**SKINTELLECT: AI + IOT BASED SKIN ANALYSIS & RECOMMENDATIONS SYSTEM**

**STORY**

In the past, obtaining a comprehensive skin analysis was a cumbersome and time-consuming process, impacting people's daily lives in various ways. The journey often began with individuals recognizing the need to assess their skin health due to concerns or changing skin conditions. To get their skin analysed, individuals had to schedule appointments with dermatologists or specialized skin clinics, juggling busy work schedules and personal commitments to make time for these visits. The first obstacle was navigating through traffic and commuting to the clinics, which consumed valuable time and added stress to an already hectic lifestyle. Once at the clinic, patients often faced long wait times before meeting the dermatologist.

During the initial consultation, the dermatologist conducted a basic examination of the skin. However, for a more detailed analysis, additional specialized tests were often recommended, requiring separate appointments and further waiting for test results. The process of visiting the clinic multiple times for different tests meant days and even weeks of waiting for comprehensive results, leading to anxiety and uncertainty, causing further frustration. For individuals with demanding careers and busy personal lives, taking time off for multiple clinic visits was not only inconvenient but also challenging, balancing professional commitments with the pursuit of skin health became a daunting task.

Upon receiving the skin analysis report, patients often found the recommendations to be generic and lacking personalization, realizing that a more tailored approach to skincare was necessary for optimal results. However, this historical narrative underwent a paradigm shift with the advent of modern solutions that revolutionized skincare analysis. The introduction of technology, artificial intelligence, and innovative concepts marked a transformative era in skincare.

The integration of artificial intelligence empowered individuals to take charge of their skin health like never before, providing personalized care that addressed their specific skin conditions. This transformative shift marked a profound change, freeing people from the burden of time-consuming clinic visits and granting them the power to make informed skincare choices on their terms. Modern solutions opened new possibilities, allowing individuals to effortlessly integrate skincare analysis into their busy lives. With personalized benefits, individuals could embark on a journey of skincare with confidence, knowing that optimal results were within reach - all through a simple, efficient, and modern solution.

**INTRODUCTION**

This innovative project is designed to revolutionize the process of obtaining a comprehensive skin analysis and personalized skincare recommendations. By leveraging advanced technologies such as the **W5300-MQTT-CAM and QRCode** based system with a small display screen, artificial intelligence, and data-driven analysis, it offers a seamless and user-friendly experience.

Users can approach the booth and view a small display screen that shows a QR code for scanning. The moment a person comes within 1-meter range of the booth, the display screen activates and presents the QR code for scanning. If no one is in the proximity, it displays an inviting message, "Try SKINTELLECT!".

Once the user scans the QR code using their smartphone, the skin analysis process is initiated. They are then prompted to input their email id, phone number, age, and gender, which is collected and stored for further communication.

The W5300-MQTT-CAM module captures the user's picture, and the image is processed in the background using AI algorithms for skin pigmentation, skin tone, wrinkles, and skin contrast analysis.

The AI module generates values for various skin parameters, creating a detailed skin analysis report. This report includes insights into the presence of wrinkles, skin tone, skin texture, and pigmentation, along with personalized recommendations. The results are sent to the user's email id and phone number via email and SMS.

Additionally, this project includes Telegram notifications, where an automated message is sent to the administrator's Telegram account after the report is generated.

Moreover, this project offers personalized product recommendations powered by OpenAI (GPT 3.5), tailored to each individual's specific skin needs and concerns based on their analysis results.

By eliminating the need for multiple visits to dermatologists or clinics and reducing waiting periods for results, this project empowers users to take charge of their skincare journey conveniently and efficiently, making informed decisions for healthier and radiant skin. It also benefits skincare professionals and booths by enhancing customer experiences with detailed skin analysis and tailored product recommendations.

With a user-friendly interface and data-driven approach, this project promises to revolutionize skincare analysis, offering a seamless and personalized skincare experience for all users. It unlocks the true potential of every individual's skin health, making it an indispensable tool in the skincare industry. The future of skincare analysis is brighter than ever with this innovative project, offering a new era of personalized care.



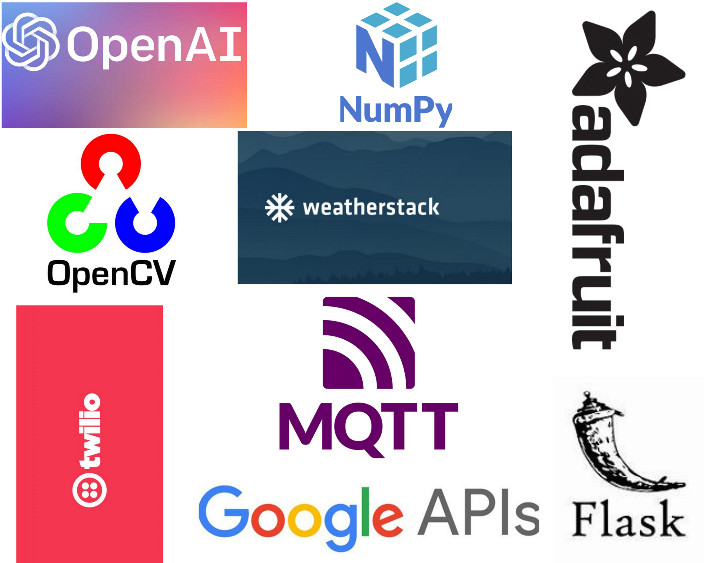
(Reference image)

**WHY SKINTELLECT?**

* Advanced technology: IoT, AI, and data-driven analysis.
* Personalized skincare recommendations based on individual analysis.
* Convenient and efficient - QR code scanning for analysis.
* Secure data handling for privacy protection.
* Enhanced customer experience with detailed reports.
* Hassle-Free Setup and Usage

**LIBRARIES, SOFTWARE COMPONENTS AND ONLINE SERVICES USED**

Here's the list of software components used in the project:



1. Flask

2. MQTT broker

3. mtcnn

4. OpenCV

5. Adafruit SSD130

6. Twilio API

7. Google Auth API (smtplib)

8. requests

9. jsonify

10. Weatherstack API

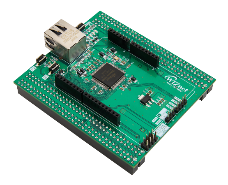
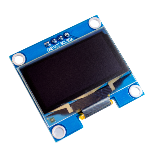
11. OpenAI (GPT 3.5)

12. Telegram Bot API

**HARDWARE COMPONENTS USED**

Here's the list of hardware components used in the project:

1. Ultrasonic sensor (HC-SR04)
2. OLED display (Adafruit SSD1306)
3. WIZnet W5300 TOE Shield Board
4. OpenMV H7 camera module
5. Jumper wires
6. Breadboard



**PROGRAMMING LANGUAGES USED**

Here's the list of programming languages used in this project:

1. Python
2. HTML
3. CSS
4. JAVASCRIPT
5. C/C++

**WORK FLOW (USER SIDE)**

From the user's perspective, the overall working of the SKINTELLECT project can be explained as follows:

1. **QR Code Interaction**: Users approach the SKINTELLECT booth with an integrated small screen and camera. A QR code is displayed on the screen when a user is within a 1-meter range, encouraging them to scan it using their smartphone's camera.

2. **Dynamic User Input**: After scanning the QR code, users are prompted to input their email address, phone number (with country code), age, and gender. Valid inputs are required for further analysis.

3. **Analysis Initiation**: Once the required details are provided, users tap the "Start Analysis" button. This triggers JavaScript to collect and send the data to the server for processing.

4. **Backend Processing**: Using Flask, the server communicates with MQTT. It sends a command via MQTT to capture the user's facial image through the attached W5300-MQTT-CAM device.

5**. Facial Image Capture**: The W5300-MQTT-CAM captures the user's facial image, which undergoes computer vision analysis. MTCNN identifies if a face is present.

6. **Skin Attribute Assessment**: Skin analysis begins if a face is detected:

- Wrinkle Detection: The number of edges on the face indicates wrinkle presence.

- Skin Tone Analysis: LAB colour space assesses skin tone.

- Skin Texture Analysis: GLCM calculates contrast and dissimilarity for texture.

- Pigmentation Analysis: HSV colour space isolates pigmented regions.

7. **Comprehensive Analysis**: Collected data, analysis results, and user inputs compile into a detailed skin analysis report.

8. **AI-Powered Recommendations**: The report prompts OpenAI. AI generates personalized skincare suggestions based on analysis results and user data.

9. **Instant Recommendations:** Users receive skincare recommendations via email and SMS, delivered by Gmail SMTP and Twilio APIs.

10. **Administrator Alert**: The administrator gets a Telegram notification when a report is generated. This can aid further product analysis.

This innovative update enhances user engagement by displaying a QR code on a screen for proximity-based interaction, streamlining the process and making personalized skincare accessible in a touchless manner.

**MODULES**

Here's a detailed explanation of each module in a step-by-step manner:

1. **Web Frontend Module**:

- This module creates the user interface for the SKINTELLECT web application.

- It displays a header with the title "SKINTELLECT" and a subheading "Intelligent Skin Analysis."

- It includes input fields for the user to enter their email address, phone number, gender, and age.

- The user clicks the "Start Analysis" button to initiate the skin analysis process.

2. **Flask App Module**:

- The Flask app module handles incoming requests from the web frontend.

- When the user clicks the "Start Analysis" button, the app collects the entered data, including email, phone number, gender, and age.

- It then sends a request to the MQTT Module to trigger image capture by the W5300-MQTT-CAM device.

- Once the image is uploaded, it is stored and an HTTP URL is generated.

- This URL offers a direct path for any internet-connected device to access the image.

3. **W5300-MQTT-CAM Interface Module**:

- WIZnet W5300-TOE-Shield: This module is intricately connected to the STM32 Nucleo-144 (F429ZI) boards, acting as the linchpin for internet connectivity. With its high-speed data transmission capability, the shield ensures real-time or near-real-time transmission of images, depending on the chosen resolution.

- Upon receiving the request from the Flask app, it sends a command to the W5300-MQTT-CAM to capture the user's facial image.

- The OpenMV camera module captures high-quality images on command

- The captured image is then saved for further processing.

4. **Image Processing Module**:

- The Image Processing module processes the captured facial image to analyse skin attributes.

- It uses the MTCNN algorithm for face detection to locate the user's face in the image.

- After detecting the face region, it calculates the number of wrinkles based on the edges in the face region.

- It analyses the colour and texture of the skin to determine skin tone and skin contrast.

- Additionally, it identifies skin pigmentation by isolating pigmented regions in the image.

5. **Recommendation Module**:

- The Recommendation module generates personalized skincare recommendations based on the user's skin analysis results and other input data.

- It sends the skin analysis results to the OpenAI API to request detailed skincare suggestions.

- The OpenAI model processes the input and generates personalized recommendations.

- The generated recommendations are received and extracted from the OpenAI API response.

6. **Email Module**:

- This module sends the personalized skincare recommendations to the user's email address.

- It uses the Gmail SMTP server for sending the email.

- The email includes the skin analysis results, along with the generated skincare suggestions.

7. **SMS Module**:

- The SMS Module sends personalized skincare recommendations to the user's phone number via SMS.

- It integrates with the Twilio API for sending SMS messages.

- The SMS includes the skin analysis results and the skincare suggestions.

8. **Telegram Notification Module**:

- This module sends a notification to the user's Telegram app to inform them that the SKINTELLECT report has been generated.

- It utilizes the Telegram bot API for sending the notification.

