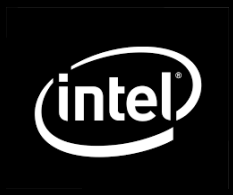


Intel Unnati Industrial Training Program

Problem Statement Report

Team - Cyber Savants

Problem Statement 3:

Customized AI Kitchen for India.

Prerequisites:

1. Embedded HW system functionality which can be used for AI application.
2. Python script to write logic and make use of existing libraries.

We are in the world of AI (Artificial Intelligence) now and idea here is to develop AI kitchen model for customized dishes. It's one the challenging that we have every day to prepare dishes in time at home. There could be AI based vending machines, coffee and bread making products in the market now, but this would not cover all the dishes of user's choice. Another bigger issue to get the vessels for cooking which must be ready / washed. We have vessels washing machines but, it requires manual intervention to put them for washing or take it after washing. Idea here is to address that as well. Pick a right vessel based on the quantity.

Unique Idea (Solution):

A completely automated system for cooking dishes as per a User's Choice. We use G**enerative AI** to automate the steps required to prepare a dish as per an existing recipe dataset.

A **user-friendly** interface has been developed for seamless operation, allowing users to select a dish of their choice. Our code is at its early stage and yet covers everything from recipe generation to automation of the machine to create the dish.

Selecting the right vessel has been the most intriguing part. We came up with a solution where the ingredients are identified using an **object detection** program and is then assigned to a vessel according to the instructions in the recipe.

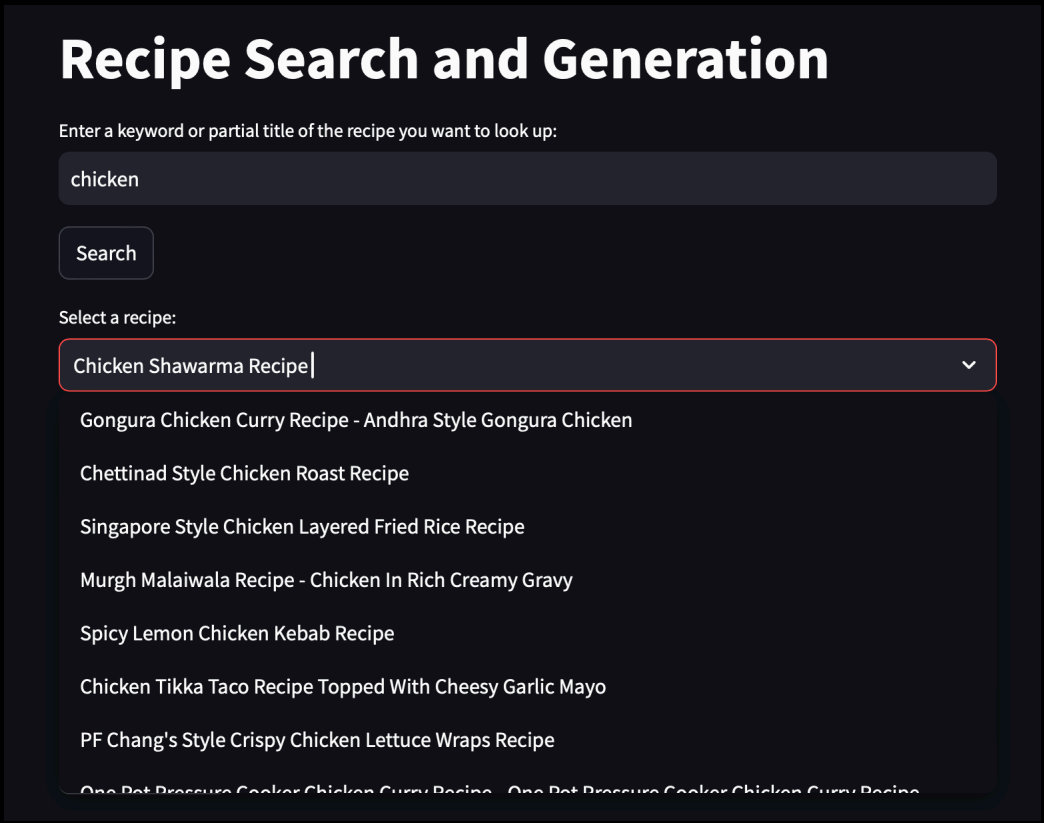
Features Offered:

