Building your own AI Nutrition Coach using a Multi-Agent System and Multimodal AI

Estimated time needed: 60 minutes

Welcome, learners! Have you ever wondered how AI could become your personal health companion, guiding you to make smarter food choices effortlessly? Imagine an app that not only identifies what's on your plate, but also gives you insightful nutritional details and actionable tips—all powered by state-of-the-art AI. In the previous project, we built the foundation for such a tool: NourishBot, an AI-driven nutrition assistant that leverages Meta's advanced multimodal model, Llama 3.2 90B Vision Instruct, alongside Flask.

But what if we could do more? What if NourishBot could evolve beyond just recognizing food to offering dynamic, real-time advice based on various factors—like dietary preferences—as well as suggesting recipes based on what you have in your fridge? That's exactly what we're doing in this project. We're stepping into the world of Multi-Agent Systems (MAS)—a framework where multiple AI agents collaborate to make complex decisions and offer tailored guidance.

Think of it this way: one agent processes the visual input (your meal), another analyzes the nutritional content, and a third could provide meal suggestions based on your dietary preference. By working together, these agents can deliver a personalized, seamless user experience that feels less like a static app and more like an intelligent partner in your wellness journey.

To make it even better, we'll use Gradio, a simple yet powerful framework for creating interactive web apps. This means you'll walk away not just understanding advanced AI concepts, but also with a fully functional, sleek app that anyone can use.

In this 60-minute hands-on journey, you'll learn how to:

- Build and deploy a `multi-agent system` that powers intelligent decision-making.
- Integrate `multimodal AI` to handle both visual and textual data.
- Create a user-friendly interface using `Gradio`, so your app is accessible to anyone.

By the end, you'll have more than just a project—you'll have a powerful tool that could inspire future innovations in health tech, personalized AI assistants, or even smart kitchen devices.

If you're curious about how cutting-edge AI frameworks can be combined to solve real-world problems and want to gain hands-on experience with Generative AI, MAS, and multimodality, this is the perfect project for you!

Disclaimer:

The recipes and nutritional suggestions provided by this AI NourishBot are generated based on image analysis and automated processes. While we strive for accuracy and safety, these recommendations may not account for all potential dietary restrictions, allergens, or toxic ingredients.

Please note:

- Always review the suggested recipes and ingredients for safety before preparation or consumption.
- If you have specific health concerns, dietary restrictions, or allergies, consult a qualified nutritionist or healthcare provider for personalized advice.
- The AI's recommendations should be used as guidance and not as a definitive medical or dietary prescription.

By using this service, you acknowledge that while we aim to filter out unsafe suggestions, the final responsibility for ensuring recipe safety rests with you.

A quick look at AI NourishBot

What Does the AI NourishBot App Do?

Let's see a very brief demo of how the app works.

The `AI NourishBot` app is a smart dietary assistant that uses advanced AI models to analyze food images and generate helpful nutritional insights or personalized recipe ideas based on user preferences. Here's a quick recap of what the app does:

- **Food Image Analysis**:
  - Upload an image of any food item, and the app will analyze the contents to provide a detailed nutrient breakdown, including:
    - Calories
    - Macronutrients (protein, carbs, fats)
    - Micronutrients (vitamins, minerals)
- **Personalized Recipe Suggestions**:
  - For users with specific dietary restrictions (e.g., vegan, gluten-free), the app can filter ingredients and generate creative recipe ideas using only the allowed ingredients.
  - It ensures that the suggested recipes are healthy, easy to prepare, and aligned with the user's dietary needs.
- **Dietary Filtering**:
  - The app takes into account dietary restrictions such as:
    - Vegan
    - Vegetarian
    - Gluten-free
    - Keto
  - It filters out ingredients that don't comply with the specified diet.
- **Health Evaluation**:
  - Once the nutrient analysis is complete, the app provides a summary of the meal's healthiness, along with suggestions for improving its nutritional balance.
- **User-Friendly Interface**:
  - The app offers an intuitive, easy-to-use web interface built using **Gradio**, where users can:
    - Upload an image of their meal
    - Specify dietary preferences (optional)
    - Choose a workflow: either "**Recipe**" or "**Analysis**"
- **Two Main Workflows**:
  - **Recipe Workflow**: Generates recipe ideas using the detected ingredients and dietary restrictions.
  - **Analysis Workflow**: Provides detailed nutritional information and health insights for the uploaded food image.

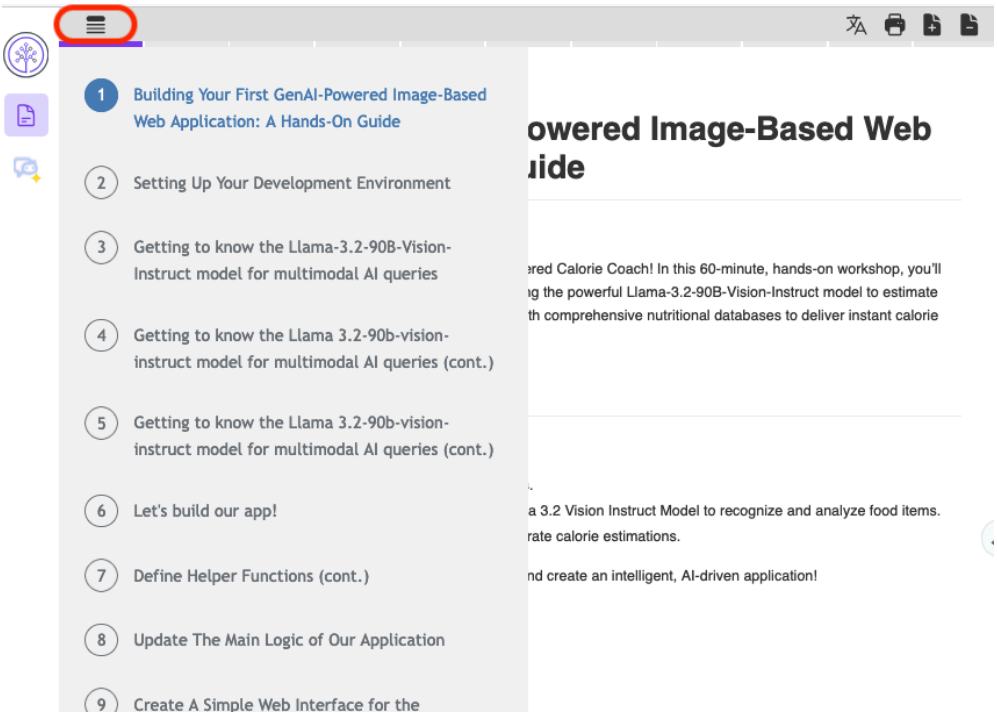
This app is designed for anyone looking to maintain a healthy diet, experiment with new recipes, or get quick insights into their meals—all powered by AI!

Tips for the best experience - Read-only

Keep the following tips handy and refer to them at any point of confusion throughout the tutorial. Do not worry if they seem irrelevant now. We will go through everything step by step later.

Note: This page is read-only, aiming to teach you how to navigate the CloudIDE environment effectively. Do NOT run any code on this page.

- You can choose to follow the `quick version` of building and running the application on the next page if you are in a hurry or you are too excited to test the final result!
- At any point throughout the project, if you are lost, click on **Table of Contents** icon on the top left of the page and navigate to your desired content.
- Whenever you make changes to a file, be sure to save your work.



- At the end of each section, you will be given the fully updated script for that part. And at the end of the project, you will be prompted to pull the complete codebase of the project as well.
- Cloud IDE automatically saves any changes you make to your files immediately. You can also save from the toolbar.
- For running the application, always ensure `app.py` is running in the background before opening the Web Application.
- You run a code block by clicking on the `>_` on bottom right.

```
python app.py
```

- Always ensure that your current directory is `/home/project/NourishBot`. If you are not in the NourishBot folder, certain code files may fail to run. Use the `cd` command to navigate to the correct location.
- Make sure you are accessing the application through port `5000`. Clicking on the purple Web Application button will run the app through port 5000 automatically.

Web Application

- If you get an error about not being able to access another port (e.g. 8888) just `refresh` the app by clicking on the small refresh icon. In case of other errors related to the server, simply refresh the page as well.

- To stop execution of `app.py` in addition to closing the application tab, hit `Ctrl+C` in the terminal.
- If you encounter an error running the application or after you entered your desired keyword try refreshing the app using the button on the top of the application's page. You can try inputting a different query too.
- Typically, using the models provided by Watsonx.ai would require Watsonx credentials, including an API key and a project ID. However, in this lab, these credentials are not needed.

- One of the agents uses the Granite 3.1 model, which, at the time of writing this lab, is unstable and may experience delays in generating responses. If the app takes more than 2-3 minutes to produce an output, press `Ctrl + C` in the terminal to terminate the process and restart the app.

Quick version - run the final app

In **less than 10 minutes** you can have a functioning app only by following this page's instructions!

Disclaimer: Ignore this page and skip to the next page for the step by step tutorial. Follow this page ONLY if you want to get the final application up and running without getting into the learning material.

First, we'll set up the NourishBot application in your development environment. By the end of this step, you'll have the final application running and ready to explore.

Step 1: Clone the project's GitHub repository

To begin, we'll clone the NourishBot application repository from GitHub. This repository contains all the source code with which we'll be working throughout this project.

Run the following command in the terminal to clone the repository by clicking on the run button >_:

```
git clone --no-checkout https://github.com/HaileyTQuach/Smart-Nutritional-App.git NourishBot
cd NourishBot
git checkout 5-final
```

Then, you need to set up a virtual environment to install the dependencies.

Step 2: Set up a Python virtual environment

Initialize a new Python virtual environment to keep required library versions tidy. Note, you can run the snippet directly by clicking on the run button >_.

```
python3.11 -m venv venv
source venv/bin/activate
```

Step 3: Install the required libraries

With your virtual environment activated, install all the required libraries via >_. It will take approximately **3-5 minutes** to install all the required libraries. Feel free to go grab a coffee while you wait!