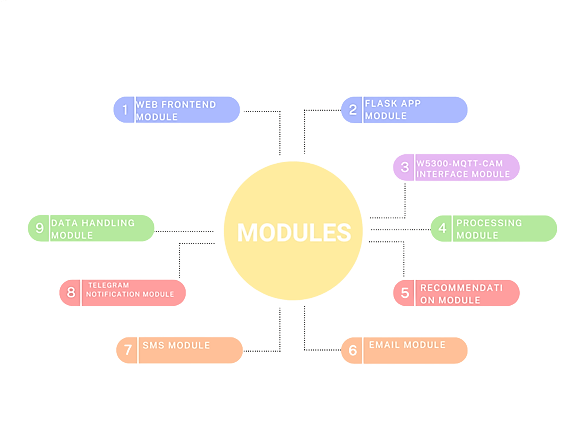
- The notification message includes a prompt for the administrator that a report has been generated for someone (providing no other information, for privacy).

- This is mainly for administrative purpose for better analysis of how product is being used etc…

9. **Data Handling Module**:

- The Data Handling module is responsible for securely storing and retrieving user data, including email addresses, phone numbers, and skin analysis results.

- It ensures that user data is properly handled and protected.



**BACKEND PROCESS (DEVELOPER SIDE)**

Backend Process of SKINTELLECT:

1. **Web Frontend Interaction and Proximity Detection:**

- When a user is within a 1-meter range of the SKINTELLECT device, they can access the SKINTELLECT web application by scanning a QR code (that gets displayed on oled screen attached only when user is in 1 meter range***: ultrasonic sensor*** otherwise shows “Try Skintellect!!” message) and opening a web browser on their device.

- They are greeted with a header displaying the title "SKINTELLECT" and a subheading "Intelligent Skin Analysis."

- Input fields are provided for the user to enter their email address, phone number (including the country code), gender, and age.

- To initiate the skin analysis process, the user clicks the "Start Analysis" button.

2. **Flask App Communication:**

- The Flask app module is activated when the user clicks the "Start Analysis" button after filling in their details.

- The Flask app collects the entered data, including the user's email address, phone number, gender, and age.

- This data is then packaged into a request to be sent to the MQTT module.

3. **W5300-MQTT-CAM Image Capture:**

- The request from the Flask app module reaches the W5300-MQTT-CAM interface module.

- Upon receiving the request, this module communicates with the W5300-MQTT-CAM device.

- A command is sent to the W5300-MQTT-CAM device to capture the user's facial image.

- The captured image is then stored for further processing.

4. **Image Processing and Analysis:**

- The captured facial image undergoes analysis in the Image Processing module.

- The MTCNN algorithm is used for face detection, accurately locating the user's face within the image.

- The module calculates the number of wrinkles by analysing the edges in the facial region.

- Skin tone and skin contrast are determined by analysing the colour and texture of the skin.

- Pigmentation regions are identified by isolating specific colours in the image using the HSV colour space.

5. **Data Integration with OpenAI:**

- After skin attributes are analysed, the Recommendation module comes into action.

- The skin analysis results, along with other input data, are sent to the OpenAI API.

- The OpenAI model processes this input and generates personalized skincare recommendations tailored to the user's skin analysis and other factors.

6. **Sending Recommendations via Email and SMS:**

- The personalized skincare recommendations are sent to the user via both email and SMS.

- The Email module uses the Gmail SMTP server to send an email containing the skin analysis results and skincare suggestions.

- The SMS module integrates with the Twilio API to send an SMS message to the user's phone number, including the same information.

7. **Telegram Notification for Admin:**

- Additionally, the Telegram Notification module provides administrative oversight.

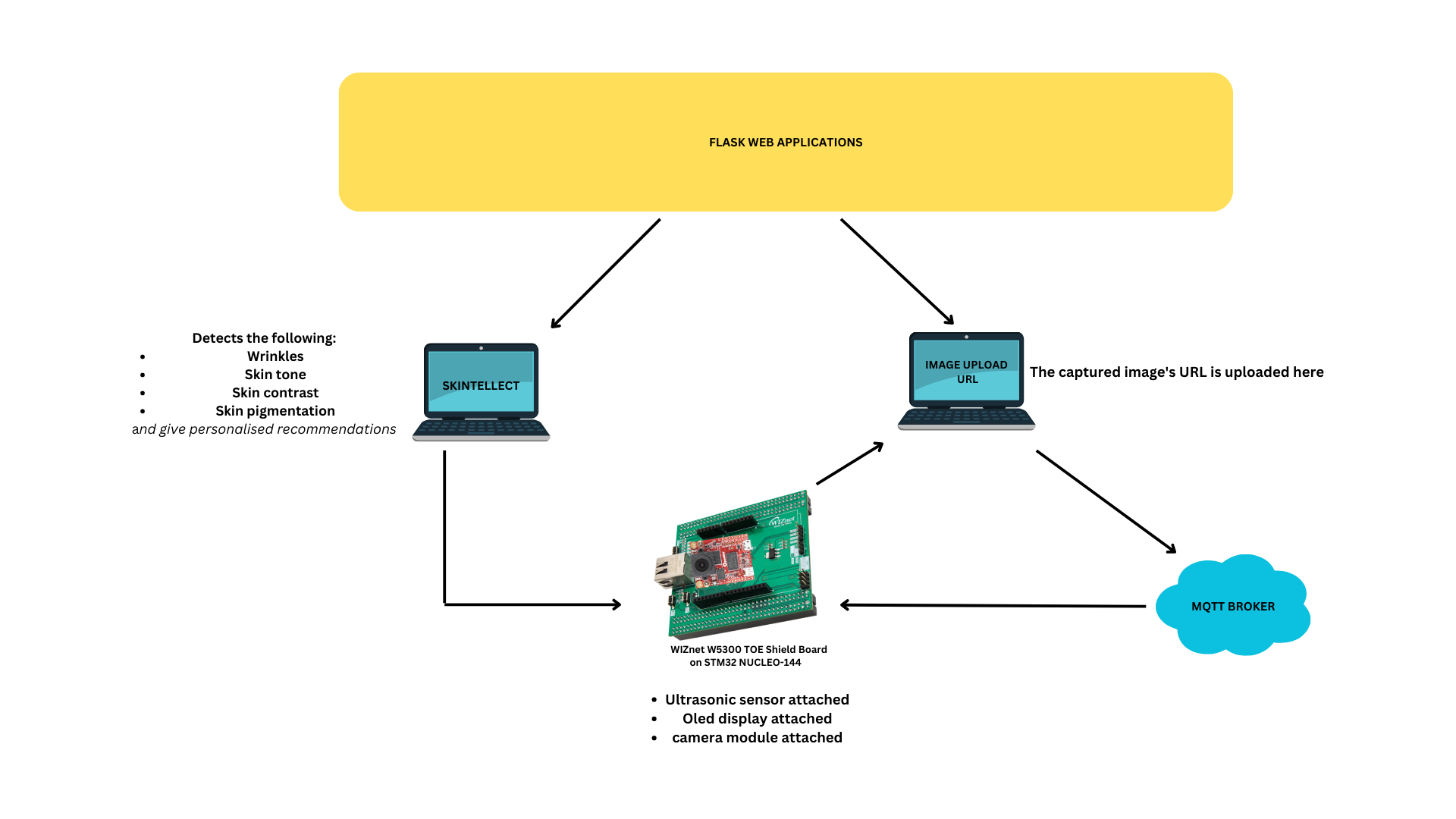
- A notification is sent to the administrator's Telegram app, indicating that a report has been generated.

- This notification helps administrators keep track of the reports generated and usage trends without compromising user privacy.

8. **Secure Data Handling:**

- Throughout this process, the Data Handling module ensures the proper handling and protection of user data.

- User information, such as email addresses, phone numbers, and skin analysis results, are securely stored and retrieved as needed.

****

**GET STARTED**

Let's walk through the detailed steps to set up and run the SKINTELLECT application on a different machine, incorporating the ultrasonic sensor and Adafruit OLED screen. This comprehensive guide covers hardware setup, software configuration, API integration, and more:

Arduino Environment:

To configure Arduino environment for STM32 Nucleo-144 (F429ZI), you need to follow the instructions of WIZnet official website.

LINK: https://github.com/Innovation4x/W5300-MQTT-CAM#arduino-environment

Hardware Setup:

1. **W5300-MQTT-CAM Device Setup:**

- Connect the W5300-MQTT-CAM device to the machine using the appropriate interface (e.g., USB).

- Ensure the device is recognized by the operating system. Install any required drivers or libraries

for the W5300-MQTT-CAM to function properly.

2. **Ultrasonic Sensor and Adafruit OLED Setup:**

- Connect the ultrasonic sensor to the machine's GPIO pins according to its specifications.

- Connect the Adafruit OLED screen to the machine's GPIO pins as well.

- Make sure both the ultrasonic sensor and OLED screen are correctly recognized by the system.

**PIN CONFIGURATION:**

|  |  |
| --- | --- |
| **Pin1 (Vcc)** | **3V3** |
| **Pin 2 (Ground)** | **GND** |
| **Pin3 (SCL)** | **GPIO PIN (15)** |
| **Pin4 (SDA)** | **GPIO PIN (14)** |

SENSOR FUNCTION

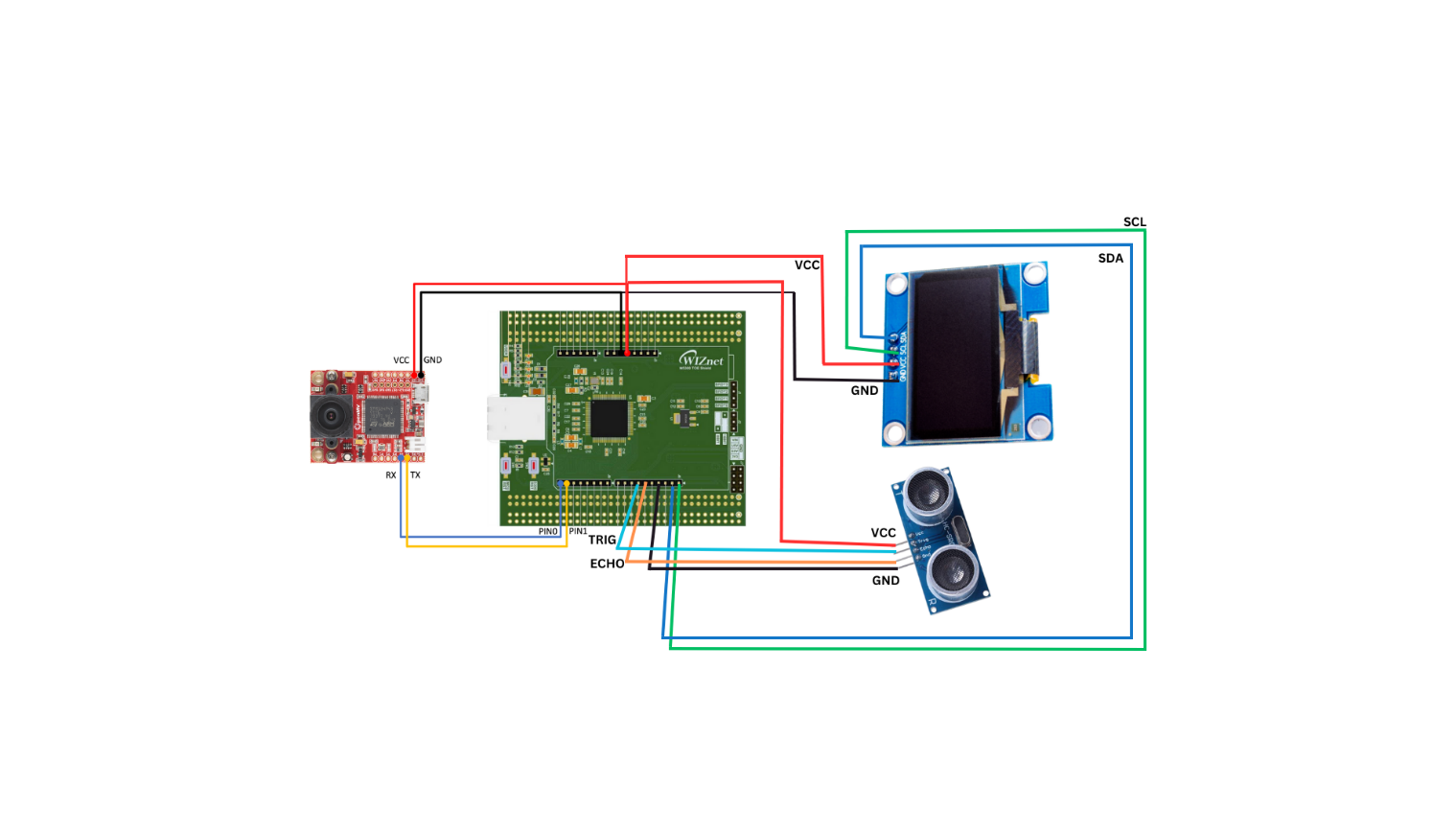
BOARD PIN

BOARD PIN

SCREEN FUNCTION

* GPIO PIN 0 , GPIO PIN 1 are reserved for connecting openMV camera

|  |  |
| --- | --- |
| **(Vcc)** | **5V (VBUS)** |
| **(Trigger)** | **GPIO PIN (11)** |
| **(Echo)** | **GPIO PIN (12)** |
| **(Ground)** | **GND** |

**CONNECTION:**

**HARDWARE CONFIGURATION FOR OPENMV CAMERA MODULE AND STM32 NUCLEO-144 BOARD:**

**Prepare Both Modules:**

* Ensure both the STM32 Nucleo-144 board and the OpenMV camera module are powered off to prevent any accidental shorts or damage.
* Lay out both modules on a non-conductive surface.