- Users can search for recipes with keywords or partial titles.
- The Recipe details provide step-by-step instructions which are then passed to the machine.
- Object Detection model that detects ingredients.
- A Dynamic user-interface allows users to select recipes from search results and view details in a user-friendly manner.
- Uses Raspberry Pi 4 with Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.8GHz, that controls the entire automation.
- Uses Google's latest large language model Gemini to generate precise instructions for the cooking machine.
- A simulation is shown after generating the instructions , that offers detailed steps along with the workflow

Process Flow:

Recipe Selection

Users interact with the web interface where they can enter a partial title or keyword of the recipe they want to search for. Upon entering the query the application searches for the recipes in the'**processed_recipes.json**' file. A predictive autocomplete dropdown menu shows the user a list of matching recipes, where they can click to view detailed information.



When a user selects a recipe from the list detailed information including title, list of ingredients, preparation time, cooking time, no. of servings and cooking instructions. The recipe is then given to the Large Language Model, Gemini which converts the recipe into a JSON file that contains detailed machine operable step-by-step instructions for the cooking machine.

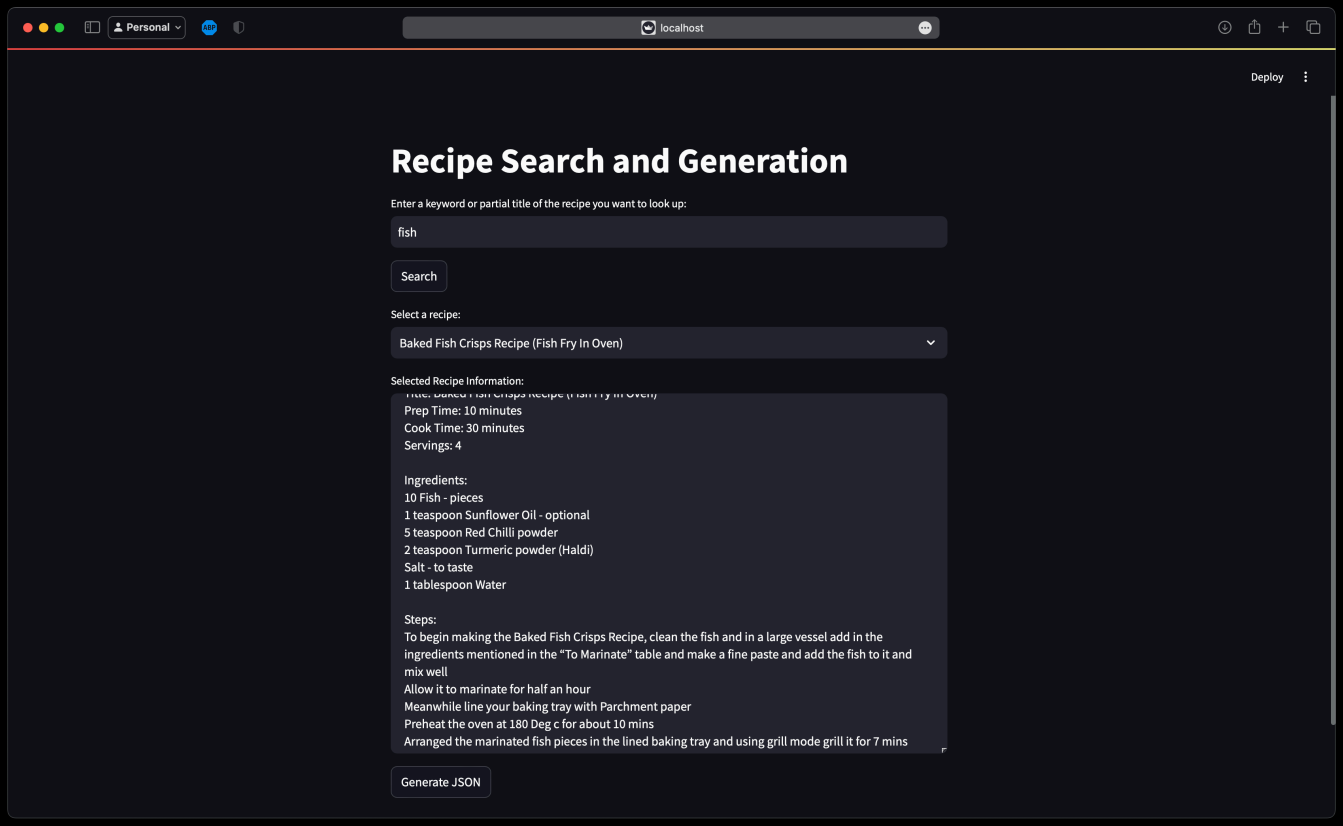
`Python 3.12.2`

```python
import json
import google.generativeai as genai
import streamlit as st


# Function to search for recipes based on user-input
def search_recipes(query):
    try:
        with open('processed_recipes.json', 'r') as file:#loading json file
            recipes = json.load(file)

        matching_recipes_list = [
            recipe for recipe in recipes if query.lower() in recipe['title'].lower()
        ]#list storing recipe information

        if not matching_recipes_list:
            return "No matching recipes found."

        return matching_recipes_list

    except FileNotFoundError:
        return "The file 'processed_recipes.json' was not found."
    except json.JSONDecodeError:
        return "Error decoding JSON."


# Function to format recipe details , formatted printing of recipies
def format_recipe_info(recipe):
    ingredients = "\n".join(recipe['ingredients'])
    steps_for_iteration = "\n".join(recipe['steps'])
    return (
        f"Title: {recipe['title']}\n"
        f"Prep Time: {recipe['prep']} minutes\n"
        f"Cook Time: {recipe['cook']} minutes\n"
        f"Servings: {recipe['servings']}\n\n"
        f"Ingredients:\n{ingredients}\n\n"
        f"Steps:\n{steps_for_iteration}"
    )


# Integrating Google's Gemini to convert recipe into a machine operable Json file
genai.configure(api_key='AIzaSyCURo4TsfWgX9Wbo7ujqhvzr8yi9RVtdYY')