This command installs `crewai`, which orchestrates AI agents for task management, and `gradio`, which builds the user-friendly web interface. Additionally, `ibm_watsonx_ai` is crucial for ingredient detection and nutritional analysis, while `langchain` and `fastapi` handle natural language generation and asynchronous operations, respectively. Other key packages include `pydantic` for data validation, `pillow` for image processing, and `requests` for API communication. Together, these packages enable the app's core functionalities, from AI-driven workflows to user interaction.

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

Step 4: Run the Gradio app in the terminal

```
python app.py
```

Note: After it runs successfully, you will see a message similar to the following example in the terminal:

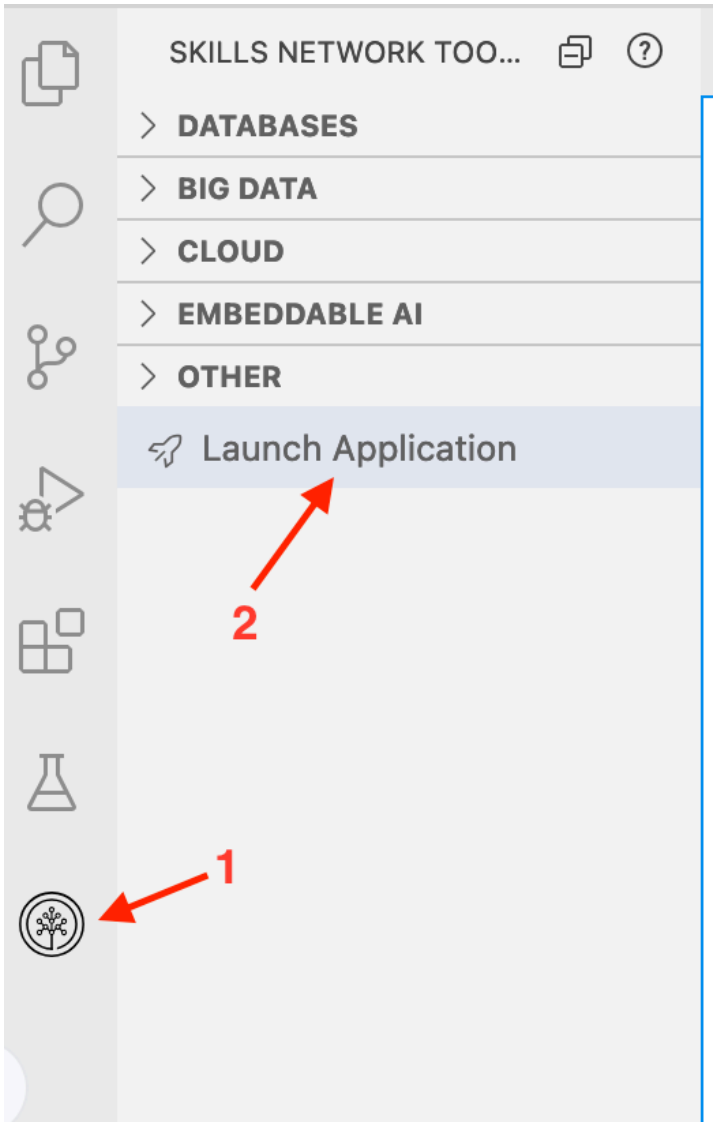
```
(venv) theiagtmeia-skillsyquath:/home/project/nutricoach$ python app.py
INFO:root:Extracting ingredients from image...
* Running on local URL:  http://127.0.0.1:5000
INFO:httpx:HTTP Request: GET http://127.0.0.1:5000/gradio_api/startup-events "HTTP/1.1 200 OK"
INFO:httpx:HTTP Request: HEAD http://127.0.0.1:5000/ "HTTP/1.1 200 OK"

To create a public link, set `share=True` in `launch()`.
INFO:httpx:HTTP Request: GET https://api.gradio.app/pkg-version "HTTP/1.1 200 OK"
```

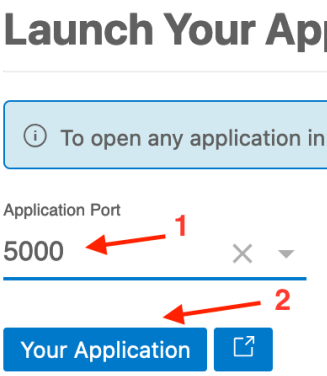
Step 5: Congrats! You can now launch the web application

Since the web application is hosted locally on port 5000, click on the following button to view the application we've developed.

Web Application

Note: If this "Web Application" button does not work, follow the following picture instructions to launch the application.





Once you launch your application, a window opens and you should be able to see the application view similar to the following example:

Upload an image and test out the application! After you hit the `Analyze` button, it takes around 20-30 seconds for the dietary crew (our multi-agent system) to analyze the uploaded image and provide output according to the selected workflow. **If the application fails, refresh it and try again.**

To stop execution of `app.py` in addition to closing the application tab, hit `Ctrl+C` in the terminal.

Step 6: You have completed the project!

Congratulations on successfully building your NourishBot application! You now have two options: continue with the rest of this tutorial to dive deeper into the codebase and enhance your understanding, or stop here if you're satisfied with your progress. The choice is yours!

Setting up your development environment

Before we dive into development, let's set up your project environment in the Cloud IDE. This environment is based on Ubuntu 22.04 and provides all the tools you need to build your AI-driven Flask application.

Note: If you completed the previous 'Quick version', run the following command in your terminal before proceeding with the rest of the instructions. If not, you can safely disregard this note and do not run this code.

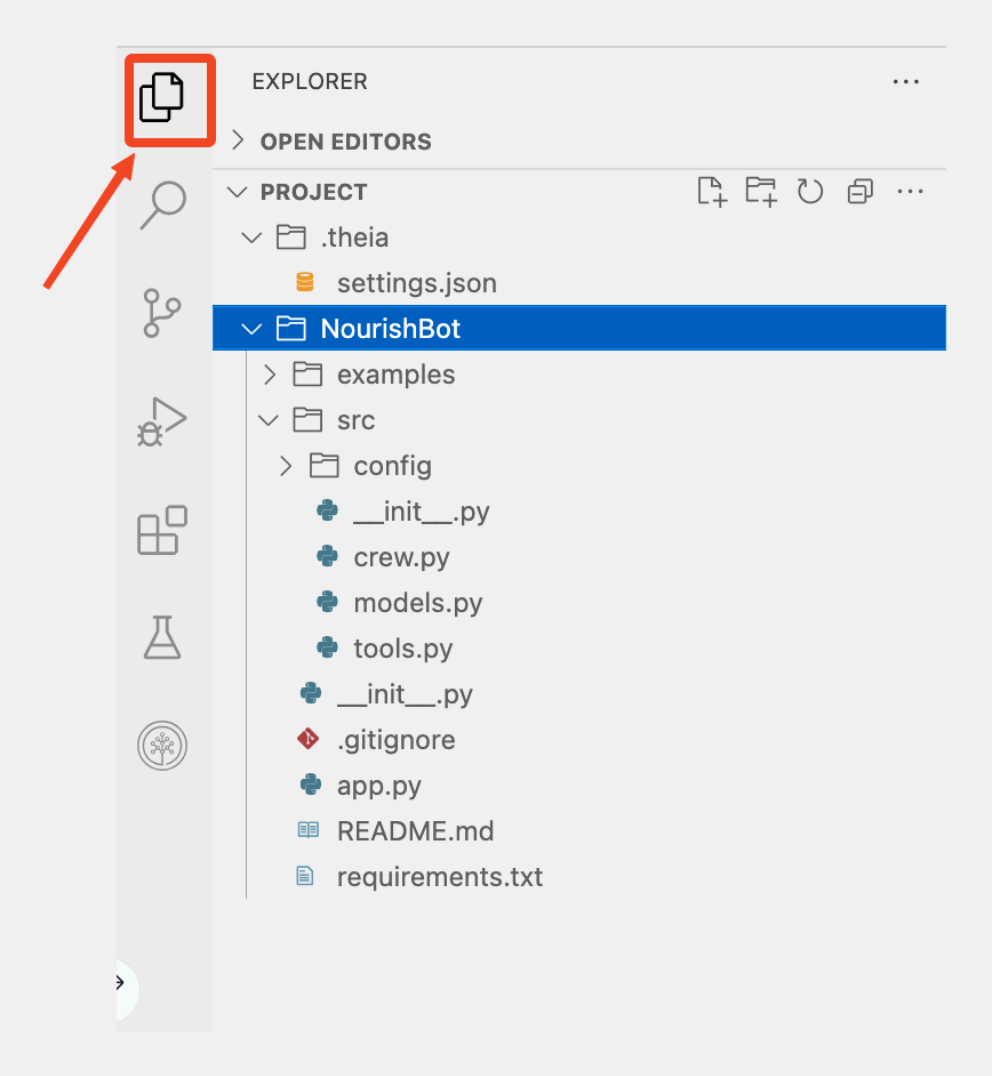
```
cd ..
rm -rf NourishBot
deactivate
```

Step 1: Create your project directory

1. Open the terminal in Cloud IDE and run the following:

```
git clone --no-checkout https://github.com/HaileyTQuach/Smart-Nutritional-App.git NourishBot
cd NourishBot
git checkout 1-start
```

Once you are all set up, select File Explorer, then the `NourishBot` folder. You should have all the files structured as below.



2. Next, we will set up a virtual environment for the project and install all the required libraries.

Note, you can run the snippet directly by clicking on the run button >_. It will take approximately **3-5 minutes** to install all the required libraries. Feel free to go grab a coffee while you wait!