**Serial Connection:**

* Connect the TX pin (pin 1) of the STM32 Nucleo-144 board to the UART3\_RX (P5) pin of the OpenMV camera module. This will allow the Nucleo-144 board to transmit data which the OpenMV camera module will receive.
* Connect the RX pin (pin 0) of the STM32 Nucleo-144 board to the UART3\_TX (P4) pin of the OpenMV camera module. This setup ensures the Nucleo-144 board can receive data transmitted by the OpenMV camera module.

**Power Connection:**

* Connect the GND (Ground) pin of the OpenMV camera module to the GND pin of the STM32 Nucleo-144 board. This establishes a common ground between the two devices.
* Connect the VCC pin of the OpenMV camera module to the VCC (or an appropriate power supply pin, depending on the voltage requirements of the OpenMV) pin of the STM32 Nucleo-144 board. This will power the camera module.

Software Setup:

 3. **Python Installation:**

- If Python is not already installed on the machine, download and install the latest version from the official Python website: <https://www.python.org/downloads/>

4. **Environment Setup:**

- Open a terminal or command prompt and navigate to the project directory.

- Create a virtual environment (recommended) to isolate project dependencies:

***python -m venv skintellect-env***

  - *Activate the virtual environment*:

- On Windows: `***skintellect-env\Scripts\activate***`

- On macOS and Linux: `***source skintellect-env/bin/activate***`

5. **Install Dependencies:**

- Install required Python packages and libraries using pip:

***pip install Flask paho-mqtt requests numpy opencv-python matplotlib scikit-image mtcnn Adafruit-SSD1306***

API Integration and Configuration:

6. **API Key Acquisition:**

- Obtain API keys and tokens for the following APIs:

- Weatherstack API: For fetching temperature and humidity data.

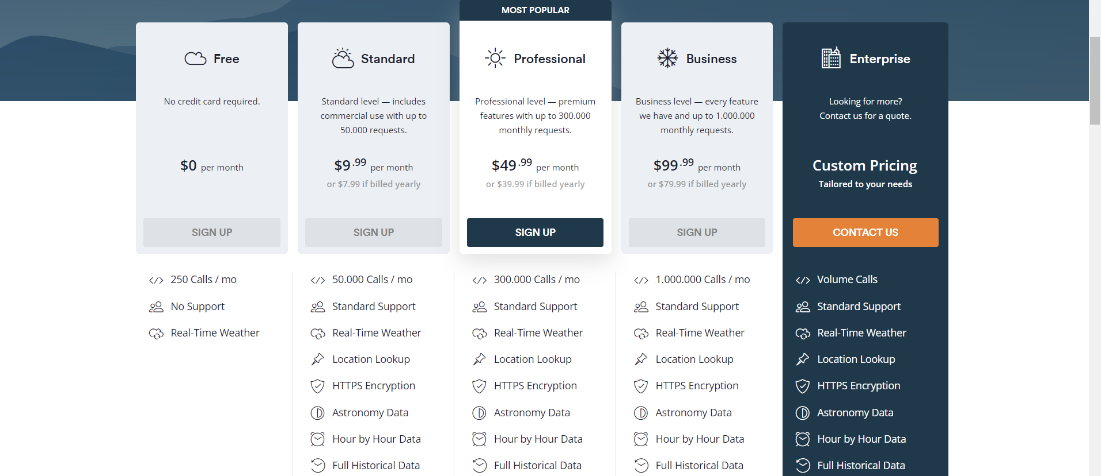
**Steps:**

1. Visit: <https://weatherstack.com/>

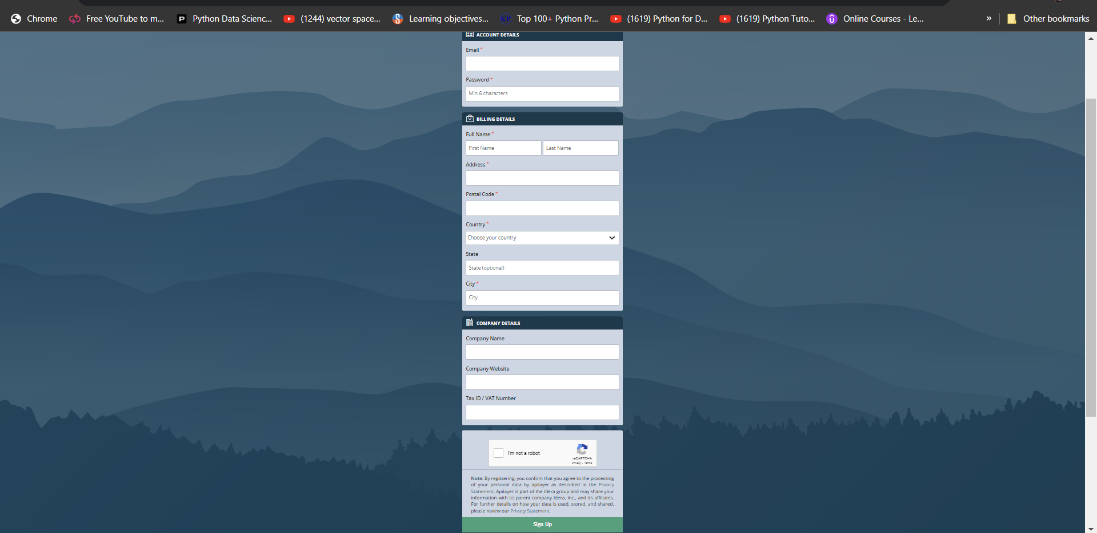
2. Click on "SIGN UP FREE"



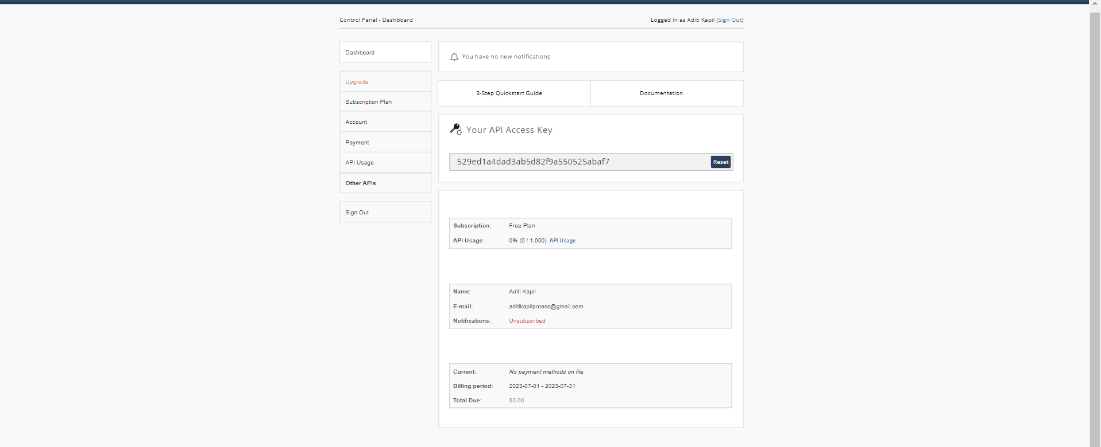
3. Select "SIGN UP FREE ACCOUNT"



4. Navigate to your DASHBOARD



5. Your API ACCESS KEY WILL BE SHOWED THERE



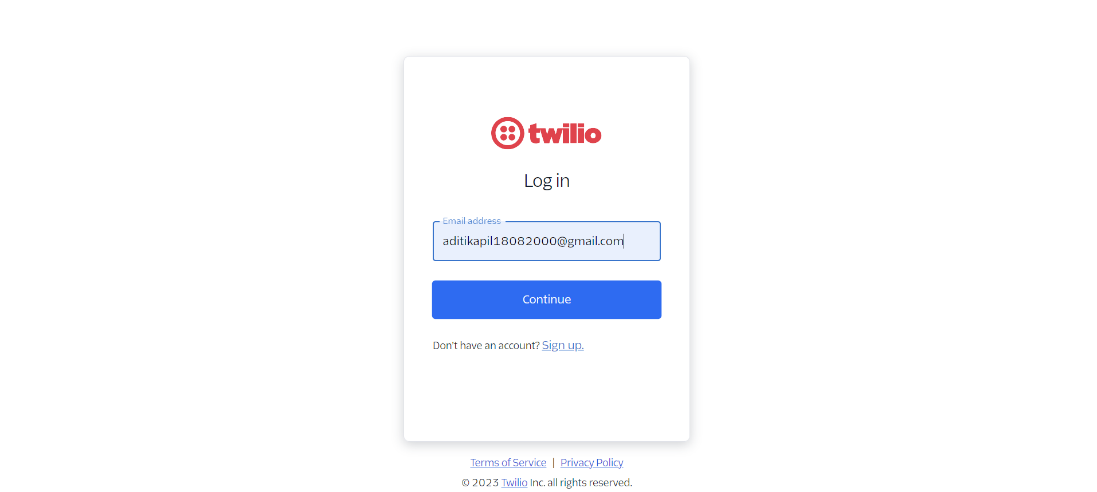
1. Copy it and use it in the python code

- Twilio API: For sending SMS notifications.