model_generate = genai.GenerativeModel('gemini-1.5-pro-latest', generation_config={"response_mime_type": "application/json"})


# Graphical user interface using python streamlit package
def main():
    st.title("Recipe Search and Generation")#<title></title>

    query = st.text_input("Enter a keyword or partial title of the recipe you want to look up:")#<h></h>

    if st.button("Search") or query in st.session_state:
        st.session_state.query = query
        matching_recipes = search_recipes(query)
        if isinstance(matching_recipes, str):  # Error message
            st.error(matching_recipes)
            st.session_state.matching_recipes = None
        else:
            st.session_state.matching_recipes = matching_recipes
```

```
    if 'matching_recipes' in st.session_state and st.session_state.matching_recipes:
        selected_recipe = st.selectbox("Select a recipe:", st.session_state.matching_recipes,
                                       format_func=lambda recipe: recipe['title'])
        selected_recipe_info = format_recipe_info(selected_recipe)
        st.text_area("Selected Recipe Information:", value=selected_recipe_info, height=200)

        if st.button("Generate JSON"):
            response = model_generate.generate_content(
                f"Using the recipe details: {selected_recipe_info}, create a JSON file with precise step-by-step actions for a "
                "cooking machine. The JSON should follow this exact format:\n\n"
                "{\n"
                "  \"steps\": [\n"
                "    {\n"
                "      \"step\": <int>,\n"
                "      \"action\": \"<string>\",\n"
                "      \"ingredients\": [\n"
                "        {\n"
                "          \"name\": \"<string>\",\n"
                "          \"quantity\": <float>,\n"
                "          \"unit\": \"<string>\"\n"
                "        }\n"
                "      ],\n"
                "      \"parameters\": \"<string>\",\n"
                "      \"time\": <int>\n"
                "    }\n"
                "  ]\n"
                "}"
            )#prompt to convert recipe details into Json file using the following input
            json_string = response.text.replace('\\', '').replace('\n', '')
            try:
                recipe_data = json.loads(json_string)
                st.json(recipe_data)
                with open("actions.json", "w") as output_file:
                    json.dump(recipe_data, output_file, indent=2)
                st.success("JSON file generated and saved as 'actions.json'.")
            except json.JSONDecodeError as error_mode:
                st.error(f"Error decoding JSON: {error_mode}")


if __name__ == "__main__":
    main()
```

steamlit.py

```
% streamlit run recipe_app.py
  For better performance, install the Watchdog module:
$ xcode-select —install
$ pip install watchdog
```