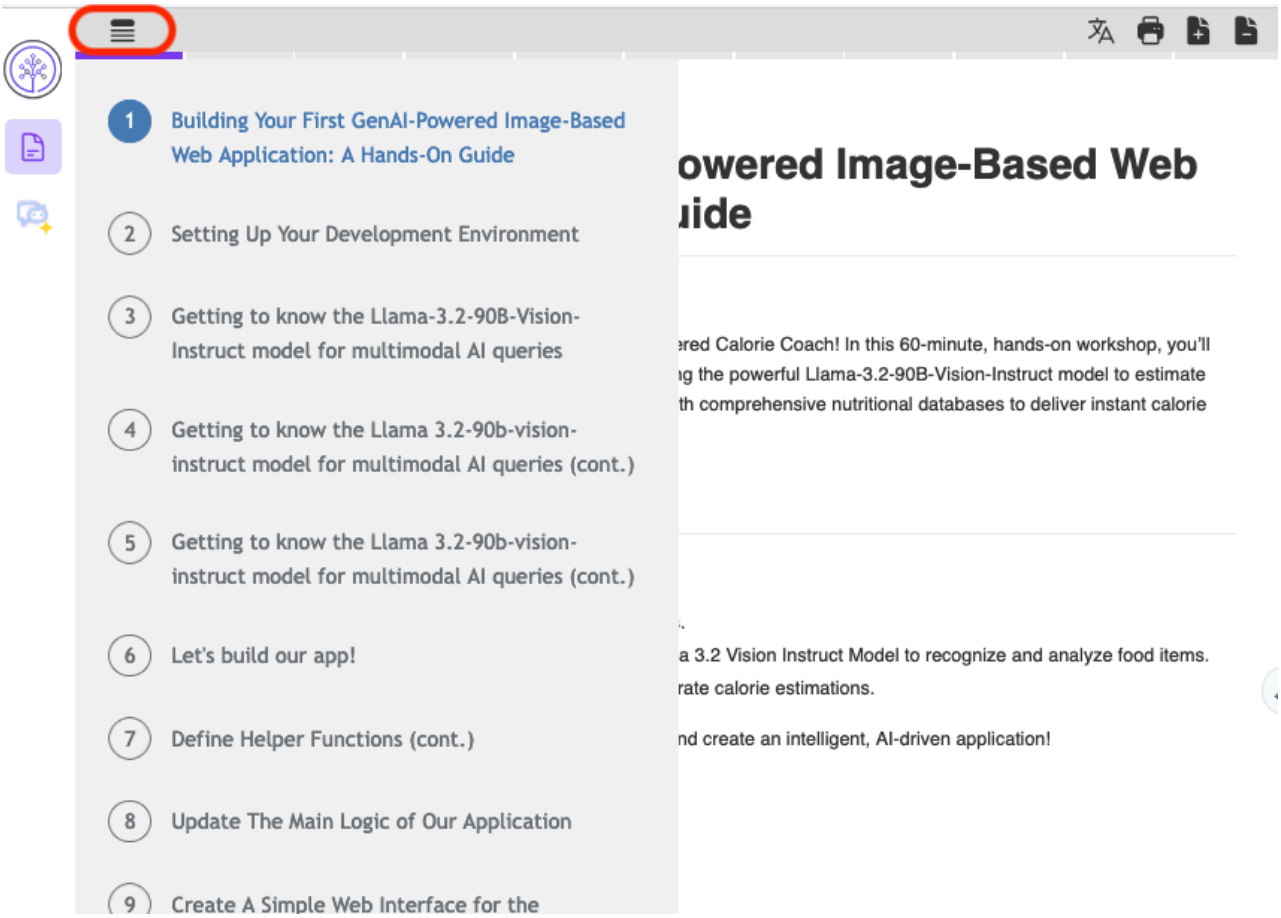
This command installs `crewai`, which orchestrates AI agents for task management, and `gradio`, which builds the user-friendly web interface. Additionally, `ibm_watsonx_ai` is crucial for ingredient detection and nutritional analysis, while `langchain` and `fastapi` handle natural language generation and asynchronous operations, respectively. Other key packages include `pydantic` for data validation, `pillow` for image processing, and `requests` for API communication. Together, these packages enable the app's core functionalities, from AI-driven workflows to user interaction.

```
python3.11 -m venv venv
source venv/bin/activate # activate venv
pip install -r requirements.txt
```

Now that your environment is set up, you're ready to start building your AI NourishBot!

Pre-requisite: Getting to know the Llama-3.2-90B-Vision-Instruct model for multimodal AI queries (Optional)

Disclaimer: Feel free to skip this step if you are already familiar with the model.



The Meta Llama 3.2-90b-vision-instruct model is a multimodal large language model (LLM) developed by Meta and designed to handle complex image recognition and visual reasoning tasks. With 90 billion parameters, this model excels at understanding and generating text based on visual inputs. It supports image captioning, visual question answering (VQA), and general image-to-text tasks, making it ideal for applications where interpreting and interacting with images is key.

In this section, we'll get familiar with the model's capabilities by performing a series of simple tasks through the script `multimodal_queries.py` before diving into the main application. This section follows the tutorial, Use Llama 3.2-90b-vision-instruct for multimodal AI queries in Python with watsonx.

Click on the following button to create a file called `multimodal_queries.py` in the NourishBot directory.

Open **multimodal_queries.py** in IDE

Let's start by adding in imports. This code imports the required modules to authenticate, interact with the API, define models, and set parameters. **Note**: From this step onwards, you can copy and paste the code snippets into `multimodal_queries.py`.

```
import requests
import base64
import os
from ibm_watsonx_ai import Credentials
from ibm_watsonx_ai import APIClient
from ibm_watsonx_ai.foundation_models import Model, ModelInference
from ibm_watsonx_ai.foundation_models.schema import TextChatParameters
from ibm_watsonx_ai.metanames import GenTextParamsMetaNames
from PIL import Image
```

Next, we'll set up the Credentials object to authenticate with IBM watsonx AI. The API key would normally be added for secure access. An instance of APIClient will be created, enabling us to interact with the IBM watsonx API.

```
credentials = Credentials(
                  url = "https://us-south.ml.cloud.ibm.com",
                  # api_key = "<YOUR_API_KEY>" # Normally you'd put an API key here, but we've got you covered here
                 )
client = APIClient(credentials)
```

Then, we load some test images.

```
url_image_1 = 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/5uo16pKhdB1f2Vz7H8Utkg/image-1.png'
url_image_2 = 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/fsuegY1q_OxKIxNhf6zeYg/image-2.png'
url_image_3 = 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/KCh_pM9BVWq_ZdzIBIA9Fw/image-3.png'
url_image_4 = 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/VaaYLw52RaykwrE3jpFv7g/image-4.png'
image_urls = [url_image_1, url_image_2, url_image_3, url_image_4]
```

To gain a better understanding of our data input, let's display the images.

Test image 1: A street view

Test image 2: A woman running

Test image 3: A flooded field

Test image 4: Nutrition label on a product

To encode these images in a way that is digestible for the LLM, we will be encoding the images to bytes that we then decode to UTF-8 representation.

```
encoded_images = []
for url in image_urls:
    encoded_images.append(base64.b64encode(requests.get(url).content).decode("utf-8"))
```

Now that our images can be passed to the LLM, let's set up an instance of the `llama-3-2-90b-vision-instruct` model through `ibm-watsonx-ai` library.

```
model_id = "meta-llama/llama-3-2-90b-vision-instruct"
project_id = "skills-network"
params = TextChatParameters()
model = ModelInference(
    model_id=model_id,
    credentials=credentials,
    project_id=project_id,
    params=params
)
```

Pre-requisite: Getting to know the Llama 3.2-90b-vision-instruct model for multimodal AI queries (Optional) (cont.)

Next, we define a function to generate responses from the model.

```
def generate_model_response(encoded_image, user_query, assistant_prompt="You are a helpful assistant. Answer the following user quer
    """
    Sends an image and a query to the model and retrieves the description or answer.
    Parameters:
    - encoded_image (str): Base64-encoded image string.
    - user_query (str): The user's question about the image.
    - assistant_prompt (str): Optional prompt to guide the model's response.
    Returns:
    - str: The model's response for the given image and query.
    """

    # Create the messages object
    messages = [
        {
            "role": "user",
            "content": [
                {
                    "type": "text",
                    "text": assistant_prompt + user_query
                },
                {
                    "type": "image_url",
                    "image_url": {
                        "url": "" + encoded_image,
                    }
                }
            ]
        }
    ]

    # Send the request to the model
    response = model.chat(messages=messages)

    # Return the model's response
    return response['choices'][0]['message']['content']
```

Code explanation

The `generate_model_response` function is designed to interact with a multimodal AI model that accepts both text and image inputs. This function takes an image, along with a user's query, and generates a response from the model.

Function purpose

The function sends an image and a query to the AI model and retrieves a description or answer. It combines a text-based prompt and an image to guide the model in generating a concise response.

Parameters

- **`encoded_image`** (`str`): A base64-encoded image string, which allows the model to process the image data.
- **`user_query`** (`str`): The user's question about the image, providing context for the model to interpret the image and answer appropriately.
- **`assistant_prompt`** (`str`): An optional text prompt to guide the model in responding in a specific way. By default, the prompt is set to: `"You are a helpful assistant. Answer the following user query in 1 or 2 sentences:"`.

Steps explained

1. **Create the Messages object:**
   The function constructs a list of messages in JSON-like format. This object includes:
   - A "user" role with a "content" array. The content array contains:
     - A text field, combining the `assistant_prompt` and the `user_query`.
     - An image URL field, which includes a base64-encoded image string. This is essential for sending image data to the model.
2. **Send the request to the model:** `response = model.chat(messages=messages)`
   The function sends the constructed messages to the model using a chat-based API. The model.chat function is invoked with the messages parameter to generate the model's response.
3. **Return the model's response:** `return response['choices'][0]['message']['content']`
   The model's response is returned as a string, extracted from the response object. Specifically, the function retrieves the content of the first choice in the model's response.

Image captioning

Now, we can loop through our images to see the text descriptions produced by the model in response to the query, "Describe the photo".

```
user_query = "Describe the photo"
for i in range(len(encoded_images)):
    image = encoded_images[i]
    response = generate_model_response(image, user_query)
    # Print the response with a formatted description
    print(f"Description for image {i + 1}: {response}")
```

Click on the button below to see the full `multimodal_queries.py` script. You can copy-paste the solution into your `multimodal_queries.py` file.