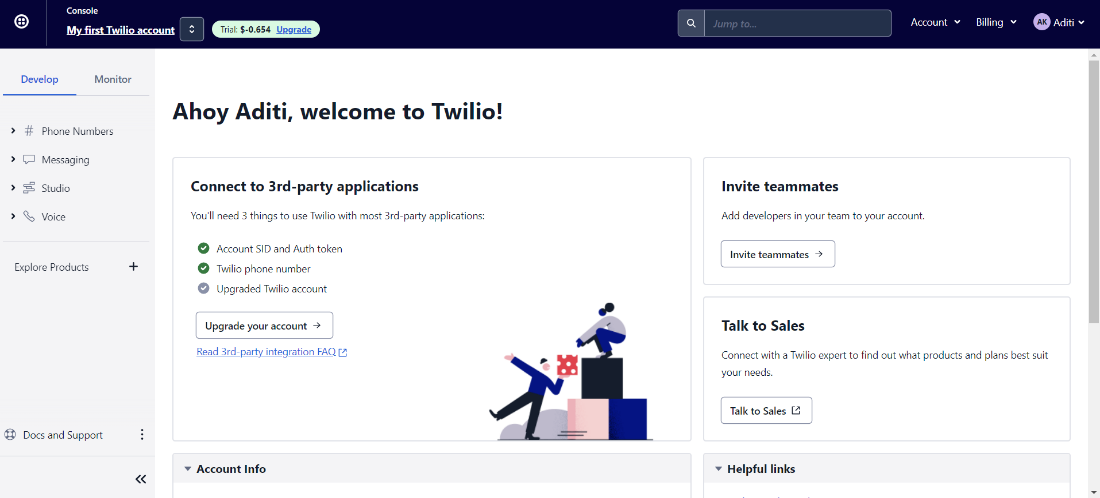
**Steps:**

1. Visit: <https://www.twilio.com/en-us>

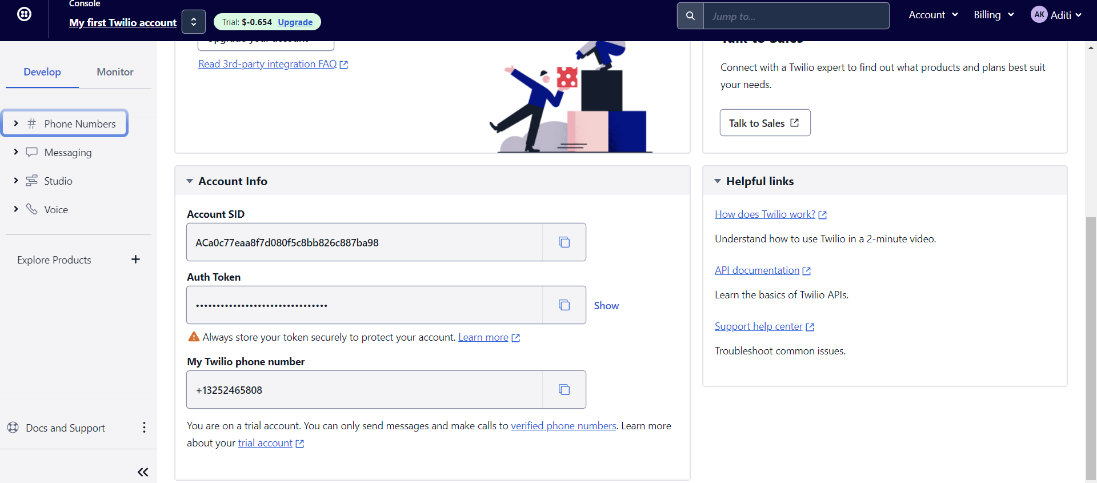
2. Login/sign up



3. Your DASHBOARD will open



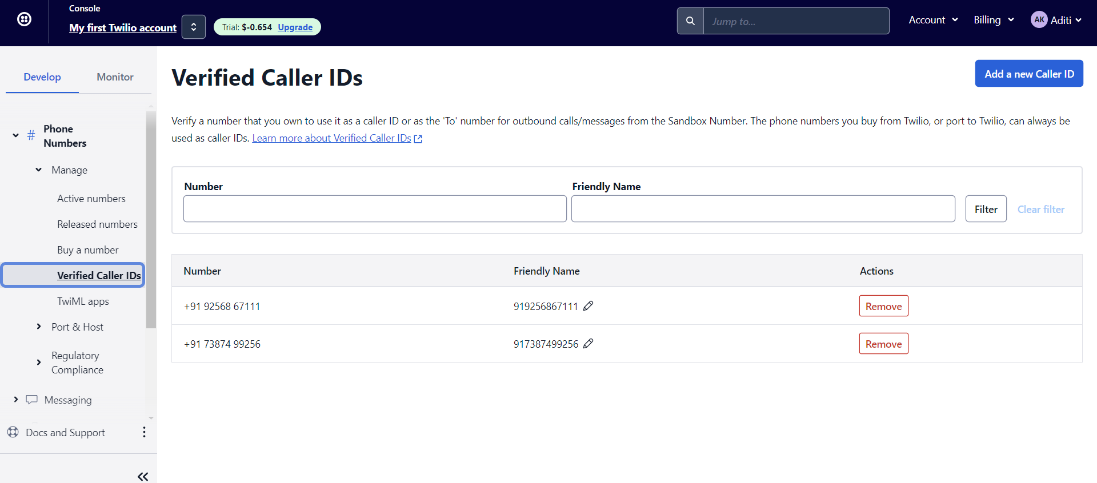
4. Navigate down, you will find account SID, Auth Token, My Twilio phone number



5. Copy and use them in python code for sending SMS

6. Remember, you can only send SMS to registered phone numbers

7. For that, click on " Phone Numbers" in left of DASHBOARD -> MANAGE -> Verified Caller IDs -> Register the numbers you want to send SMS to



- Telegram Bot API: For sending Telegram notifications.

**Steps:**

1. Visit: Your Telegram App

2. Search for "BotFather"

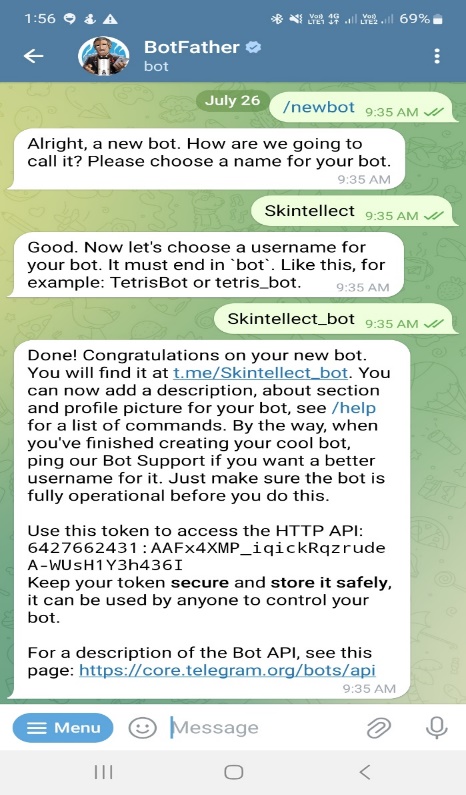
3. send "/newbot" message

4. Enter a name for bot you want to create

5. Choose a username

6. Your bot will be created

7. Copy the API TOKEN given in last message of "BotFather"



8. Start chatting with your bot (send a few messages)

9. Paste the API Token in python code file "getchat\_id", run it

10. It will print the chat id

11. Copy the chat Id and paste it wherever asked

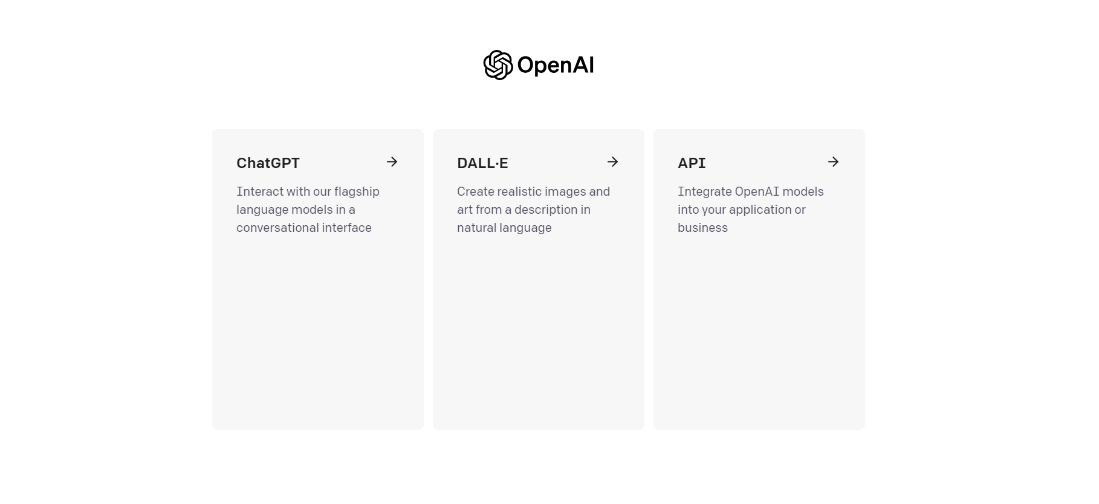
- OpenAI API: For generating personalized skincare recommendations

**Steps:**

1. Visit: <https://openai.com/>

2. Click on Sign in/Log in

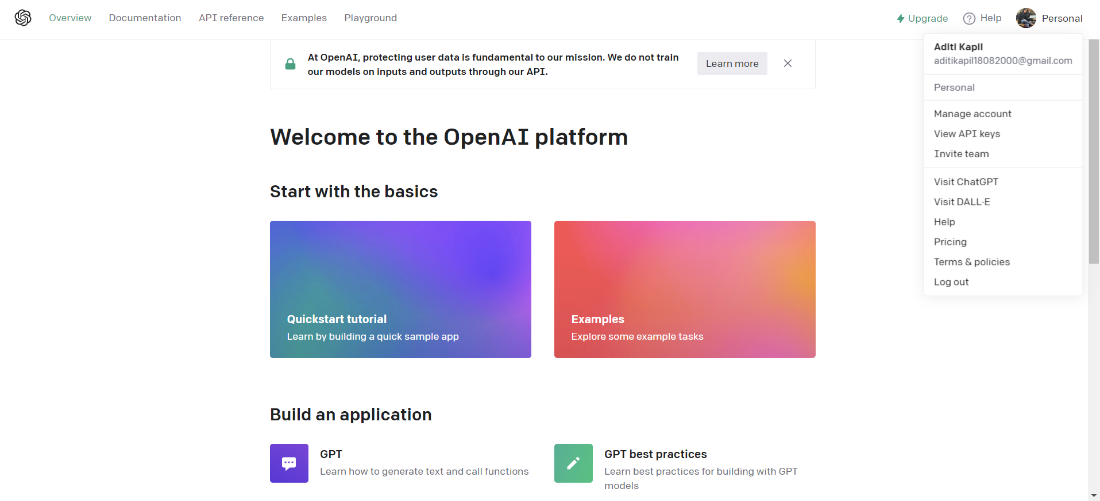
3. Select "API"



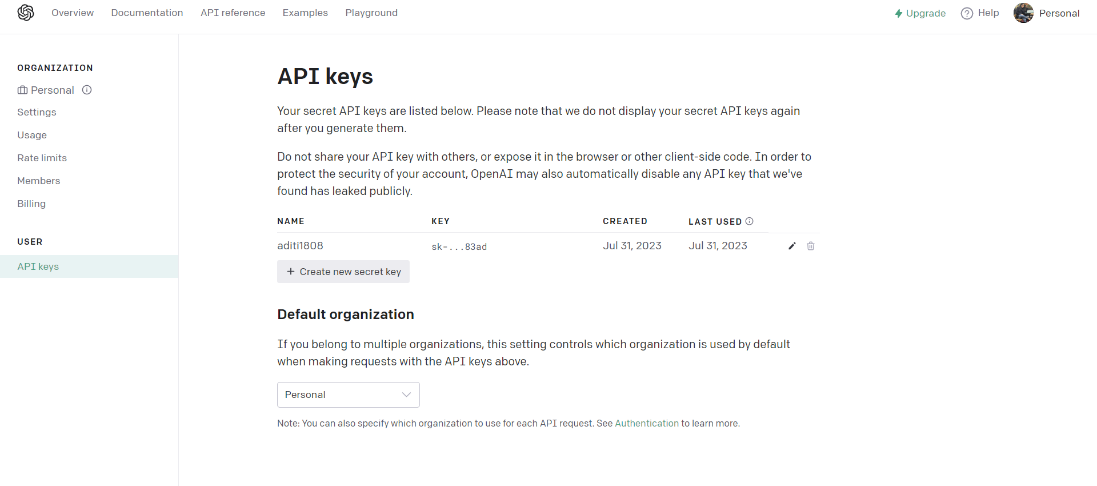
4. Your DASHBOARD will open

5. Click on "PERSONAL"

6. Select "View API keys"



7. Copy API KEY given/ create yours and use it in your code



7. API Key Integration:

- In the relevant code files (`twilio\_module.py`, `telegram\_module.py`, `openai\_module.py`), replace placeholder strings (e.g., 'YOUR\_API\_KEY', 'YOUR\_TELEGRAM\_BOT\_TOKEN') with your actual API keys.