We integrate Google’s Gemini LLM and streamlit python package to create an interactive web application for recipe generation.

recipe_search() — This function reads recipes from a local processed_recipes.json file. Formal parameter query is passed to check for the recipes in the json file , and returns a suitable error message if the recipe is absent inside the file.

format_recipe_info() — It formats the selected recipe details including the ingredients and steps and displays it

genai.configure() — To convert formatted recipe details into step-by-step instructions which is saved into a JSON file.

Users input a keyword to search for recipes. Matching recipes are displayed in a select box. The selected recipe's formatted details are shown in a text area. Users can click a button to generate a JSON file of the recipe using the generative model. JSON Generation: The app sends a prompt to the Gemini model, requesting a JSON format of the recipe. It handles the response, formats

it into JSON, and saves it to actions.json. If the conversion fails, an error message is displayed.

Object Detection

Based on the instructions received after generating the json file, the machine detects ingredients on the conveyor and passes it to the required vessel. The code sets up a web based image classification system.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ingredient Image Detection Model</title>
    <link rel="stylesheet" href="styles_object.css">
    <script src="javascript_object.js"></script>
    <script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest/dist/tf.min.js"></script>
    <script src="https://cdn.jsdelivr.net/npm/@teachablemachine/image@latest/dist/teachablemachine-image.min.js"></script>
</head>
<body>
    <div class="container">
        <h1>Ingredient Image Detection Model</h1>
        <button type="button" onclick="init()">Start</button>
        <div id="webcam-container-ui"></div>
        <div id="label-container-ui"></div>
    </div>

</body>
</html>
```

The code sets up web-based image classification system using Google's Machine Learning Tool, Teachable Machine. The system uses a webcam to capture real time images and classify ingredients based on the trained model. A number of prediction classes are invoked from the loaded model. The webcam is started and a loop for continuous image capture is set up. The webcam output is shown in the website GUI. Model.predict() uses the webcam feed to make a prediction. A loop iterates over each prediction class and a confidence level is shown of the

```
const URL = "https://teachablemachine.withgoogle.com/models/8hJsh2CJ5/";
let model, webcam, labelContainer, maxPredictions;

async function init() {
    const modelURL = URL + "model.json";
    const metadataURL = URL + "metadata.json";
    model = await tmImage.load(modelURL, metadataURL);
    maxPredictions = model.getTotalClasses();

    const flip = true;
    webcam = new tmImage.Webcam(200, 200, flip);
    await webcam.setup();
    await webcam.play();
    window.requestAnimationFrame(loop);

    document.getElementById("webcam-container-ui").appendChild(webcam.canvas);
    labelContainer = document.getElementById("label-container-ui");
    for (let i = 0; i < maxPredictions; i++) {
        labelContainer.appendChild(document.createElement("div"));
    }
}

async function loop() {
    webcam.update();
    await predict();
    window.requestAnimationFrame(loop);
}

async function predict() {
    const prediction = await model.predict(webcam.canvas);
    labelContainer.innerHTML = '';
    for (let i = 0; i < maxPredictions; i++) {
        const probability = prediction[i].probability.toFixed(2);
        if (probability > 0.7) {
            const classPrediction = prediction[i].className + ": " + probability;
            const predictionElement = document.createElement("div");
            predictionElement.innerHTML = classPrediction;
            labelContainer.appendChild(predictionElement);
        }
    }
}
```

object. If the confidence level is above 0.7, it is displayed in the GUI and then passed onto the required vessel.

Simulation