▶ Click for the script

Once you've finished writing the script, be sure to save your work.

Next, let's test the script. Click on the >_ button below.

```
python ../multimodal_queries.py
```

You should be able to see the description of each image in the terminal. An example response looks like this:

> “”
> Description for image 1: The photo depicts a bustling city street, with tall buildings and skyscrapers lining the sides of the road. The street is filled with cars, taxis, and pedestrians, creating a vibrant and energetic atmosphere.
> Description for image 2: The image depicts a woman running on a street in front of a building, with a car parked to the right. The woman is wearing a yellow hoodie, black leggings, and black shoes, and has her hair pulled back into a ponytail.
> Description for image 3: The image presents a bird’s-eye view of a flooded area, with water covering the ground and partially submerging several buildings. The scene is characterized by a large expanse of dark brown water, punctuated by the presence of trees, buildings, and silos, which are only partially visible above the surface.
> Description for image 4: The image is a close-up of a nutrition label, with a finger pointing to the “Total Fat” section. The label is white with black text and is attached to a purple container, possibly a food product.
> “”

Knowledge Check

What You’ll Do

- Click on the button below to open the file called `assignment-1.py`.
- In this file, create the main function that sends images and questions to the AI model, then test it out with 4 different images!

Open **assignment-1.py** in IDE

Why This Matters

This function is the core of any multimodal AI app. Once you build it, you’ll understand exactly how to talk to vision AI models.

Your Task

Part 1: Build the function
Complete the `generate_model_response` function that:

- Takes a base64 image, a question, and an optional prompt
- Packages them up for the AI model
- Returns the AI’s answer

Part 2: Test your code

- Image 1 (street): “Describe this image”
- Image 2 (woman running): “What is the person wearing?”
- Image 3 (flooded field): “What weather condition is shown in this image?”
- Image 4 (nutrition label): “What is the serving size listed on this label?”

When you are ready, run your code in the terminal with the command below:

```
python assignments/assignment-1.py
```

Expected Output
Your program should output something like:

```
“”
=== Test 1: Basic Image Description ===
Query: Describe this image
Response: [Model’s description of the street scene]
=== Test 2: Specific Object Detection ===
Query: What is the person wearing?
Response: [Model’s description of clothing]
=== Test 3: Scene Analysis ===
Query: What weather condition is shown in this image?
Response: [Model’s analysis of weather]
=== Test 4: Text Recognition ===
Query: What is the serving size listed on this label?
Response: [Model’s extracted serving size]
All tests completed successfully! ✓
“”
```

Click on the button below to see the solution.

▶ Click for the script

Pre-requisite: Introduction to multi-agent system and CrewAI (Optional)

1. What is agentic AI?

Agentic AI describes a system or program that can independently execute tasks for a user or another system. It autonomously designs workflows, utilizes available tools, makes decisions, takes actions, solves complex problems, and interacts with external environments, extending its capabilities beyond the data used to train its machine learning (ML) models.

2. What is a multi-agent system (MAS)?

A multi-agent system (MAS) is composed of multiple artificial intelligence (AI) agents collaborating to carry out tasks for a user or another system.

3. Key relationship between agentic AI and MAS

Characteristic	Agentic AI	Multi-Agent Systems (MAS)
Autonomy	Central focus—autonomous task execution	May include agents with varying levels of autonomy
Interaction	Limited to tools, systems, or environments	Key focus—agents interact, communicate, and coordinate
Scope	Individual agent	Multiple agents in a shared system
Dependency	Agentic AI can exist independently	MAS may involve agentic AI but doesn't require it

4. What is CrewAI?

CrewAI is an innovative open-source framework designed to orchestrate autonomous AI agents, enabling them to collaborate seamlessly to accomplish complex tasks. By assigning specific roles, tools, and objectives to each agent, CrewAI facilitates the creation of AI teams that function much like a human crew, working together to achieve your goals.

Source: CrewAI Introduction

5. Key features

- **Role-Based Agents**: Develop specialized agents with defined roles and expertise, ranging from researchers to analysts to writers.
- **Flexible Tools**: Equip agents with custom tools and APIs to interact with external services and data sources.
- **Intelligent Collaboration**: Enable agents to share insights and coordinate tasks, enhancing their ability to achieve complex objectives.

- **Task Management**: Define sequential or parallel workflows, with agents automatically handling task dependencies to ensure efficient execution.

6. How CrewAI works

CrewAI operates by organizing AI agents into a structured framework:

1. **Crew**: The top-level organization that manages AI agent teams, oversees workflows, ensures collaboration, and delivers outcomes.
2. **AI Agents**: Specialized team members with specific roles who use designated tools, can delegate tasks, and make autonomous decisions.
3. **Process**: The workflow management system that defines collaboration patterns, controls task assignments, manages interactions, and ensures efficient execution.
4. **Tasks**: Individual assignments with clear objectives that feed into the larger process, producing actionable results.

This hierarchical structure allows for efficient coordination among agents, mirroring the dynamics of a human team working together under leadership to achieve business goals.

Learn more

Explore the official CrewAI website and documentation for detailed information, tutorials, and resources to help you harness the full potential of AI agent collaboration.

Understanding NourishBot agents and tasks

NOTE: At any point throughout the project, if you encounter issues, go to the bottom of the page to see the solutions.

Now that you have all the pre-requisites to continue, let's start building our AI NourishBot app. Let's explore the files you cloned previously.

Click on Explorer to verify that all the necessary files have been created properly. Next, expand the `src` folder and check the `config` subfolder. Inside the `config` folder, you will find two pre-configured files: `agents.yaml` and `tasks.yaml`. These files are already set up with the required configurations for the project.

Let's explore `agents.yaml` first. Click on the button below to open the file.

Open **agents.yaml** in IDE

Agents and Their Roles in the NourishBot Project

In this project, several AI agents are defined in the `agents.yaml` file. Each agent plays a specific role in either analyzing the uploaded food image or generating personalized recipe suggestions. Below is a breakdown of the agents and their roles in different workflows:

---

1. ingredient_detection_agent

- **Role:** Vision AI Specialist
- **Goal:** Detect ingredients from user-uploaded images of food by leveraging advanced computer vision models to ensure no key ingredient is overlooked.
- **Backstory:** This agent is highly trained in visual recognition, capable of identifying a wide range of food ingredients, even under varying image qualities and lighting conditions.
- **Workflow:**
  - Included in the **Recipe Workflow** to detect ingredients from the uploaded image before applying dietary restrictions or generating recipes.
  - Excluded from the **Analysis Workflow** since ingredient detection is not required for detailed nutrient breakdown.

---

2. dietary_filtering_agent

- **Role:** Nutritionist AI Specialist
- **Goal:** Filter the detected ingredients based on the user's specified dietary restrictions, ensuring only compatible ingredients remain.
- **Backstory:** With deep knowledge of various diets (e.g., vegan, gluten-free, keto), this agent ensures that users receive ingredient lists that fit their dietary needs.
- **Workflow:**
  - Included in the **Recipe Workflow** to ensure the suggested recipes adhere to user-specific dietary preferences.
  - Not applicable in the **Analysis Workflow**, where the primary goal is nutrient evaluation, not ingredient filtering.

---

3. nutrient_analysis_agent

- **Role:** Nutrition Analysis Specialist
- **Goal:** Provide a detailed nutrient breakdown, including macronutrients (protein, carbs, fats), micronutrients (vitamins, minerals), and caloric content.
- **Backstory:** This agent offers expert-level nutritional insights, helping users understand the health value of their meals.
- **Workflow:**
  - Core component of the **Analysis Workflow**, providing the main output of detailed nutritional information.
  - Excluded from the **Recipe Workflow**, which focuses on ingredient filtering and recipe generation.

---

4. recipe_suggestion_agent

- **Role:** Recipe Generation Specialist
- **Goal:** Generate creative recipe ideas using the filtered list of ingredients while ensuring they fit the user's dietary restrictions and calorie goals.
- **Backstory:** Acting as an AI chef, this agent suggests easy-to-prepare, healthy recipes based on the filtered ingredients. It aims to inspire users with diverse and exciting meal ideas.
- **Workflow:**
  - Exclusively part of the **Recipe Workflow**, where the goal is to provide tailored recipes based on the user's dietary needs.
  - Not included in the **Analysis Workflow**, as recipe suggestions are not relevant when analyzing the nutrient content of a meal.

Agents by Workflow

Recipe Workflow

- `ingredient_detection_agent`
- `dietary_filtering_agent`
- `recipe_suggestion_agent`

This workflow focuses on detecting ingredients from the image, applying dietary restrictions, and generating creative recipes that align with the user's dietary preferences.

Analysis Workflow

- `nutrient_analysis_agent`

This workflow emphasizes analyzing the food image to provide a comprehensive nutrient breakdown and evaluate the overall healthiness of the meal.

These agents collectively enable the app to deliver a complete user experience, whether the goal is personalized recipe suggestions or detailed nutritional insights.

Next, let's explore `tasks.yaml` first. Click on the button below to open the file.

Open **tasks.yaml** in IDE

Tasks and Their Roles in the NourishBot Project

The `tasks.yaml` file defines the key tasks involved in the app's workflows. Each task corresponds to a specific agent and outlines the process required to either generate recipe ideas or provide a detailed nutritional analysis. Below is a detailed breakdown of each task, its description, and how it is used in different workflows:

---

1. ingredient_detection_task

- **Description:** Detects ingredients from the uploaded image using advanced computer vision models and filters the raw list of detected items.
- **Agent:** `ingredient_detection_agent`
- **Expected Output:** A list of detected ingredients from the image.
- **Workflow:**
  - Used in the **Recipe Workflow** as the first step to identify ingredients before applying dietary restrictions.
  - Not included in the **Analysis Workflow**, where direct nutrient analysis is prioritized without ingredient detection.