MQTT Broker Setup:

8. **Install and Configure MQTT Broker:**

- If you don't have an MQTT broker, consider using MQTTX or another suitable option.

- Install and configure the MQTT broker according to your machine's specifications.

9**. Update Flask App Code:**

- In the `app.py` file, ensure the MQTT broker's address and port are correctly set in the `broker\_address` and `broker\_port` variables.

Web Frontend and UI Customization:

10. **Frontend Customization**:

- Customize the web frontend using HTML, CSS, and JavaScript files in the `templates`(should be under Python folder) directory.

- Modify the layout, styles, and UI components to match your preferences.

Email Configuration:

12. **Email Settings**:

- If using Gmail for sending emails:

- Replace `sender\_email` and `sender\_password` placeholders in the `email\_module.py` file with your Gmail credentials.

- Adjust the email configuration if you're using a different email provider.

Final Configuration:

13. **Adjust Paths and Parameters**:

- Update paths, filenames, and other parameters in the code to match your machine's directory structure and configuration.

Running the Application:

14. **Start Flask App:**

- Two flask web apps are being used in this project:

**1. FOR ACCESSING SKINTELLECT**

**2. FOR ACCESSING IMAGE UPLOADED**

- Open a terminal or command prompt.

- Navigate to the project directory.

- Activate the virtual environment if you created one.

- Run the Flask apps: ***python app.py****(port number: 5000)* ***and cam\_app.py****(port number: 5001)*

- The apps will start, and you'll see output indicating that it's running, likely something like

"***Running on*** [***http://127.0.0.1:5000/***](http://127.0.0.1:5000/)".

15. **Access the Application:**

- Open a web browser and enter the URL shown in the output (e.g., <http://127.0.0.1:5000/>) to access the SKINTELLECT application.

16. **Use the Application:**

- On the web interface, enter your email address, phone number, gender, and age.

- Interact with the ultrasonic sensor to trigger the display of information on the OLED screen based on user proximity.

- The W5300-MQTT-CAM device will capture your facial image, and the system will process the image to analyse various skin attributes.

- The personalized skincare recommendations will be sent to your email and phone number via SMS. Additionally, a notification will be sent to your Telegram app.

Important Considerations:

- Ensure all APIs, services, and devices are set up and functioning correctly before running the code.

- Implement data handling and security measures to protect user information and comply with relevant data privacy regulations.

By following these comprehensive steps, you can successfully set up and run the SKINTELLECT application on a different machine, complete with the ultrasonic sensor and Adafruit OLED screen integration, hardware setup, software configuration, and API integrations. This detailed guide ensures a seamless process for analysing skin attributes and providing personalized skincare recommendations.

**CODE EXPLANATION**

FILE NAME: index.html

<!DOCTYPE html>

<html>

<head>

<!-- Set the initial viewport configuration -->

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<!-- Set the title of the webpage -->

<title>SKINTELLECT</title>

<!-- Define the CSS styles for the webpage -->

<style>

/\* Styling for the body and header \*/

body {

font-family: Arial, sans-serif;

background-color: #f0f0f0;

margin: 0;

padding: 0;

}

/\* Styling for the header section \*/

header {

background-color: #007bff;

color: #fff;

padding: 20px;

text-align: center;

}

/\* Styling for the main heading \*/

h1 {

margin: 0;

font-size: 48px;

letter-spacing: 2px;

text-transform: uppercase;

font-weight: bold;

text-shadow: 2px 2px 4px rgba(0, 0, 0, 0.3);

}

/\* Styling for the subheading \*/

h2 {

margin: 0;

font-size: 24px;

letter-spacing: 1px;

font-weight: bold;

}

/\* Styling for regular paragraphs \*/

p {

font-size: 18px;

max-width: 600px;

margin: 20px auto;

}

/\* Styling for the main content container \*/

.container {

text-align: center;

}

/\* Styling for the submit button \*/

#submit-btn {

display: block;

margin: 0 auto;

background-color: #007bff;

color: #fff;

border: none;

padding: 12px 20px;

font-size: 16px;

cursor: pointer;

border-radius: 4px;

transition: background-color 0.3s ease;

}

/\* Styling for submit button hover effect \*/

#submit-btn:hover {

background-color: #0056b3;

}

/\* Styling for text areas \*/

textarea {

width: 300px;

height: 50px;

display: block;

margin: 20px auto;

resize: none;

}

/\* Media Queries for responsive design \*/

@media (max-width: 768px) {

/\* Adjust font sizes and padding for mobile view \*/

h1 {

font-size: 36px;

}

h2 {

font-size: 20px;

}

p {

font-size: 16px;

}

header {

padding: 10px;

}

}

/\* Media Query for larger screens, e.g., tablets \*/

@media (min-width: 768px) and (max-width: 1024px) {

/\* Adjust font sizes for tablet view \*/

h1 {

font-size: 42px;

}

h2 {

font-size: 22px;

}

}

</style>

</head>

<body>

<!-- Header section with title and subheading -->

<header>

<h1>SKINTELLECT</h1>

<h2>Intelligent Skin Analysis</h2>

</header>

<!-- Main content container -->

<div class="container">

<!-- Welcome message and instructions for users -->

<p>Welcome to SKINTELLECT!<br>Click the button below to start the skin analysis process.</p>

<!-- Text areas for user input -->

<textarea type="text" id="phone-number" rows="2" cols="40" placeholder="Enter your phone number along with country code"></textarea>

<textarea id="text" rows="2" cols="40" placeholder="Please enter a valid email id"></textarea>

<!-- Additional text areas for gender and age input -->

<textarea type="text" id="gender" rows="2" cols="40" placeholder="Enter your gender"></textarea>

<textarea type="text" id="age" rows="2" cols="40" placeholder="Enter your age"></textarea>

</div>

<!-- Container for the submission button -->

<div>

<!-- Button to trigger data submission -->

<button id="submit-btn" onclick="publishText()">Start Analysis</button>

</div>

<!-- Include jQuery library for AJAX functionality -->

<script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>

<script>

// Function to send user data to the server

function publishText() {

// Get values from input fields

var text = document.getElementById('text').value; // Get email input

var phoneNumber = document.getElementById('phone-number').value; // Get phone number input

var gender = document.getElementById('gender').value; // Get gender input

var age = document.getElementById('age').value; // Get age input

// AJAX request to the server

$.ajax({

url: '/mqtt-publish', // Server endpoint

type: 'POST', // HTTP method

data: {

text: text,

'phone-number': phoneNumber,

gender: gender, // Send gender

age: age // Send age

},

success: function(response) { // On success

console.log(response); // Log server response

showAlert('Note: Please stand closer to the camera to capture a picture of your face'); // Show alert message

},

error: function(error) { // On error

console.error(error); // Log error

}

});

}

// Function to display an alert message

function showAlert(message) {

alert(message); // Show alert with the provided message

}

</script>

</body>

</html>

WHAT THIS FILE CONTAINS (IN SHORT):

The HTML code establishes the user interface for the SKINTELLECT web application. It features a responsive design with a dynamic header displaying the title "SKINTELLECT" and a subheading "Intelligent Skin Analysis." Users are prompted to input their email, phone number, gender, and age. Upon clicking the "Start Analysis" button, the data is sent via AJAX to the Flask app. The code also incorporates media queries for varied screen sizes and includes explanatory comments throughout the code for easy understanding of its components and functionalities.

 FILE NAME: app.py

# Import required libraries

from flask import Flask, request, jsonify # Import Flask framework and other modules

import paho.mqtt.client as mqtt # Import MQTT client library

import WrinkleDetection # Import your custom WrinkleDetection module

# Create a Flask app instance

app = Flask(\_\_name\_\_) # Initialize the Flask app

# Initialize variables to store data

stored\_text = "" # Variable to store entered text

stored\_phone\_number = "" # Variable to store phone number

stored\_age = "" # Variable to store age

stored\_gender = "" # Variable to store gender

response\_text = "" # Variable to store MQTT response

# MQTT broker configuration

broker\_address = "10.21.70.16" # MQTT broker's IP address

broker\_port = 1883 # MQTT broker's port

mqtt\_topic = "MQTT-CAM" # Topic for capturing images

response\_topic = "CAM-MQTT" # Topic for receiving image analysis results

# Callback function when the client connects to the broker

def on\_connect(client, userdata, flags, rc):

print("Connected with result code " + str(rc)) # Print connection result code

# Subscribe to the response topic

client.subscribe(response\_topic) # Subscribe to MQTT response topic

# Callback function when a message is received from the broker

def on\_message(client, userdata, msg):

global response\_text # Use the global response\_text variable

if msg.topic == response\_topic:

new\_response\_text = msg.payload.decode('utf-8') # Decode MQTT message payload

if new\_response\_text.startswith("http://"):

response\_text = new\_response\_text

# Process captured image using WrinkleDetection functions

finalmailid = WrinkleDetection.image\_save(stored\_text, response\_text)

image\_url = response\_text

save\_path = 'C:/Users/Aditi/OneDrive/Desktop/SkinTellect/'

WrinkleDetection.save\_image\_from\_url(image\_url, save\_path)

wrinkles = WrinkleDetection.detect()

tone = WrinkleDetection.skin\_tone\_analysis()

contrastt = WrinkleDetection.skin\_texture\_analysis()

pig = WrinkleDetection.pigmentation\_analysis()

WrinkleDetection.letdo(finalmailid, stored\_phone\_number, wrinkles, tone, contrastt, pig, stored\_age, stored\_gender)

# Create an MQTT client instance

client = mqtt.Client() # Create MQTT client instance

# Set the callback functions

client.on\_connect = on\_connect # Set on\_connect callback

client.on\_message = on\_message # Set on\_message callback

# Connect to the broker

client.connect(broker\_address, broker\_port) # Connect to MQTT broker

# Start the MQTT loop to handle incoming messages in the background

client.loop\_start() # Start MQTT loop

# Define routes for the Flask app

@app.route('/') # Define route for the root URL

def index():

# Read and return the HTML file

with open('templates/index.html') as f:

return f.read() # Read and serve HTML file

@app.route('/mqtt-publish', methods=['POST']) # Define route for MQTT publish

def mqtt\_publish():

global stored\_text, stored\_phone\_number, stored\_age, stored\_gender # Use global variables

text = request.form.get('text') # Get 'text' field from form data

phone\_number = request.form.get('phone-number') # Get 'phone-number' field from form data

stored\_text = text # Store entered text

stored\_phone\_number = phone\_number # Store phone number

stored\_age = request.form.get('age') # Store age from form data

stored\_gender = request.form.get('gender') # Store gender from form data

# Publish MQTT message to trigger image capture

client.publish(mqtt\_topic, "cmd:capture") # Publish MQTT message

return jsonify({'message': 'Text, phone number, age, and gender saved successfully'}) # Return JSON response

@app.route('/get-stored-text', methods=['GET']) # Define route to get stored text

def get\_stored\_text():

global stored\_text # Use global stored\_text variable

return jsonify({'stored\_text': stored\_text}) # Return stored\_text in JSON response

@app.route('/get-response-text', methods=['GET']) # Define route to get response text

def get\_response\_text():

global response\_text # Use global response\_text variable

return jsonify({'response\_text': response\_text}) # Return response\_text in JSON response

# Run the Flask app

if \_\_name\_\_ == '\_\_main\_\_':

app.run(host="0.0.0.0", debug=True) # Run the app on 0.0.0.0 (all network interfaces) with debugging enabled

WHAT THIS FILE CONTAINS (IN SHORT):

This code is the backbone of the SKINTELLECT application, leveraging Flask and MQTT to capture facial images, analyze skin attributes, and provide personalized skincare recommendations. It establishes MQTT communication with a camera device, handles user inputs through Flask routes, and employs the WrinkleDetection module for image analysis. The app's core functionality lies in capturing user data, triggering image capture, processing the images, and delivering the analysis results, thereby offering users insightful skincare recommendations.