```
import streamlit as st
import json
import time

# Loading recipe instructions from actions.json
def load_actions():
    with open('actions.json', 'r') as f:
        return json.load(f)

# Initializing session state variables
if 'current_step_index' not in st.session_state:
    st.session_state.current_step_index = 0

if 'steps' not in st.session_state:
    st.session_state.steps = load_actions()['steps']


def display_current_step():
    step_len = st.session_state.steps[st.session_state.current_step_index]
    st.markdown(f"### Step {step_len['step']}")
    st.markdown(f"**Action:** {step_len['action']}")

    st.markdown("**Ingredients:**")
    for ing in step_len['ingredients']:
        names = ing.get('name', 'Unknown ingredient')
        quantity = ing.get('quantity')
        unit = ing.get('unit')

        # Creating the ingredient based on the given instruction
        ingredient_str = names
        if quantity is not None:
            ingredient_str = f"{quantity} {unit} of {names}" if unit is not None else f"{quantity} of {names}"

        st.markdown(f"- {ingredient_str}")

    if 'parameters' in step_len:
        st.markdown(f"**Parameters:** {step_len['parameters']}")

    if 'time' in step_len and step_len['time'] is not None:
        st.markdown(f"**Time:** {step_len['time']} minutes")
    else:
        st.markdown("**Time:** Not specified")

    if st.button('Execute Step'):
        execute_step()
```

```
def execute_step():
    step_len_2 = st.session_state.steps[st.session_state.current_step_index]
    st.write(f"Executing: {step_len_2['action']}")
    if 'time' in step_len_2 and step_len_2['time'] is not None:
        time.sleep(step_len_2['time'] * 60)  # Simulate the cooking time (convert minutes to seconds)
    st.write(f"Completed: {step_len_2['action']}")
    next_step()


def next_step():
    st.session_state.current_step_index += 1
    if st.session_state.current_step_index < len(st.session_state.steps):
        st.experimental_rerun()
    else:
        st.write("Recipe completed!")