---

2. dietary_filtering_task

- **Description:** Filters the detected ingredients based on the user's dietary restrictions (e.g., vegan, gluten-free) to ensure only compatible ingredients remain.
- **Agent:** `dietary_filtering_agent`
- **Expected Output:** A list of filtered ingredients based on dietary restrictions.
- **Workflow:**
  - Included in the **Recipe Workflow** to provide personalized recipe suggestions by ensuring only allowed ingredients are used.
  - Excluded from the **Analysis Workflow**, as dietary filtering is not necessary when the goal is nutritional evaluation.

---

3. nutrient_analysis_task

- **Description:** Provides a detailed nutrient breakdown, including macronutrients (protein, carbohydrates, fats), micronutrients (vitamins, minerals), and caloric content of the food. Ends with a health evaluation of the dish.
- **Agent:** `nutrient_analysis_agent`
- **Expected Output:** Nutrient analysis and calorie estimation for the dish.
- **Workflow:**
  - Core component of the **Analysis Workflow**, providing users with comprehensive nutritional insights.
  - Not included in the **Recipe Workflow**, as recipe generation does not require a nutrient breakdown.

---

4. recipe_suggestion_task

- **Description:** Generates creative recipe ideas using the filtered list of ingredients. Ensures the recipes fit the user's dietary restrictions and calorie goals.
- **Agent:** `recipe_suggestion_agent`
- **Expected Output:** A list of suggested recipes based on the filtered ingredients.
- **Workflow:**
  - Used in the **Recipe Workflow** as the final step to provide users with tailored recipe ideas based on the filtered ingredients.
  - Excluded from the **Analysis Workflow**, where recipe suggestions are not needed.

Tasks by Workflow

Recipe Workflow

- **Tasks:**
  - `ingredient_detection_task`
  - `dietary_filtering_task`
  - `recipe_suggestion_task`
- **Purpose:**
  This workflow identifies ingredients from the uploaded image, applies user-defined dietary restrictions, and generates recipe ideas that adhere to the user's preferences.

Analysis Workflow

- **Tasks:**
  - `nutrient_analysis_task`
- **Purpose:**
  This workflow focuses on providing detailed nutritional information for the uploaded food image, helping users understand the health value of their meals without requiring ingredient filtering or recipe generation.

These tasks ensure that the app can cater to different user needs—whether it's generating personalized recipes or offering in-depth nutritional insights—by following clear, well-defined processes for each workflow.

Overall, the workflows with their different agents and tasks can be summarized as the diagrams below.

Now that you understand the crews, let's work on `crew.py` next!

Knowledge Check

What You'll Do

- Click the button below to open the `agents.yaml` file for the assignment.

Open **agents.yaml** in IDE

- Complete the `TODO` sections and create three different agents using YAML configuration file to understand how agent properties affect their behavior.

Why This Matters
In professional CrewAI projects, agents are typically configured using YAML files for better organization and maintainability. This matches industry best practices.

Your Task
Create three specialized agents in the `agents.yaml` config file:

- Fast Analyzer Agent: Quick, focused, limited iterations
- Thorough Researcher Agent: Detailed, can delegate, more iterations
- Creative Writer Agent: No tools, focused on creative output

Click on the button below to see the solution.

▶ Click for the script

Defining the crews

NOTE: At any point throughout the project, if you encounter issues, go to the bottom of the page to see the solutions.

Click on the button below to open `crew.py`

Open **crew.py** in IDE

CrewBase Classes in the NourishBot Project

In this project, we have three distinct `CrewBase` classes, each representing a specific workflow for analyzing food images. Below is a breakdown of why we need separate `CrewBase` classes:

---

1. BaseNourishBotCrew

This is the foundational class that defines shared logic, including common agents and tasks. It serves as a base class from which the other specialized crews inherit.

---

2. NourishBotRecipeCrew

This crew handles the **Recipe Workflow**, which focuses on:

- Detecting ingredients in the uploaded image
- Filtering ingredients based on dietary restrictions
- Generating personalized recipe suggestions

This workflow is designed for users seeking creative and healthy meal ideas tailored to their dietary preferences.

---

3. NourishBotAnalysisCrew

This crew handles the **Analysis Workflow**, which focuses on:

- Providing a detailed breakdown of the nutritional content of the food
- Evaluating the overall healthiness of the meal

This workflow is ideal for users who want to gain insights into the nutritional value of their meals and make informed dietary choices.

Steps to Complete `crew.py`

Now that we understand the purpose of each `CrewBase` class, let's go step by step to define the agents, tasks, and workflows.

Step 1: Completing `BaseNourishBotCrew`

1. Defining Agents

Agents are responsible for performing specific tasks. In the base crew class, we'll define four agents:

1. **Ingredient Detection Agent:** Detects ingredients from the uploaded food image.
2. **Dietary Filtering Agent:** Filters ingredients based on dietary restrictions.
3. **Nutrient Analysis Agent:** Analyzes the nutrients in the detected ingredients.
4. **Recipe Suggestion Agent:** Generates recipe ideas using the filtered ingredients.

Each agent uses specific tools to perform its tasks. Tools are modular functions that agents invoke to interact with the data (e.g., extracting ingredients, filtering based on restrictions).

Code for Agents

1. **Ingredient Detection Agent**

```
@agent
def ingredient_detection_agent(self) -> Agent:
    return Agent(
        config=self.agents_config['ingredient_detection_agent'],
        tools=[
            ExtractIngredientsTool.extract_ingredient,
            FilterIngredientsTool.filter_ingredients
        ],
        allow_delegation=False,
        verbose=True
    )
```

- Uses `ExtractIngredientsTool` to identify ingredients from the image.
- Uses `FilterIngredientsTool` to process the extracted ingredients.
- Delegation is disabled (allow_delegation=False), meaning the agent performs all steps independently.

2. **Dietary Filtering Agent**

```
@agent
    def dietary_filtering_agent(self) -> Agent:
        return Agent(
            config=self.agents_config['dietary_filtering_agent'],
            tools=[DietaryFilterTool.filter_based_on_restrictions],
            allow_delegation=True,
            max_iter=6,
            verbose=True
        )
```

- Filters ingredients based on user-defined dietary restrictions using `DietaryFilterTool`.
- Limits the number of iterations (max_iter=10).

3. **Nutrient Analysis Agent**

```
@agent
def nutrient_analysis_agent(self) -> Agent:
    return Agent(
        config=self.agents_config['nutrient_analysis_agent'],
        tools=[NutrientAnalysisTool.analyze_image],
        allow_delegation=False,
        max_iter=4,
        verbose=True
    )
```

- Analyzes the nutrient content (calories, macronutrients, micronutrients) using NutrientAnalysisTool.

4. **Recipe Suggestion Agent**

```
@agent
def recipe_suggestion_agent(self) -> Agent:
    return Agent(
        config=self.agents_config['recipe_suggestion_agent'],
        allow_delegation=False,
        verbose=True
    )
```

- Generates creative recipe ideas using detected and filtered ingredients.

2. Defining Tasks

Tasks represent individual steps in a workflow. Each task is linked to a specific agent and has a clear description and expected output. Tasks can also depend on the output of previous tasks.

In the CrewAI framework, the output of a task is encapsulated within the `TaskOutput` class. This class offers a structured approach to accessing task results, supporting various formats such as raw output, JSON, and Pydantic models.

By default, the `TaskOutput` includes only the `raw` output. However, Pydantic or JSON outputs will be included only if the original Task object is explicitly configured with `output_pydantic` or `output_json`, respectively. In this project, we opt for `output_json` to allow for greater flexibility in manipulating the structure of the output. We will go over defining these custom `TaskOutput` classes later.

Code for Tasks

1. **Ingredient Detection Task**

```
@task
def ingredient_detection_task(self) -> Task:
    task_config = self.tasks_config['ingredient_detection_task']
    return Task(
        description=task_config['description'],
        agent=self.ingredient_detection_agent(),
        expected_output=task_config['expected_output']
    )
```

2. **Dietary Filtering Task**

```
@task
def dietary_filtering_task(self) -> Task:
```

```
    task_config = self.tasks_config['dietary_filtering_task']
    return Task(
        description=task_config['description'],
        agent=self.dietary_filtering_agent(),
        depends_on=['ingredient_detection_task'],
        input_data=lambda outputs: {
            'ingredients': outputs['ingredient_detection_task'],
            'dietary_restrictions': self.dietary_restrictions
        },
        expected_output=task_config['expected_output']
    )
```

3. **Nutrient Analysis Task**

```
@task
def nutrient_analysis_task(self) -> Task:
    task_config = self.tasks_config['nutrient_analysis_task']
    return Task(
        description=task_config['description'],
        agent=self.nutrient_analysis_agent(),
        expected_output=task_config['expected_output'],
        output_json=NutrientAnalysisOutput
    )
```

4. **Recipe Suggestion Task**

```
@task
def recipe_suggestion_task(self) -> Task:
    task_config = self.tasks_config['recipe_suggestion_task']
    return Task(
        description=task_config['description'],
        agent=self.recipe_suggestion_agent(),
        depends_on=['dietary_filtering_task'],
        input_data=lambda outputs: {
            'filtered_ingredients': outputs['dietary_filtering_task']
        },
        expected_output=task_config['expected_output'],
        output_json=RecipeSuggestionOutput
    )
```

Step 2: Completing `NourishBotRecipeCrew`

The `NourishBotRecipeCrew` class inherits from `BaseNourishBotCrew` and defines the Recipe Workflow, which includes the following tasks:

- Ingredient Detection Task
- Dietary Filtering Task
- Recipe Suggestion Task

Code to Add:

```
@CrewBase
class NourishBotRecipeCrew(BaseNourishBotCrew):
    @crew
    def crew(self) -> Crew:
        tasks = [
            self.ingredient_detection_task(),
            self.dietary_filtering_task(),
            self.recipe_suggestion_task()
        ]
        agents = [
            self.ingredient_detection_agent(),
            self.dietary_filtering_agent(),
            self.recipe_suggestion_agent()
        ]
        return Crew(
            agents=agents,
            tasks=tasks,
            process=Process.sequential,
```

```
            verbose=True
        )
```

Step 3: Completing `NourishBotAnalysisCrew`

The `NourishBotAnalysisCrew` class handles the Analysis Workflow, which includes only one task, since the `Llama vision instruct model` alone is powerful enough to analyze the image and provide insights:

- Nutrient Analysis Task

Code to Add:

```
@CrewBase
class NourishBotAnalysisCrew(BaseNourishBotCrew):
    @crew
    def crew(self) -> Crew:
        tasks = [
            self.nutrient_analysis_task(),
        ]
        agents = [
            self.nutrient_analysis_agent(),
        ]
        return Crew(
            agents=agents,
            tasks=tasks,
            process=Process.sequential,
            verbose=True
        )
```

Summary

By following these steps, you will complete the `crew.py` file with:

1. `BaseNourishBotCrew` class that defines shared agents and tasks.
2. `NourishBotRecipeCrew` class that handles personalized recipe suggestions.
3. `NourishBotAnalysisCrew` class that provides detailed nutritional analysis.