 FILE NAME: wrinkleDetection.py

# Import required libraries and modules

import cv2 # OpenCV for image processing

import numpy as np # NumPy for numerical operations

from matplotlib import pyplot as plt # Matplotlib for visualization

from IPython.display import display, Image # IPython display for showing images

import requests # Requests library for making HTTP requests

import os # OS module for operating system-related functions

# Import the MTCNN face detection model

from mtcnn.mtcnn import MTCNN

# Import the 'combine' module for further processing

import combine

# Initialize variables to store analysis results

wrinkles = ""

tone = ""

contrastt = ""

diss = ""

pig = ""

# Initialize variables to store input data

stored\_text = ""

# Function to save image path and response text

def image\_save(stored\_text2, response\_text2):

global stored\_text # Use the global variable

stored\_text = stored\_text2 # Store the input data

return stored\_text # Return the stored data

# Initialize variables for image URL and save path

image\_url = ""

save\_path = ""

# Function to save an image from a given URL to a specified path

def save\_image\_from\_url(image\_url, save\_path):

try:

# Send a GET request to the image URL

response = requests.get(image\_url)

response.raise\_for\_status() # Check for any errors in the request

# Get the file name from the URL

file\_name = os.path.basename(image\_url)

# Combine the file name with the save\_path to create the complete file path

file\_path = os.path.join(save\_path, "W5300-MQTT-CAM\_pic.jpg")

# Save the image to the specified location

with open(file\_path, 'wb') as file:

file.write(response.content)

return True # Image saved successfully

except Exception as e:

print(f"Error occurred: {e}")

return False # Image saving failed

# Load an image using OpenCV

img = cv2.imread("C:/Users/Aditi/OneDrive/Desktop/SkinTellect/W5300-MQTT-CAM\_pic.jpg")

# Function to detect wrinkles on a face using edge detection

def detect():

# Load the pre-trained MTCNN face detector

detector = MTCNN()

# Detect faces in the image using MTCNN

faces = detector.detect\_faces(img)

# Check if any faces were detected

if len(faces) > 0:

for face in faces:

x, y, w, h = face['box']

# Crop the face region

face\_region = img[y:y+h, x:x+w]

# Convert the face region to grayscale

gray\_face = cv2.cvtColor(face\_region, cv2.COLOR\_BGR2GRAY)

# Apply Canny edge detection on the grayscale face region

edges = cv2.Canny(gray\_face, 130, 1000)

# Count the number of edges in the cropped face region

num\_edges = np.count\_nonzero(edges)

if num\_edges > 1000:

return "wrinkles found" # Wrinkles detected

else:

return "wrinkles not found" # No wrinkles detected

else:

return "No face detected" # No face detected in the image

# Function to analyze skin tone based on LAB color space

def skin\_tone\_analysis():

# Convert the image from BGR to RGB color space

img\_rgb = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)

# Convert the RGB image to LAB color space

img\_lab = color.rgb2lab(img\_rgb)

# Extract the L\*, a\*, and b\* channels

l\_channel, a\_channel, b\_channel = cv2.split(img\_lab)

# Calculate the mean a\* and b\* values

mean\_a = np.mean(a\_channel)

mean\_b = np.mean(b\_channel)

# Calculate the skin tone index (STI)

sti = np.sqrt(mean\_a \*\* 2 + mean\_b \*\* 2)

# Determine the skin tone category based on the STI value

if sti < 20:

skin\_tone\_category = "Dark skin"

elif sti < 40:

skin\_tone\_category = "Medium skin"

else:

skin\_tone\_category = "Light skin"

return skin\_tone\_category

# Function to analyze skin texture using GLCM properties

def skin\_texture\_analysis():

# Convert the image to grayscale

gray\_img = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

# Calculate the GLCM with specific properties (contrast and dissimilarity)

glcm = greycomatrix(gray\_img, [1], [0], 256, symmetric=True, normed=True)

contrast = greycoprops(glcm, 'contrast')[0, 0]

dissimilarity = greycoprops(glcm, 'dissimilarity')[0, 0]

return contrast, dissimilarity # Return the texture analysis results

# Function to analyze pigmentation in the image

def pigmentation\_analysis():

# Convert the image from BGR to HSV color space

img\_hsv = cv2.cvtColor(img, cv2.COLOR\_BGR2HSV)

# Define the lower and upper bounds of pigmented color in HSV

lower\_bound = np.array([0, 50, 50]) # Lower bound for hue (red)

upper\_bound = np.array([20, 255, 255]) # Upper bound for hue (yellow)

# Create a mask to isolate pigmented regions

mask = cv2.inRange(img\_hsv, lower\_bound, upper\_bound)

# Count the number of pigmented pixels

pigmented\_pixels = np.sum(mask == 255)

# Calculate the total number of pixels in the image

total\_pixels = mask.shape[0] \* mask.shape[1]

# Calculate the percentage of pigmented pixels in the image

pigmentation\_percentage = (pigmented\_pixels / total\_pixels) \* 100

return pigmentation\_percentage

# Main function to perform all analyses and call 'combine' module

def letdo(finalmailid, ph\_no, wrinkles, tone, contrastt, pig, age, gender):

# Determine skin texture properties

contrast, dissimilarity = skin\_texture\_analysis()

# Determine skin tone category

skin\_tone = skin\_tone\_analysis()

# Check and display if wrinkles are detected

wrinkle\_result = detect()

# Call 'combine' module to process and combine results

combine.var(finalmailid, ph\_no, wrinkle\_result, skin\_tone, contrast, pig, age, gender)

WHAT THIS FILE CONTAINS (IN SHORT):

The code is a Python script that employs OpenCV, NumPy, and other libraries for analyzing facial images in various ways. It includes functions to detect wrinkles on a face using edge detection, assess skin tone through LAB color space analysis, evaluate skin texture using GLCM properties, and identify pigmentation in the image. The script also interacts with the 'combine' module for further processing and analysis of the collected data. The code's modularity and well-commented sections make it easy to understand and modify for tailored image analysis applications.

FILE NAME: combine.py

import temp # Importing the 'temp' module

import recommend # Importing the 'recommend' module

# Define a function named 'var' that takes various parameters

def var(finalmailid, ph\_no, wrinkles, tone, contrastt, pig, age, gender):

result\_text = "" # Initialize an empty string to store the result text

final\_mail\_id = finalmailid # Assign the provided email ID to the variable 'final\_mail\_id'

gender = gender # Assign the provided gender to the variable 'gender'

age = age # Assign the provided age to the variable 'age'

# Append the values with labels to the 'result\_text' variable

result\_text += "Gender: " + str(gender) + "\n"

result\_text += "Age: " + str(age) + "\n"

result\_text += "Wrinkles: " + str(wrinkles) + "\n"

result\_text += "Skin Tone: " + str(tone) + "\n"

result\_text += "Skin Contrast: " + str(contrastt) + "\n"

result\_text += "Skin Pigmentation: " + str(pig) + "\n"

# Call the 'temp.temp\_hum' function from the 'temp' module

# Pass the 'result\_text', 'final\_mail\_id', and 'ph\_no' as parameters

temp.temp\_hum(result\_text, final\_mail\_id, ph\_no)

WHAT THIS FILE CONTAINS (IN SHORT):

This code defines a function named 'var' that takes several parameters, including the user's email ID, phone number, analysis results (wrinkles, skin tone, contrast, pigmentation, age, and gender). It assembles these values with appropriate labels into a text string called 'result\_text'. The function then calls another function 'temp.temp\_hum' from the 'temp' module, passing the assembled result text, final email ID, and phone number as arguments. This function likely involves temperature and humidity handling, but specific details would be found in the 'temp' module.

FILE NAME: temp.py

import recommend # Importing the 'recommend' module

# Define a function named 'temp\_hum' that takes three parameters

def temp\_hum(result\_text, final\_mail\_id, ph\_no):

import requests # Importing the 'requests' library for making API requests

# Replace 'YOUR\_API\_KEY' with your actual Weatherstack API key

api\_key = '529ed1a4dad3ab5d82f9a550525abaf7'

# Replace 'YOUR\_CITY' with the city name for which you want to get weather data

city = 'pune' # Assuming it is installed in Pune

# API endpoint URL with API key and city

url = f'http://api.weatherstack.com/current?access\_key={api\_key}&query={city}'

# Send API request and get response data

response = requests.get(url)

data = response.json() # Parse the response JSON data

# Check if the API request was successful

if response.status\_code == 200:

# Extract temperature and humidity data from the response

temperature = data['current']['temperature']

humidity = data['current']['humidity']

# Append temperature and humidity information to the 'result\_text' variable

result\_text += "Temperature of place: " + str(temperature) + "\n"

result\_text += "Humidity of place: " + str(humidity) + "\n"

# Call the 'recommend.recommend\_chat' function from the 'recommend' module

# Pass the 'result\_text', 'final\_mail\_id', and 'ph\_no' as parameters

recommend.recommend\_chat(result\_text, final\_mail\_id, ph\_no)

else:

print('Failed to fetch weather data.')

WHAT THIS FILE CONTAINS (IN SHORT):

This code defines a function named 'temp\_hum' that takes three parameters: 'result\_text', 'final\_mail\_id', and 'ph\_no'. It uses the 'requests' library to make an API request to the Weatherstack API to fetch current weather data for a specified city (in this case, Pune). If the API request is successful (status code 200), it extracts the temperature and humidity information from the response and appends it to the 'result\_text' variable. Then, it calls the 'recommend.recommend\_chat' function from the 'recommend' module, passing the assembled result text, final email ID, and phone number as arguments. If the API request fails, it prints a failure message.

FILE NAME: recommend.py

import openai # Importing the 'openai' library

import json # Importing the 'json' module

import mailsend # Importing the 'mailsend' module

import smsSend # Importing the 'smsSend' module

# Define a function named 'recommend\_chat' that takes three parameters

def recommend\_chat(result\_text, final\_mail\_id, ph\_no):

openai.api\_key = "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" # Set OpenAI API key

# Declare the 'chat' variable to hold the input text for generating suggestions

chat = result\_text

# Create a completion request to the OpenAI API

respond = openai.Completion.create(

model="text-davinci-003", # Specify the language model to use

prompt="""Take the chat and Based on the provided values, give detailed personalized skin care suggestions:

- suggest recommendation based on if wrinkles are found or not

- suggest recommendation based on temperature and humidity of place user lives in

- suggest recommendation based on contrast of skin whether low, medium, high

- suggest recommendation based on pigmentation of skin found low, medium, high

- suggestions should be purely based on temperature and humidity taking into consideration

Text: """ + chat + """

""",

temperature=0, # Set the temperature for randomness (0 means deterministic)

max\_tokens=1000, # Set the maximum number of tokens in the response

top\_p=1, # Set the nucleus sampling probability

frequency\_penalty=0.2, # Set the frequency penalty

presence\_penalty=0 # Set the presence penalty

)

# Extract the generated text from the API response

generated\_text = respond.choices[0].text.strip()

# Display the generated text as output

print(generated\_text)

# Call the 'send\_mailToCustomer' function from the 'mailsend' module

# Pass the 'final\_mail\_id', 'result\_text', and 'generated\_text' as parameters

mailsend.send\_mailToCustomer(final\_mail\_id, result\_text, generated\_text)

# Call the 'send\_sms' function from the 'smsSend' module

# Pass the 'ph\_no', 'result\_text', and 'generated\_text' as parameters

smsSend.send\_sms(ph\_no, result\_text, generated\_text)

WHAT THIS FILE CONTAINS (IN SHORT):

This code defines a function named 'recommend\_chat' that takes three parameters: 'result\_text', 'final\_mail\_id', and 'ph\_no'. Inside the function, it sets the OpenAI API key, creates a completion request to the OpenAI API using the 'openai.Completion.create' method, and provides a prompt containing the input 'chat'. The generated text is extracted from the API response and displayed. Then, the 'send\_mailToCustomer' function from the 'mailsend' module is called, passing the final email ID, result text, and generated text as arguments. Similarly, the 'send\_sms' function from the 'smsSend' module is called, passing the phone number, result text, and generated text as arguments.