# interface
st.title("Cooking Machine Simulation")

if st.session_state.current_step_index == 0:
    if st.button('Start Simulation'):
        st.session_state.current_step_index = 1
        st.experimental_rerun()
else:
    display_current_step()
```

This code creates a simulation of the machine workflow which runs alongside the machine. It reads steps from actions.json and executes each step of the recipe. load_actions()- this function reads the recipe from the 'actions.json' file and returns its contents as a python dictionary. display_current_step()- this function prints each step, lists the ingredients, including quantity and units and adds a button to execute the step. execute_step()- this function displays a message indicating the execution of the action and pauses the step till the end of the duration if given and moves to the next step. next_step()- this function checks if there are more steps and reruns the streamlit app to display the next step.

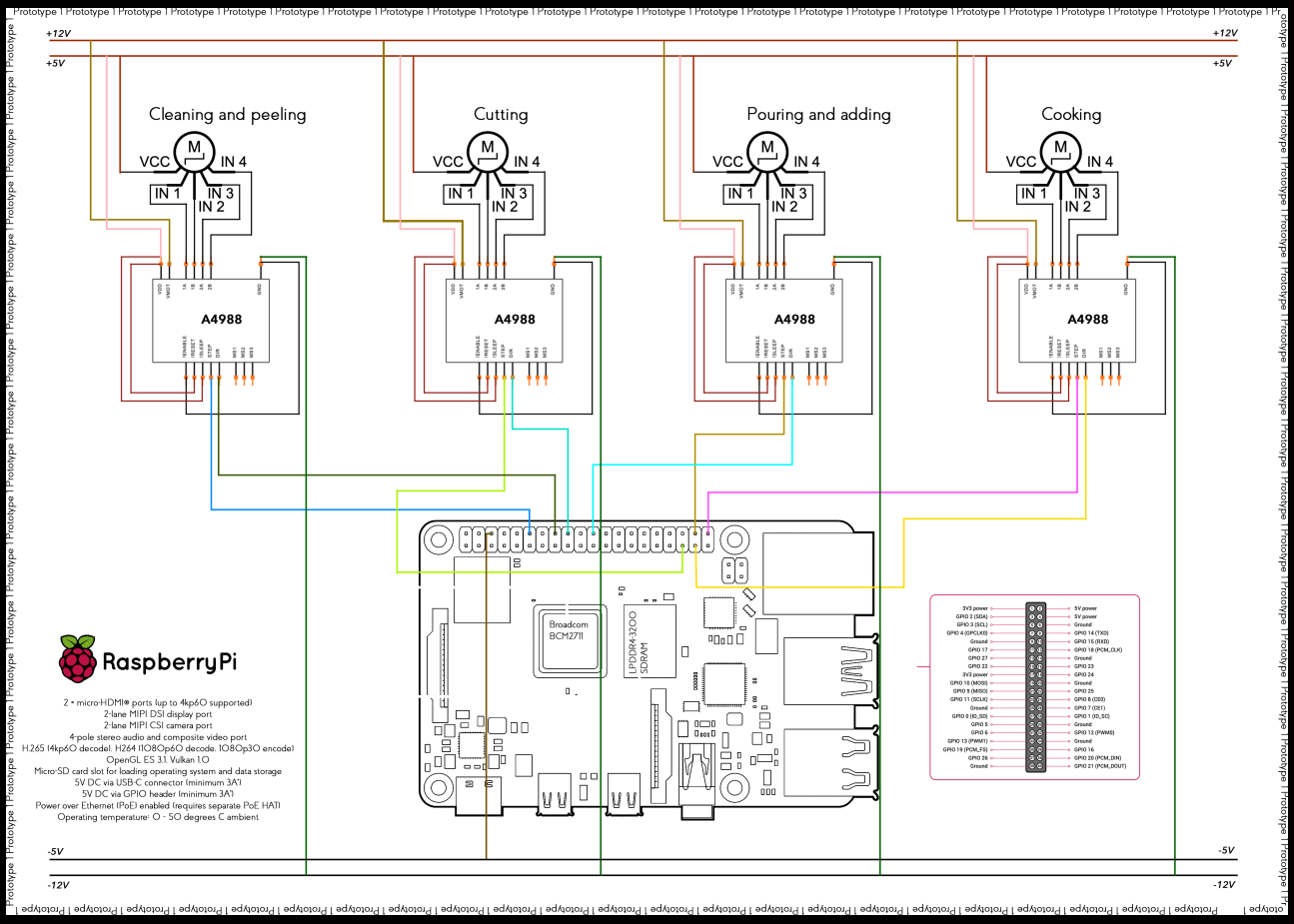
Hardware Integration

This code is designed to control a cooking machine using a Raspberry Pi 4 and stepper motors, executing a series of cooking steps defined in a JSON file. Each step in the JSON file corresponds to a specific action, such as cleaning, peeling, cutting, adding and cooking. RPi.GPIO is used for controlling the GPIO pins on

```python
import json
import time
import RPi.GPIO as GPIO

# GPIO pin definitions for stepper motors
STEP_PIN1 = 18  # Stepper motor cleaning and peeling
DIR_PIN1 = 23  # Stepper motor cleaning and peeling
STEP_PIN2 = 19  # Stepper motor 2 cutting
DIR_PIN2 = 24  # Stepper motor 2 cutting
STEP_PIN3 = 20  # Stepper motor 3 pouring and adding
DIR_PIN3 = 25  # Stepper motor 3 pouring and adding
STEP_PIN4 = 21  # Stepper motor 4 cooking
DIR_PIN4 = 26  # Stepper motor 4 cooking

# Setup for GPIO
GPIO.setmode(GPIO.BCM)
GPIO.setup(STEP_PIN1, GPIO.OUT)
GPIO.setup(DIR_PIN1, GPIO.OUT)
GPIO.setup(STEP_PIN2, GPIO.OUT)
GPIO.setup(DIR_PIN2, GPIO.OUT)
GPIO.setup(STEP_PIN3, GPIO.OUT)
GPIO.setup(DIR_PIN3, GPIO.OUT)
GPIO.setup(STEP_PIN4, GPIO.OUT)
GPIO.setup(DIR_PIN4, GPIO.OUT)