This modular design ensures flexibility and scalability, making it easy to add new workflows or modify existing ones in the future.

Click on the button below to see the fully updated `crew.py` and copy-paste the code into your starter file.

▶ Click for the solution

Now that the core crews are in place, the next step is to define some helper files to support the crews in successfully completing their tasks.

Knowledge Check - Implement a Bookbuddy crew

What This Crew Does
The BookBuddyCrew is a simple two-step workflow that takes a short book blurb (a sentence describing a story) and:

- Detects the genre of the story (e.g., fantasy, mystery, sci-fi, romance).
- Generates a one-line tagline based on that genre and the blurb.

This is a toy example designed to help you practice setting up agents, tasks, and assembling them into a Crew — just like you would in a professional CrewAI project, but with a much simpler domain than the full NourishBot example.

What You'll Do

- Click the button below to open the `bookbuddy.py` file for the assignment.

Open **bookbuddy.py** in IDE

- Complete the `TODO`s to define:
  - Two agents (genre detector, tagline writer).
  - Two tasks (detect genre, write tagline).
- Load their properties from the provided YAML config files (agents.yaml and tasks.yaml).
- Assemble the agents and tasks into a sequential crew (BookBuddyCrew).
- Run the script to test the workflow on a sample blurb.

```
python assignments/bookbuddy.py
```

- If you get stuck, click on the button below to open the solution file.

Open **bookbuddy_solution.py** in IDE

Why This Matters
This exercise gives you hands-on practice with the core building blocks of CrewAI:

- Defining agents with roles and behaviors.
- Creating tasks with clear inputs/outputs.
- Running everything in a controlled workflow.

These same steps scale up to bigger projects where crews handle complex analysis and decision-making.

Defining tools for agents

NOTE: At any point throughout the project, if you encounter issues, go to the bottom of the page to see the solutions.

In this step, we will update `tools.py` which provide our agents with the tools needed for their workflows.

First, click the button below to open `tools.py`:

Open **tools.py** in IDE

Why Do We Need Tools?

Tools in this project are specialized functions that interact with external APIs, models, or perform specific operations. They:

1. **Automate Complex Tasks**: Extracting ingredients, filtering dietary items, and analyzing nutrients can be challenging tasks, but tools simplify them.
2. **Integrate AI Models**: Tools call the IBM WatsonX AI models to process images and textual inputs.
3. **Standardize Workflow**: Each tool has a specific purpose, ensuring clarity and reusability.

Steps to Complete `tools.py`

Step 1: Define `ExtractIngredientsTool`

The **ExtractIngredientsTool** processes the uploaded image to extract a list of food ingredients.

1. **Purpose**: Extract ingredients from the uploaded image using Llama vision instruct model.
2. **Logic**:
   - Accept the `image_input` (file path or URL).
   - Encode the image to Base64.
   - Send it to the model for ingredient detection.
3. **Implementation**:

```
class ExtractIngredientsTool():
    @tool("Extract ingredients")
    def extract_ingredient(image_input: str):
        """
        Extract ingredients from a food item image.

        :param image_input: The image file path (local) or URL (remote).
        :return: A list of ingredients extracted from the image.
        """
        if image_input.startswith("http"):  # Check if input is a URL
            # Download the image from the URL
            response = requests.get(image_input)
            response.raise_for_status()
            image_bytes = BytesIO(response.content)
        else:
            # Open the local image file in binary mode
            if not os.path.isfile(image_input):
                raise FileNotFoundError(f"No file found at path: {image_input}")
            with open(image_input, "rb") as file:
                image_bytes = BytesIO(file.read())
        # Encode the image to a base64 string
        encoded_image = base64.b64encode(image_bytes.read()).decode("utf-8")
        # Call the model with the encoded image
        model = ModelInference(
            model_id="meta-llama/llama-3-2-90b-vision-instruct",
            credentials=credentials,
            project_id=project_id,
            params={"max_tokens": 300},
        )
        response = model.chat(
            messages=[
                {
                    "role": "user",
                    "content": [
```

```
                    {"type": "text", "text": "Extract ingredients from the food item image"},
                    {"type": "image_url", "image_url": {"url": "" + encoded_image}}
                ],
            }
        ]
    )
    return response['choices'][0]['message']['content']
```

Step 2: Define `FilterIngredientsTool`

The **FilterIngredientsTool** processes raw ingredients to clean and refine the list.

1. **Purpose:** Filter irrelevant or noisy data from the list of detected ingredients.
2. **Logic:**
   - Accept the raw ingredient string.
   - Split the string into individual items.
   - Clean each item (e.g., strip extra spaces, convert to lowercase).
3. **Implementation:**

```
class FilterIngredientsTool:
    @tool("Filter ingredients")
    def filter_ingredients(raw_ingredients: str) -> List[str]:
        """
        Processes the raw ingredient data and filters out non-food items or noise.

        :param raw_ingredients: Raw ingredients as a string.
        :return: A list of cleaned and relevant ingredients.
        """
        # Example implementation: parse the raw ingredients string into a list
        # This can be enhanced with more sophisticated parsing as needed
        ingredients = [ingredient.strip().lower() for ingredient in raw_ingredients.split(',') if ingredient.strip()]
        return ingredients
```

Step 3: Define `DietaryFilterTool`

The **DietaryFilterTool** is responsible for filtering the cleaned ingredient list based on user-provided dietary restrictions.

1. **Purpose:** Ensure the ingredient list aligns with dietary requirements (e.g., vegan, gluten-free).
2. **Logic:**
   - Accept the ingredient list and dietary restrictions.
   - Use an AI model to filter out incompatible ingredients.
3. **Implementation:**

```
class DietaryFilterTool:
    @tool("Filter based on dietary restrictions")
    def filter_based_on_restrictions(ingredients: List[str], dietary_restrictions: Optional[str] = None) -> List[str]:
        """
        Uses an LLM model to filter ingredients based on dietary restrictions.
        :param ingredients: List of ingredients.
        :param dietary_restrictions: Dietary restrictions (e.g., vegan, gluten-free). Defaults to None.
        :return: Filtered list of ingredients that comply with the dietary restrictions.
        """
        # If no dietary restrictions are provided, return the original ingredients
        if not dietary_restrictions:
            return ingredients
        # Initialize the WatsonX model
        model = ModelInference(
            model_id="ibm/granite-3-8b-instruct",
            credentials=credentials,
            project_id=project_id,
            params={"max_tokens": 150},
        )
        # Create a prompt for the LLM to filter ingredients
        prompt = f"""
        You are an AI nutritionist specialized in dietary restrictions.
        Given the following list of ingredients: {', '.join(ingredients)},
        and the dietary restriction: {dietary_restrictions},
        remove any ingredient that does not comply with this restriction.
        Return only the compliant ingredients as a comma-separated list with no additional commentary.
```

```
        """
        # Send the prompt to the model for filtering
        response = model.chat(
            messages=[
                {
                    "role": "user",
                    "content": [
                        {"type": "text", "text": prompt}
                    ],
                }
            ]
        )
        # Parse the response to return the filtered list
        filtered = response['choices'][0]['message']['content'].strip().lower()
        filtered_list = [item.strip() for item in filtered.split(',') if item.strip()]
        return filtered_list
```

Step 4: Define `NutrientAnalysisTool`

The **NutrientAnalysisTool** analyzes the uploaded image for a detailed nutritional breakdown.

1. **Purpose:** Provide calorie estimates and key nutrient data from the image.
2. **Logic:**
   - Accept the image_input (file path or URL).
   - Encode the image to Base64.
   - Call the AI model for nutrient analysis.
   - Return the detailed breakdown (e.g., protein, carbs, fats).
3. **Implementation:**

```
class NutrientAnalysisTool():
    @tool("Analyze nutritional values and calories of the dish from uploaded image")
    def analyze_image(image_input: str):
        """
        Provide a detailed nutrient breakdown and estimate the total calories of all ingredients from the uploaded image.

        :param image_input: The image file path (local) or URL (remote).
        :return: A string with nutrient breakdown (protein, carbs, fat, etc.) and estimated calorie information.
        """
        if image_input.startswith("http"):  # Check if input is a URL
            # Download the image from the URL
            response = requests.get(image_input)
            response.raise_for_status()
            image_bytes = BytesIO(response.content)
        else:
            # Open the local image file in binary mode
            if not os.path.isfile(image_input):
                raise FileNotFoundError(f"No file found at path: {image_input}")
            with open(image_input, "rb") as file:
                image_bytes = BytesIO(file.read())
        # Encode the image to a base64 string
        encoded_image = base64.b64encode(image_bytes.read()).decode("utf-8")
        # Call the model with the encoded image
        model = ModelInference(
            model_id="meta-llama/llama-3-2-90b-vision-instruct",
            credentials=credentials,
            project_id=project_id,
            params={"max_tokens": 300},
        )
        # Assistant prompt (can be customized)
        assistant_prompt = """
            You are an expert nutritionist. Your task is to analyze the food items displayed in the image and provide a detailed nut
        1. **Identification**: List each identified food item clearly, one per line.
        2. **Portion Size & Calorie Estimation**: For each identified food item, specify the portion size and provide an estimated r
        - **[Food Item]**: [Portion Size], [Number of Calories] calories
        Example:
        *   **Salmon**: 6 ounces, 210 calories
        *   **Asparagus**: 3 spears, 25 calories
        3. **Total Calories**: Provide the total number of calories for all food items.
        Example:
        Total Calories: [Number of Calories]
        4. **Nutrient Breakdown**: Include a breakdown of key nutrients such as **Protein**, **Carbohydrates**, **Fats**, **Vitamins
        Example:
        *   **Protein**: Salmon (35g), Asparagus (3g), Tomatoes (1g) = [Total Protein]
        5. **Health Evaluation**: Evaluate the healthiness of the meal in one paragraph.
        6. **Disclaimer**: Include the following exact text as a disclaimer:
        The nutritional information and calorie estimates provided are approximate and are based on general food data.
        Actual values may vary depending on factors such as portion size, specific ingredients, preparation methods, and individual
        For precise dietary advice or medical guidance, consult a qualified nutritionist or healthcare provider.
        Format your response exactly like the template above to ensure consistency.
        """
        response = model.chat(
            messages=[
                {
                    "role": "user",
                    "content": [
                        {"type": "text", "text": assistant_prompt},
                        {"type": "image_url", "image_url": {"url": "" + encoded_image}}
```

```
                ],
            }
        ]
    )
    return response['choices'][0]['message']['content']
```