FILE NAME: smsSend.py

def send\_sms(ph\_no, result\_text, recommendation):

# Import the 'Client' class from the 'twilio.rest' module

from twilio.rest import Client

# Concatenate the result text and recommendation for the SMS body

final\_text = result\_text + "\n" + "\n" + recommendation

# Set your Twilio Account SID and Auth Token

SID = 'ACa0c77eaa8f7d080f5c8bb826c887ba98'

AUTH\_TOKEN = 'd0effba4995c97f648caa1bc97e316a0'

# Create a Twilio Client instance using your SID and Auth Token

cl = Client(SID, AUTH\_TOKEN)

# Send an SMS using the 'messages.create' method of the Twilio Client

cl.messages.create(body=final\_text, from\_='+13252465808', to=ph\_no)

# Print a success message

print("SMS SENT SUCCESSFULLY ! ")

WHAT THIS FILE CONTAINS (IN SHORT):

This code defines a function named 'send\_sms' that takes three parameters: 'ph\_no' (phone number), 'result\_text', and 'recommendation'. Inside the function, it imports the 'Client' class from the 'twilio.rest' module. Then, it concatenates the 'result\_text' and 'recommendation' to form the body of the SMS. The Twilio Account SID and Auth Token are set as variables. A Twilio Client instance is created using the SID and Auth Token. The 'messages.create' method of the Twilio Client is used to send an SMS, specifying the body, sender's phone number, and recipient's phone number ('ph\_no'). Finally, a success message is printed.

FILE NAME: mailsend.py

import smtplib

from email.mime.text import MIMEText

from email.mime.multipart import MIMEMultipart

from google.oauth2.credentials import Credentials

from google\_auth\_oauthlib.flow import InstalledAppFlow

from email.mime.image import MIMEImage

import requests

# Telegram Bot token and chat ID for sending notifications

telegram\_bot\_token = '6427662431:AAFx4XMP\_iqickRqzrudeA-WUsH1Y3h436I'

telegram\_chat\_id = '6119638322'

def send\_telegram\_notification(message):

# Construct the Telegram API URL

telegram\_api\_url = f"https://api.telegram.org/bot{telegram\_bot\_token}/sendMessage"

# Data for sending the message

data = {

"chat\_id": telegram\_chat\_id,

"text": message,

}

# Send the message using the requests library

response = requests.post(telegram\_api\_url, data=data)

return response.json()

def send\_mailToCustomer(stored\_text, found, recommendations):

# Email sender's credentials

sender\_email = "aditi.kapil@msds.christuniversity.in"

sender\_password = "ornimint123"

# Receiver's email is passed as 'stored\_text'

receiver\_email = stored\_text

# Email subject and body

subject = "SKINTELLECT 📑 Your report is ready report !!! "

body = f"{found}\n\n{recommendations}"

# Path to the image to be attached

image\_path = 'W5300-MQTT-CAM\_pic.jpg'

# Create a message container (multipart/alternative represents text and HTML formats)

message = MIMEMultipart()

message["From"] = sender\_email

message["To"] = receiver\_email

message["Subject"] = subject

# Attach the plain text body to the message

message.attach(MIMEText(body, 'plain'))

# Attach the image as an attachment to the message

with open(image\_path, 'rb') as fp:

img = MIMEImage(fp.read())

img.add\_header('Content-Disposition', 'attachment', filename=image\_path)

message.attach(img)

try:

# Connect to the Gmail SMTP server securely

server = smtplib.SMTP\_SSL('smtp.gmail.com', 465)

# Log in to the sender's email account

server.login(sender\_email, sender\_password)

# Send the email

server.sendmail(sender\_email, receiver\_email, message.as\_string())

print("Email sent successfully!")

# Send a Telegram notification

send\_telegram\_notification("SKINTELLECT 📑 Another report generated!")

except Exception as e:

print("Error: ", e)

finally:

# Close the connection to the server

server.quit()

WHAT THIS FILE CONTAINS (IN SHORT):

This code defines the send\_mailToCustomer function that takes stored\_text, found, and recommendations as input. Inside the function, it uses the Gmail SMTP server to send an email with the provided data. It also attaches the image 'W5300-MQTT-CAM\_pic.jpg'. The send\_telegram\_notification function is used to send a notification to a Telegram chat using the provided telegram\_bot\_token and telegram\_chat\_id.

FILE NAME: botchatid.py

import requests

# Replace 'YOUR\_TELEGRAM\_BOT\_TOKEN' with your actual Telegram bot token

telegram\_bot\_token = '6427662431:AAFx4XMP\_iqickRqzrudeA-WUsH1Y3h436I'

def get\_chat\_id():

# Construct the Telegram API URL for getting updates

telegram\_api\_url = f"https://api.telegram.org/bot{telegram\_bot\_token}/getUpdates"

# Send a GET request to the Telegram API

response = requests.get(telegram\_api\_url)

# Parse the JSON response

data = response.json()

if "result" in data and data["result"]:

# Assuming the latest message is the most recent one, get the chat ID from it

chat\_id = data["result"][-1]["message"]["chat"]["id"]

return chat\_id

else:

raise ValueError("Failed to retrieve chat ID. Make sure your bot has received messages.")

if \_\_name\_\_ == "\_\_main\_\_":

try:

# Call the function to get the chat ID

chat\_id = get\_chat\_id()

print("Chat ID:", chat\_id)

except Exception as e:

print("Error:", e)

WHAT THIS FILE CONTAINS (IN SHORT):

This code defines a get\_chat\_id function that retrieves the chat ID from the latest message received by the Telegram bot. It constructs the Telegram API URL using the provided telegram\_bot\_token, sends a GET request, and then parses the JSON response to extract the chat ID. The extracted chat ID is printed in the if \_\_name\_\_ == "\_\_main\_\_": block when the script is run.

FILE NAME: skintellect2.ino

#include <Ethernet.h>

#include <Wire.h>

#include <SoftWire.h>

// MQTT [[

#include <PubSubClient.h>

// ]]

#include <ArduinoJson.h>

#include "HardwareSerial.h"

#include <Adafruit\_GFX.h>

#include <Adafruit\_SSD1306.h>

#include "qrcodegen.h"

const char\* device\_unique\_id = "ArdCam1\_" \_\_DATE\_\_ "\_" \_\_TIME\_\_;

// #define SERIAL\_OUTPUT

#ifdef SERIAL\_OUTPUT

#define PRINT(x) Serial.print(x)

#define PRINTLN(x) Serial.println(x)

#else

#define PRINT(x)

#define PRINTLN(x)

#endif

#define SYNC 0x96

int trigPin = 11; // TRIG pin

int echoPin = 12;

float duration\_us, distance\_cm;

#define PIN0\_SDA 14

#define PIN0\_SCL 15

#define SCREEN\_WIDTH 128

#define SCREEN\_HEIGHT 64

#define OLED\_RESET -1

#define SCREEN\_ADDRESS 0x3C

Adafruit\_SSD1306 display(SCREEN\_WIDTH, SCREEN\_HEIGHT, &Wire, OLED\_RESET);

// Enter a MAC address and IP address for your controller below.

// The IP address will be dependent on your local network:

byte mac[] = {

0xDE, 0xAD, 0xBE, 0xEF, 5, 0xF7

};

IPAddress ip(192, 168, 0, 77);

//IPAddress ip(10, 5, 15, 109);

// Enter the IP address of the server you're connecting to:

//IPAddress http\_server(192, 168, 0, 107);

//IPAddress http\_server(10, 21, 70, 16);

const char \*http\_server = "10.21.70.16";

IPAddress myDns(192, 168, 0, 1);

uint16\_t port = 5000;

// Initialize the Ethernet client library

// with the IP address and port of the server

// that you want to connect to (port 23 is default for telnet;

// if you're using Processing's ChatServer, use port 10002):

#define max\_transfer 1024

#define max\_buffer (100 \* max\_transfer)

byte img\_buf[max\_buffer];

EthernetClient client;

//[ MQTT

IPAddress mqtt\_server(10, 21, 70, 16);

// IPAddress mqtt\_server(44, 195, 202, 69);

EthernetClient mqttClient;

PubSubClient mqtt\_client(mqttClient);

bool capture\_requested = 0;

//]

//===================================================================================================

// HTTP POST

int count = 0;

char c = 0;

void client\_write\_large(byte \*bptr, size\_t len) {

size\_t sent = 0;

for (; sent + max\_transfer < len; ) {

client.write(bptr, max\_transfer);

sent += max\_transfer;

bptr += max\_transfer;

//PRINTLN(max\_transfer);

}

client.write(bptr, len - sent);

PRINT("Sent: "); PRINTLN(len);

}

void httpPostForm(byte \*imageData, uint32\_t imageSize) {

String textData = "OpenMVCam1"; // Replace with your text data

// Prepare the POST request body

String requestBody = "";

// Append the text data

requestBody += "--ArduinoBoundary\_OpenMVCam1\r\n";

requestBody += "Content-Disposition: form-data; name=\"text\"\r\n\r\n";

requestBody += textData;

// Append the image data

requestBody += "\r\n--ArduinoBoundary\_OpenMVCam1\r\n";

requestBody += "Content-Disposition: form-data; name=\"image\"; filename=\"image.jpg\"\r\n\r\n";

//requestBody.append(imageData, imageSize);

// Append the closing boundary

String requestBodyEnd = "";

requestBodyEnd += "\r\n--ArduinoBoundary\_OpenMVCam1--\r\n";

// Prepare the POST request headers

String requestHeaders = "POST /upload HTTP/1.1\r\n";

requestHeaders += "Host: ";

requestHeaders += http\_server;

requestHeaders += ":";

requestHeaders += port;

requestHeaders += "\r\n";

requestHeaders += "Content-Type: multipart/form-data; boundary=ArduinoBoundary\_OpenMVCam1\r\n";

requestHeaders += "Connection: close\r\n";

requestHeaders += "Content-Length: " + String(requestBody.length()+imageSize+requestBodyEnd.length()) + "\r\n\r\n";

// Send the POST request headers

client.print(requestHeaders);

// Send the POST request body

client.print(requestBody);

client\_write\_large(imageData, imageSize);

client.print(requestBodyEnd);

client.flush();

delay(100);

}

void http\_postData(byte \*buf, uint32\_t length) {

count = 0;

// if the server's disconnected, reconnect the client:

while (!client.connected()) {

PRINTLN();

PRINTLN("disconnected. Reconnecting...");

if (client.connect(http\_server, port)) {

PRINTLN("connected");

break;

} else {

// if you didn't get a connection to the server:

if (++count > 5) { // Retry 5 times.