def to_perform_stepper_action(step_pin, dir_pin, steps, direction_length):
    GPIO.output(dir_pin, direction_length)
    for _ in range(steps):
        GPIO.output(step_pin, GPIO.HIGH)
        time.sleep(0.01)  # to adjust the delay
        GPIO.output(step_pin, GPIO.LOW)
        time.sleep(0.01)


def cleaning_and_peeling():
```

```
    to_perform_stepper_action(STEP_PIN1, DIR_PIN1, 200, GPIO.HIGH)


def cutting():
    to_perform_stepper_action(STEP_PIN2, DIR_PIN2, 200, GPIO.HIGH)


def pouring_and_adding():
    to_perform_stepper_action(STEP_PIN3, DIR_PIN3, 200, GPIO.HIGH)


def cooking():
    to_perform_stepper_action(STEP_PIN4, DIR_PIN4, 200, GPIO.HIGH)


def further_execute_step(step):
    action = step['action'].lower()
    parameters = step['parameters'].lower()
    time_required = step['time']

    if action == "cleaning and peeling":
        cleaning_and_peeling()
    elif action == "add":
        to_perform_stepper_action(STEP_PIN4, DIR_PIN4, 100, GPIO.LOW)

    elif action == "cutting":
        cutting()
    elif action == "pouring and adding":
        pouring_and_adding()
    elif action == "cooking":
        cooking()
    elif action == "cook":
        print(f"Cooking for {time_required} minutes, {parameters}")

    if time_required > 0:
        time.sleep(time_required * 60)


def main():
```

```
    with open('actions.json', 'r') as file:
        data = json.load(file)

    for step in data['steps']:
        further_execute_step(step)

    GPIO.cleanup()