By following these instructions, you'll complete `tools.py` and ensure it integrates seamlessly with the rest of the NourishBot system.

Click the below button to see the fully updated `tools.py` and copy-paste the code into your starter file.

▶ Click to see the solution

Now that we've provided our agents with the tools they need for their jobs, we'll move on to define some custom `TaskOutput` classes to turn agents' output from `raw` data to `json` data.

Knowledge Check - Implement tools for Bookbuddy crew

What You'll Do

- Click the button below to open `tools.py`.

Open **tools.py** in IDE

- Implement one tool:
    - `TitleCaser.enforce(text: str) -> str` — convert a tagline to Title Case.
- Attach the tools to `Tagline Writer Agent` in your completed `bookbuddy.py` Python file from the previous knowledge check.
- Re-run the crew and confirm the tagline is title-cased (printed in logs).
- If you get stuck, click on the button below to open the solution file.

Open **tools_solution.py** in IDE

Open **bookbuddy_with_tools_solution.py** in IDE

Why This Matters
Real agents rarely act alone; they compose small, testable utilities. This exercise builds the habit of clean separation of concerns: generation (agent) vs post-processing and validation (tools).

Click on the button below to see hints for building `TitleCaser` tool.

▶ Click for hints

Defining custom `TaskOutput` classes

NOTE: At any point throughout the project, if you encounter issues, go to the bottom of the page to see the solutions.

First, click the button below to open `models.py`.

Open **models.py** in IDE

In this file, we define data models that structure the output of various agents and tasks in the NourishBot system. These models are built using **Pydantic**, a library for data validation and settings management in Python. Each model represents a specific type of data used in the application's workflows, ensuring consistency, clarity, and ease of integration. Read more about CrewAI's `TaskOutput` in their official documentation.

Additionally, these models allow us to:

1. **Generate JSON Outputs**: Pydantic makes it simple to serialize these models into JSON format, enabling easy integration with APIs or other tools.
2. **Extract Relevant Fields**: Once the output is in JSON format, we can selectively extract fields to display on the NourishBot web app or any user-facing interface.

Key Models in `models.py`

1. `Recipe`

This model represents a single recipe suggestion.

- **Fields**:
    - `title` (str): The name of the recipe.
    - `ingredients` (List[str]): A list of ingredients required for the recipe.
    - `instructions` (str): Step-by-step instructions for preparing the dish.
    - `calorie_estimate` (int): The estimated number of calories per serving.

- **Purpose**: Provides a clear structure for each recipe suggestion, ensuring consistency across recipe outputs.

---

2. `RecipeSuggestionOutput`

This model organizes multiple recipes as output from the **Recipe Suggestion Agent**.

- **Fields**:
  - `recipes` (List[Recipe]): A list of `Recipe` objects.
- **Purpose**: Represents the output of the **Recipe Suggestion Task**, containing one or more recipe suggestions.

---

3. `VitaminInfo`

This model details information about a specific vitamin.

- **Fields**:
  - `name` (str): The name of the vitamin (e.g., "Vitamin C").
  - `percentage_dv` (str): The percentage of the Daily Value (%DV) provided by the food.
- **Purpose**: Structures data about vitamins for inclusion in the nutrient breakdown.

---

4. `MineralInfo`

This model details information about a specific mineral.

- **Fields**:
  - `name` (str): The name of the mineral (e.g., "Calcium").
  - `amount` (str): The amount and unit of the mineral (e.g., "100mg").
- **Purpose**: Structures data about minerals for inclusion in the nutrient breakdown.

---

5. `NutrientBreakdown`

This model provides a detailed breakdown of the nutrients in a dish.

- **Fields**:
  - `protein` (Optional[str]): Protein content of the dish.
  - `carbohydrates` (Optional[str]): Carbohydrate content of the dish.
  - `fats` (Optional[str]): Fat content of the dish.
  - `vitamins` (List[VitaminInfo]): A list of vitamin details.
  - `minerals` (List[MineralInfo]): A list of mineral details.
- **Purpose**: Aggregates detailed nutritional information for use in analysis tasks.

---

6. `NutrientAnalysisOutput`

This is the comprehensive output model for the **Nutrient Analysis Task**.

- **Fields**:
  - `dish` (Optional[str]): The identified name of the dish (if recognized).
  - `portion_size` (Optional[str]): A description of the portion size (e.g., "1 cup").
  - `estimated_calories` (Optional[int]): The estimated number of calories per portion.
  - `nutrients` (NutrientBreakdown): A detailed breakdown of nutrients.
  - `health_evaluation` (Optional[str]): A summary evaluating the healthiness of the dish.
- **Purpose**: Provides structured output for detailed nutritional analysis, including calories, nutrients, and health evaluation.

Why Do We Need These Models?

1. **Structured Output**: By defining these models, we ensure all outputs are consistent in format and structure.
2. **Validation**: Pydantic automatically validates the data, ensuring that the outputs are accurate and error-free.
3. **JSON Serialization**: These models can be serialized into JSON format, which is essential for:
   - Displaying outputs on the NourishBot web app.
   - Sharing data with other APIs or services.
4. **Field Extraction**: Once in JSON format, we can easily extract relevant fields (e.g., total calories, recipe titles) to display concise and user-friendly outputs on the app interface.
5. **Scalability**: If new features or tasks are added, new fields or models can be seamlessly incorporated.

Usage in the Application

- **`RecipeSuggestionOutput`**: Used as the output for the **Recipe Suggestion Task**, providing users with recipe ideas based on filtered ingredients.
- **`NutrientAnalysisOutput`**: Used as the output for the **Nutrient Analysis Task**, offering a detailed breakdown of the food's nutrients and health evaluation.
- **`VitaminInfo`** and **`MineralInfo`**: Provide granular details about specific vitamins and minerals in the food.
- **`NutrientBreakdown`**: Aggregates data from multiple sources (e.g., detected nutrients, vitamins, minerals) into a unified format.

By following this structured approach, the models in `models.py` will serve as a robust foundation for building and expanding the NourishBot application.

Click the button below to see the fully updated `models.py` and copy-paste the code into your starter file.

▶ Click to see the solution

Having finished `models.py`, we've now prepared every component of our MAS. Next, we'll update the application's main logic—and then we'll be all set to run our app!

Defining the main logic of the application

NOTE: At any point throughout the project, if you encounter issues, go to the bottom of the page to see the solutions.

In this step, we will define the main logic of the application in `app.py`, which integrates all the components of the MAS. To get started, click the button below to open `app.py`.

Open **app.py** in IDE

In this section, we will break down the `app.py` file to understand its components and how it connects the NourishBot backend to a user-friendly Gradio interface. This file allows users to interact with the NourishBot system by uploading food images, specifying dietary preferences, and choosing between workflows for recipe suggestions or nutritional analysis.

Why Use Gradio?

Gradio is a powerful Python library for building web-based interfaces for machine learning and AI models. While we provide a simple Gradio-based interface to start with, customizing the interface further is beyond the scope of this project. You are welcome to explore more advanced Gradio features to enhance the user experience.

Key Functions in `app.py`

1. `analyze_food`

This is the core function that connects the Gradio interface to the NourishBot backend.

Purpose:

- Accepts user inputs: an uploaded image, dietary restrictions, and a workflow type.
- Saves the uploaded image locally.
- Initializes the appropriate crew (`NourishBotRecipeCrew` or `NourishBotAnalysisCrew`) based on the selected workflow.
- Executes the crew's workflow and formats the result for display.

Key Logic:

- The `workflow_type` determines which crew to use:
  - `NourishBotRecipeCrew`: For generating recipe suggestions.
  - `NourishBotAnalysisCrew`: For nutritional analysis.
- After running the workflow, the output is formatted using helper functions:
  - `format_recipe_output` for recipes.
  - `format_analysis_output` for nutritional analysis.