PRINTLN("HTTP Post failed. Give up.");

return;

}

PRINTLN("connection failed");

delay(1000);

}

}

delay(200);

httpPostForm(buf, length);

int len = client.available();

String msg = "HTTP Response: ";

msg += len;

//mqtt\_client.publish("W5300-MQTT", msg.c\_str());

if (len > 0) {

byte buffer[500];

if (len > 500) len = 500;

int recvlen = 0;

recvlen = client.read(buffer+recvlen, len);

if (recvlen < len) {

delay(10);

recvlen += client.read(buffer+recvlen, len-recvlen);

}

String msg = "HTTP Received: ";

msg += recvlen;

//mqtt\_client.publish("W5300-MQTT", msg.c\_str());

//Serial.write(buffer, len); // show in the serial monitor (slows some boards)

//PRINTLN("");

//byteCount = byteCount + len;

byte prev\_char = 0;

String response = "";

int index = 0;

for (index=0; index<len; ++index) {

if (buffer[index] == '\n') {

if (prev\_char == '\n')

break;

} else if (buffer[index] == '\r')

continue;

prev\_char = buffer[index];

}

for (; index<len; ++index)

response += (char)buffer[index];

response.trim();

PRINTLN("Contents: " + response);

mqtt\_client.publish("W5300-MQTT", response.c\_str());

}

client.stop();

}

//===================================================================================================

// MQTT [[

void callback(char\* topic, byte\* payload, unsigned int length) {

PRINT(">>>>>>>>>>> Message arrived [");

PRINT(topic);

PRINT("] ");

String cmd = "";

for (int i=0;i<length;i++) {

cmd += (char)payload[i];

PRINT((char)payload[i]);

}

PRINTLN();

if (cmd == "cmd:capture") {

capture\_requested = 1;

PRINTLN("Capture requested!!!");

}

}

void reconnect() {

// Loop until we're reconnected

while (!mqtt\_client.connected()) {

PRINT("MQTT: Attempting MQTT connection...");

// Attempt to connect

if (mqtt\_client.connect(device\_unique\_id)) {

PRINTLN("MQTT: connected");

mqtt\_client.publish("W5300-MQTT", "Ready");

// ... and resubscribe

mqtt\_client.subscribe("MQTT-W5300");

} else {

PRINT("MQTT: failed, rc=");

PRINT(mqtt\_client.state());

PRINTLN(" MQTT: try again in 5 seconds");

// Wait 5 seconds before retrying

delay(2000);

}

}

}

// ]]

//===================================================================================================

void setup() {

// Open serial communications and wait for port to open:

Serial3.setRx(PC11);

Serial3.setTx(PC10);

delay(50);

// Open serial communications and wait for port to open:

#ifdef SERIAL\_OUTPUT

Serial.begin(9600);

#else

Serial.setRx(0);

Serial.setTx(1);

//Serial.begin(1000000);

Serial.begin(500000);

//Serial.begin(38400);

//Serial.begin(19200);

delay(50);

#endif

pinMode(trigPin, OUTPUT);

pinMode(echoPin, INPUT);

while (!Serial) {

; // wait for serial port to connect. Needed for native USB port only

}

// start the Ethernet connection:

PRINTLN("Initialize Ethernet with DHCP:");

if (Ethernet.begin(mac) == 0) {

PRINTLN("Failed to configure Ethernet using DHCP");

// Check for Ethernet hardware present

if (Ethernet.hardwareStatus() == EthernetNoHardware) {

PRINTLN("Ethernet shield was not found. Sorry, can't run without hardware. :(");

while (true) {

delay(1); // do nothing, no point running without Ethernet hardware

}

}

if (Ethernet.linkStatus() == LinkOFF) {

PRINTLN("Ethernet cable is not connected.");

}

// try to congifure using IP address instead of DHCP:

Ethernet.begin(mac, ip, myDns);

} else {

PRINT(" DHCP assigned IP ");

PRINTLN(Ethernet.localIP());

}

Wire.setSDA(PIN0\_SDA);

Wire.setSCL(PIN0\_SCL);

if (!display.begin(SSD1306\_SWITCHCAPVCC, SCREEN\_ADDRESS)) {

for (;;) ;

}

display.display();

delay(1000);

display.clearDisplay();

// give the Ethernet shield a second to initialize:

// MQTT [[

mqtt\_client.setServer(mqtt\_server, 1883);

mqtt\_client.setCallback(callback);

// ]]

// give the Ethernet shield a second to initialize:

delay(500);

}

unsigned long prevmillis = 0;

uint8\_t qrcode[qrcodegen\_BUFFER\_LEN\_MAX];

uint8\_t tempBuffer[qrcodegen\_BUFFER\_LEN\_MAX];

int prevstat = -1;

void loop() {

digitalWrite(trigPin, LOW);

delayMicroseconds(2);

digitalWrite(trigPin, HIGH);

delayMicroseconds(10);

digitalWrite(trigPin, LOW);

// measure duration of pulse from ECHO pin

duration\_us = pulseIn(echoPin, HIGH);

// calculate the distance

distance\_cm = duration\_us / 29 / 2;

// Check if someone is within 100 cm range

if (distance\_cm < 100 && distance\_cm > 0) { // Check if distance is within the valid range (0 to 100 cm)

// Display the image on the OLED screen

if (prevstat != 0) {

display.clearDisplay();

//display.setCursor(34, 0);

bool ok = genQR("http://192.168.0.102:5000/", tempBuffer, qrcode);

if (ok) {

drawQR(34, 0, qrcode);

}

prevstat = 0;

debounce = 0;

}

} else {

if (prevstat != 1) {

display.clearDisplay();

display.setTextSize(1);

display.setTextColor(SSD1306\_WHITE);

display.setCursor(20, 5);

display.print(F("Try Skintellect!!"));

display.display();

prevstat = 1;

}

}

//\*// MQTT [[

if (!mqtt\_client.connected()) {

reconnect();

}

mqtt\_client.loop();

// ]]

//\*/

//if (millis() - prevmillis > 5000)

if (capture\_requested) {

capture\_requested = 0;

// Flush serial buffer.

while (Serial.available())

Serial.read();

// Send SYNC code

Serial.write(SYNC);

uint32\_t length = serial\_read\_length();

if (length > 0) {

String response = "Length: ";

response += length;

mqtt\_client.publish("W5300-MQTT", response.c\_str());

uint32\_t received = serial\_read\_data(img\_buf, length);

if (received != length) {

// Time-out error!!!!

String response = "Time-out: "; // + received;

response += received;

mqtt\_client.publish("W5300-MQTT", response.c\_str());

} else {

// Send it to the server

String response = "Image received: "; // + length;

mqtt\_client.publish("W5300-MQTT", response.c\_str());

http\_postData(img\_buf, length);

}

} else {

mqtt\_client.publish("W5300-MQTT", "No data");

}

}

}

bool genQR(const char\* text, uint8\_t\* buf, uint8\_t\* qrcode) {

bool ret = qrcodegen\_encodeText(text, buf, qrcode, qrcodegen\_Ecc\_LOW,

//qrcodegen\_VERSION\_MIN, qrcodegen\_VERSION\_MAX, qrcodegen\_Mask\_AUTO, true);

3, 3, qrcodegen\_Mask\_AUTO, true);

return ret;

}

void drawQR(int xinit, int yinit, uint8\_t\* qrcode) {

int size = qrcodegen\_getSize(qrcode);

int scale = SCREEN\_HEIGHT / size;

display.clearDisplay();

for (int y = 0; y < size; y++) {

for (int x = 0; x < size; x++) {

if (qrcodegen\_getModule(qrcode, x, y)) {

display.fillRect(xinit+x\*scale, yinit+y\*scale, scale, scale, SSD1306\_WHITE);

}

}

}

display.display();

}

uint32\_t serial\_read\_length() {

uint32\_t length = 0;

byte recv[4];

int index = 0;

prevmillis = millis();

//Loop with 1sec timeout.

while (millis()-prevmillis < 1000) {

if (Serial.available()) {

// Read the most recent byte

recv[index++] = Serial.read();

if (index >= 4) {

// Big endian

length = (recv[0] << 24) | (recv[1] << 16) | (recv[2] << 8) | recv[3];

break;

}

}

}

return length;

}

uint32\_t serial\_read\_data(byte \*buf, uint32\_t length) {

int index = 0;

uint32\_t maxsize = 1024;

uint32\_t recvlen = 0;

uint32\_t remain = length;

prevmillis = millis();

while (remain > 0 && (millis()-prevmillis < 2000)) {

if (remain > maxsize) {

recvlen = serial\_read\_data0(buf, maxsize);

} else {

recvlen = serial\_read\_data0(buf, remain);

}

if (recvlen == -1) {

return length - remain;

}

buf += recvlen;

remain -= recvlen;

/\*

String response = "RecvLen: ";

response += recvlen;

mqtt\_client.publish("W5300-MQTT", response.c\_str());

\*/

}

return length - remain;

}

uint32\_t serial\_read\_data0(byte \*buf, uint32\_t length) {

int index = 0;

prevmillis = millis();

//Loop with 2sec timeout.

while (millis()-prevmillis < 1000) {

if (Serial.available()) {

// Read the most recent byte

buf[index++] = Serial.read();

if (index >= length)

return index;

}

}

return -1;

}

WHAT THIS FILE CONTAINS (IN SHORT):

This code combines various functionalities such as ultrasonic distance measurement, QR code generation, Ethernet communication, MQTT (Message Queuing Telemetry Transport) communication, and serial communication:

1. **Library Inclusions**:

- Several libraries are included at the beginning of the code, such as Ethernet, Wire (for I2C communication), PubSubClient (for MQTT communication), ArduinoJson (for JSON manipulation), Adafruit\_GFX (for graphics functions), Adafruit\_SSD1306 (for OLED display control), and qrcodegen (for QR code generation).

2. **Global Variables and Constants**:

- `device\_unique\_id`: A unique identifier for the device, likely combining a string, compilation date, and time.

- Various pin definitions for ultrasonic sensor, I2C communication, and reset pin for the OLED display.

- Ethernet configuration parameters, including MAC address, IP address, server IP address, and port.

- `max\_transfer` and `max\_buffer`: Constants used for data transfer buffer sizes.

- Arrays and variables used for image data storage and communication.

- MQTT server IP address, MQTT client, and a flag `capture\_requested` for capturing data.

3. **HTTP POST Function** (`httpPostForm`):

- This function is responsible for sending an HTTP POST request to a server, including both text and image data. It prepares the request body, headers, and sends the data using the `client\_write\_large` function.

4. **MQTT Functions** (`callback`, `reconnect`):

- `callback`: This function is called when a new MQTT message arrives. It prints the received message to the serial output.

- `reconnect`: This function is used to reconnect to the MQTT broker if the client is not connected.

5. **Setup Function**:

- Initializes serial communication, pins, Ethernet connection, and OLED display.

- Sets up the MQTT client with server and callback.

6.**Loop Function**:

- Measures distance using an ultrasonic sensor and displays information on an OLED screen based on the measured distance.

- Handles MQTT communication, including connecting to the broker, subscribing, and publishing messages.

- If `capture\_requested` is true, the code listens for image data over serial communication, then sends it to a server using HTTP POST.

7. **QR Code Generation Functions** (`genQR`, `drawQR`):

- `genQR`: Generates a QR code for a given text using the `qrcodegen` library.

- `drawQR`: Draws the generated QR code on the OLED screen.

8. **Serial Data Handling Functions** (`serial\_read\_length`, `serial\_read\_data`, `serial\_read\_data0`):

- `serial\_read\_length`: Reads a 4-byte length value from serial communication.

- `serial\_read\_data`: Reads a specified amount of data from serial communication and stores it in a buffer.

- `serial\_read\_data0`: Reads a specified amount of data from serial communication without timeout handling.

***This code is designed for a device that performs the following tasks:***

- Measures distance using an ultrasonic sensor and displays QRcode on an OLED screen.

- Communicates with a server using HTTP POST to send text and image data.

- Utilizes MQTT communication for subscribing and publishing messages.

- Generates QR codes for specified URLs and displays them on the OLED screen.