if __name__ == '__main__':
    main()
```

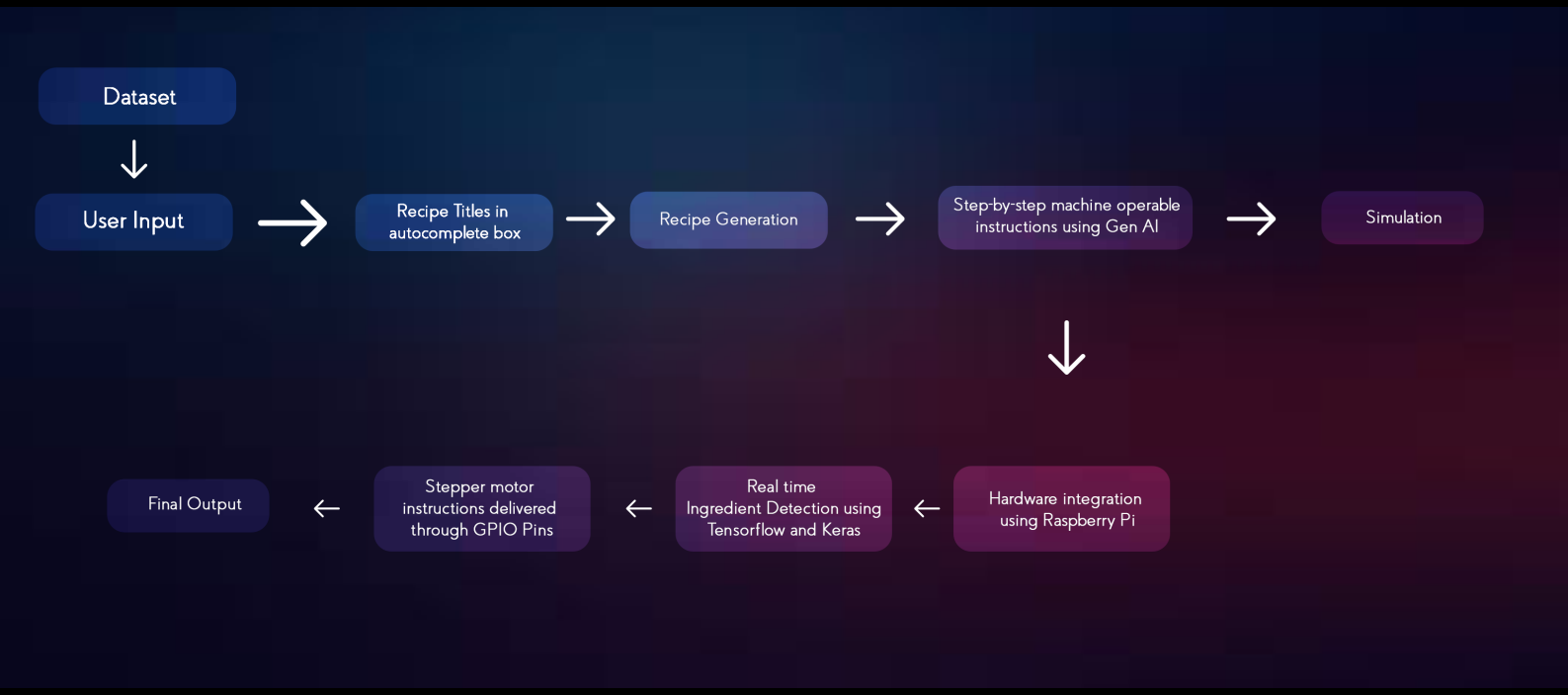


the Raspberry Pi. We define the GPIO pins associated with different cooking actions and then set up the pins for output mode using Broadcom SOC channel numbering system. to_perform_stepper_action()- This function controls the stepper motor by toggling the step and direction pins depending on the instruction in the 'actions.json' file. cleaning, peeling, cutting, adding and cooking are then done by calling this function with appropriate pins and parameters. further_execute_step()- This function takes an instruction form the

JSON file, determines the action and paramters, and calls the corresponding function. It pauses if the step involves a specific time duration. main()- The main function reads from the JSON file, iterates through each step and calls further_execute_step() for each one.

Architecture Diagram:

The process starts with the dataset, selects a recipe, generates actions, simulates them, integrates with hardware, and produces the final product.



Technologies used:

Streamlit is an open-source Python library which is used to create and share custom web apps for machine learning and data science. In this project, Streamlit is used to create a user interface for Recipe Selection and Cooking Simulation. It provides users with interactions and displays step dynamically.

Google's Generative AI model has been used to convert the recipe instructions into a JSON file which contains step-by-step instructions which is then passed to the simulation and hardware mechanisms.

Teachable Machine: Provides an easy way to create and integrate machine learning models for recognizing images, sounds, or posesTeachable Machine is integrated into the project to implement real-time ingredient detection.

JSON (JavaScript Object Notation) JSON is a lightweight data interchange format that is easy for humans to read and machine to parse and generate. In this project the recipe dataset has been converted to a JSON file for easy manipulation. The recipe instructions are then converted into a JSON file which contains machine operable instructions.

Team Members and Contributions:

Jithin Rajesh: Contributed for the interface development , backend development , loading and processing of JSON file operations and Generative AI development.

Abin S Manoj: Contributed for the interface development , object detection model training and hardware integration.

Elvis Mathews Olickal: Hardware assembly and model logic.

Alan Benny: Oversaw the entire project , interface development and data collection.

Athul Bijo: Oversaw the entire project , contributed to the final report for the project and ensured coordination between team members

Conclusion:

In conclusion, our system makes cooking easy and fun by using smart technology to help prepare dishes that users choose. The user-friendly interface allows anyone to select a dish from a recipe list. Although the project is in its early stages, it successfully automates everything from following the recipe to cooking the dish. A big challenge was choosing the right pot for the ingredients, but we solved it by using a program that detects ingredients and picks the right pot based on the recipe. This project shows how cooking can become super easy and fully automated in the future.

Future Directions:

- Enhanced Object detection: Training of the model with extensive and diverse datasets could imporve the accuracy and robustness of the model , in recognising obscure ingredients and actions.
- Recipe Database Expansion: Integrating a larger database of recipes and allowing users to add their own recipes could make the application more versatile and appealing to a broader audience.
- Advanced Hardware Integration: Incorporating more sophisticated hardware components, such as robotic arms or precision-controlled heating elements, could further automate and enhance the cooking process.
- User Feedback Loop: Implementing a feedback mechanism where users can rate the accuracy and usefulness of the simulation could help continuously refine the system.