv preprint arXiv:2212.04356.

McFee, Brian, Colin Raffel, Dawen Liang, Daniel PW Ellis, Matt McVicar, Eric Battenberg, and Oriol Nieto. "librosa: Audio and music signal analysis in python." In Proceedings of the 14th python in science conference, pp. 18-25. 2015.

Crabbé, A. H., Cahy, T., Somers, B., Verbeke, L.P., Van Coillie, F. (2020). Neural Network MLPClassifier (Version 0.18) [Software]. Available from https://bitbucket.org/kul-reseco/lnns.

Deng, L., 2012. The mnist database of handwritten digit images for machine learning research. IEEE Signal Processing Magazine, 29(6), pp. 141–142.

Chan, W., Park, D., Lee, C., Zhang, Y., Le, Q., and Norouzi, M. Speech-Stew: Simply mix all available speech recogni- tion data to train one large neural network. arXiv preprint arXiv:2104.02133, 2021.

Galvez, D., Diamos, G., Torres, J. M. C., Achorn, K., Gopi, A., Kanter, D., Lam, M., Mazumder, M., and Reddi, V. J. The people's speech: A large-scale diverse english speech recognition dataset for commercial usage. arXiv preprint arXiv:2111.09344, 2021.

Facade. Refactoring.Guru.(n.d.).Retrieved January 24, 2023, from https://refactoring.guru/design-patterns/facade