Implementation:

```
def analyze_food(image, dietary_restrictions, workflow_type, progress=gr.Progress(track_tqdm=True)):
    """
    Wrapper function for the Gradio interface.

    :param image: Uploaded image (PIL format)
    :param dietary_restrictions: Dietary restriction as a string (e.g., "vegan")
    :param workflow_type: Workflow type ("recipe" or "analysis")
    :return: Result from the NourishBot workflow.
    """

    # Save the uploaded image temporarily to the local file system
    # This allows the backend to access and process the image file
    image.save("uploaded_image.jpg")
    image_path = "uploaded_image.jpg"
    # Create a dictionary to store inputs for the crew workflow
    # This includes the image path, dietary restrictions, and workflow type
    inputs = {
        'uploaded_image': image_path,
        'dietary_restrictions': dietary_restrictions,
        'workflow_type': workflow_type
    }

    # Initialize the appropriate crew instance based on the selected workflow type
    if workflow_type == "recipe":
        # Use the NourishBotRecipeCrew for recipe generation
        crew_instance = NourishBotRecipeCrew(
            image_data=image_path,  # Pass the image path
            dietary_restrictions=dietary_restrictions  # Include dietary restrictions
        )
    elif workflow_type == "analysis":
        # Use the NourishBotAnalysisCrew for nutritional analysis
        crew_instance = NourishBotAnalysisCrew(
            image_data=image_path  # Pass the image path
        )
    else:
        # Handle invalid workflow types by returning an error message
        return "Invalid workflow type. Choose 'recipe' or 'analysis'."
    # Run the workflow associated with the selected crew
    crew_obj = crew_instance.crew()  # Get the crew instance
    final_output = crew_obj.kickoff(inputs=inputs)  # Execute the workflow with the provided inputs
    # Convert the final output to a dictionary format
    # This makes it easier to process and format for display
    final_output = final_output.to_dict()
```

```
    # Format the result based on the selected workflow type
    if workflow_type == "recipe":
        # Format recipe suggestions as Markdown
        recipe_markdown = format_recipe_output(final_output)
        return recipe_markdown  # Return formatted recipe output
    elif workflow_type == "analysis":
        # Format nutritional analysis as Markdown
        nutrient_markdown = format_analysis_output(final_output)
        return nutrient_markdown  # Return formatted nutritional analysis output
```

---

2. Formatting Functions

Purpose:

- Ensures the output is user-friendly and visually appealing.

`format_recipe_output`

Formats the recipe suggestions into a structured Markdown table.

Logic:

- Checks if recipes exist in the workflow output.
- For each recipe:
  - Displays its title.
  - Lists ingredients in a table.
  - Shows instructions and calorie estimates.
- Returns a Markdown-formatted string.

Implementation:

```
def format_recipe_output(final_output):
    """
    Formats the recipe output into a table-based Markdown format.

    :param final_output: The output from the NourishBotRecipe workflow.
    :return: Formatted output as a Markdown string.
    """
    # Initialize the Markdown output with a header
    output = "## 🍽️ Recipe Ideas\n\n"
    recipes = []
    # Check if the final output directly contains the "recipes" key
    if "recipes" in final_output:
        recipes = final_output["recipes"]
    else:
        # Fallback logic: Try to extract recipes from nested task outputs
        # Check if there is a task output for "recipe_suggestion_task"
        recipe_task_output = final_output.get("recipe_suggestion_task")
        if recipe_task_output and hasattr(recipe_task_output, "json_dict") and recipe_task_output.json_dict:
            # Extract recipes from the task output
            recipes = recipe_task_output.json_dict.get("recipes", [])

    if recipes:
        # Loop through each recipe and format it into Markdown
        for idx, recipe in enumerate(recipes, 1):
            # Add a numbered header for each recipe
            output += f"### {idx}. {recipe['title']}\n\n"

            # Create a Markdown table for the ingredients
            output += "**Ingredients:**\n"
            output += "| Ingredient |\n"
            output += "|------------|\n"
            for ingredient in recipe['ingredients']:
                output += f"| {ingredient} |\n"
            output += "\n"

            # Add instructions and calorie estimate for the recipe
            output += f"**Instructions:**\n{recipe['instructions']}\n\n"
            output += f"**Calorie Estimate:** {recipe['calorie_estimate']} kcal\n\n"
            output += "---\n\n"  # Add a separator between recipes
    else:
        # Handle the case where no recipes were generated
        output += "No recipes could be generated."

    # Return the formatted Markdown string
    return output
```

`format_analysis_output`

Formats the nutritional analysis results into a structured Markdown table.

Logic:

- Displays basic information such as dish name, portion size, and calorie count.
- Creates a table for macronutrients (protein, carbohydrates, fats).
- Includes separate tables for vitamins and minerals if available.
- Appends a health evaluation summary at the end.

Implementation:

```
def format_analysis_output(final_output):
    """
    Formats nutritional analysis output into a table-based Markdown format,
    including health evaluation at the end.

    :param final_output: The JSON output from the NourishBotAnalysis workflow.
    :return: Formatted output as a Markdown string.
    """
    output = "## 🥦 Nutritional Analysis\n\n"

    # Basic dish information
    if dish := final_output.get('dish'):
        output += f"**Dish:** {dish}\n\n"
    if portion := final_output.get('portion_size'):
        output += f"**Portion Size:** {portion}\n\n"
    if est_cal := final_output.get('estimated_calories'):
        output += f"**Estimated Calories:** {est_cal} calories\n\n"
    if total_cal := final_output.get('total_calories'):
        output += f"**Total Calories:** {total_cal} calories\n\n"
    # Nutrient breakdown table
    output += "**Nutrient Breakdown:**\n\n"
    output += "| **Nutrient**       | **Amount** |\n"
    output += "|--------------------|------------|\n"

    nutrients = final_output.get('nutrients', {})
    # Display macronutrients
    for macro in ['protein', 'carbohydrates', 'fats']:
        if value := nutrients.get(macro):
            output += f"| **{macro.capitalize()}** | {value} |\n"

    # Display vitamins table if available
    vitamins = nutrients.get('vitamins', [])
    if vitamins:
        output += "\n**Vitamins:**\n\n"
        output += "| **Vitamin** | **%DV** |\n"
        output += "|-------------|--------|\n"
        for v in vitamins:
            name = v.get('name', 'N/A')
            dv = v.get('percentage_dv', 'N/A')
            output += f"| {name} | {dv} |\n"

    # Display minerals table if available
    minerals = nutrients.get('minerals', [])
    if minerals:
        output += "\n**Minerals:**\n\n"
        output += "| **Mineral** | **Amount** |\n"
        output += "|-------------|-----------|\n"
        for m in minerals:
            name = m.get('name', 'N/A')
            amount = m.get('amount', 'N/A')
            output += f"| {name} | {amount} |\n"

    # Append health evaluation at the end
    if health_eval := final_output.get('health_evaluation'):
        output += "\n**Health Evaluation:**\n\n"
        output += health_eval + "\n"

    return output
```

---

3. Gradio Interface

The Gradio interface provides a web-based UI for interacting with the NourishBot system. The main components include:

- **Inputs**:
  - `gr.Image`: For uploading food images.
  - `gr.Textbox`: For specifying dietary restrictions.
  - `gr.Radio`: For selecting the workflow type (recipe or analysis).
- **Outputs**:
  - `gr.Markdown`: For displaying formatted results.
- **Submit Button**:
  - Triggers the `analyze_food` function and displays the results in the output section.

Gradio Customization

We added:

- Custom CSS (`css`) for styling.
- Custom JavaScript (`js`) for animations.

If you want to further customize the interface, you can explore Gradio's official documentation.

Implementation:

```
# Define custom CSS for styling
css = """
.title {
    font-size: 1.5em !important;
    text-align: center !important;
    color: #FFD700;
}
.text {
    text-align: center;
}
"""
js = """
function createGradioAnimation() {
    var container = document.createElement('div');
    container.id = 'gradio-animation';
    container.style.fontSize = '2em';
    container.style.fontWeight = 'bold';
    container.style.textAlign = 'center';
    container.style.marginBottom = '20px';
    container.style.color = '#eba93f';
    var text = 'Welcome to your AI NourishBot!';
    for (var i = 0; i < text.length; i++) {
        (function(i){
            setTimeout(function(){
                var letter = document.createElement('span');
                letter.style.opacity = '0';
                letter.style.transition = 'opacity 0.1s';
                letter.innerText = text[i];
                container.appendChild(letter);
                setTimeout(function() {
                    letter.style.opacity = '0.9';
                }, 50);
            }, i * 250);
        })(i);
    }
    var gradioContainer = document.querySelector('.gradio-container');
    gradioContainer.insertBefore(container, gradioContainer.firstChild);
    return 'Animation created';
}
"""
# Use a theme and custom CSS with Blocks
with gr.Blocks(theme=gr.themes.Citrus(), css=css, js=js) as demo:
    gr.Markdown("# How it works", elem_classes="title")
    gr.Markdown("Upload an image of your fridge content, enter your dietary restriction (if you have any!) and select a workflow typ
    gr.Markdown("Upload an image of a complete dish, leave dietary restriction blank and select a workflow type 'analysis' then clic
    gr.Markdown("You can also select one of the examples provided to autofill the input sections and click 'Analyze' right away!", e
    with gr.Row():
        with gr.Column(scale=1, min_width=400):
            gr.Markdown("## Inputs", elem_classes="title")
            image_input = gr.Image(type="pil", label="Upload Image")
            dietary_input = gr.Textbox(label="Dietary Restrictions (optional)", placeholder="e.g., vegan")
            workflow_radio = gr.Radio(["recipe", "analysis"], label="Workflow Type")
            submit_btn = gr.Button("Analyze")

        with gr.Column(scale=2, min_width=600):
            # Place Examples directly under the Analyze button
            gr.Examples(
                examples=[
                    ["examples/food-1.jpg", "vegan", "recipe"],
                    ["examples/food-2.jpg", "", "analysis"],
                    ["examples/food-3.jpg", "keto", "recipe"],
                    ["examples/food-4.jpg", "", "analysis"],
                ],
                inputs=[image_input, dietary_input, workflow_radio],
                label="Try an Example: Select one of the examples below to autofi;l the input section then click Analyze"
                # No function or outputs provided, so it only autofills inputs
            )
            gr.Markdown("## Results will appear here...", elem_classes="title")
            # result_display = gr.Markdown(height=800, )
            result_display = gr.Markdown(
                "<div style='border: 1px solid #ccc; "
                "padding: 1rem; text-align: center; "
                "color: #666;'>No results yet</div>",
                height=500
            )
    submit_btn.click(
        fn=analyze_food,
        inputs=[image_input, dietary_input, workflow_radio],
        outputs=result_display
    )
# Launch the Gradio interface
if __name__ == "__main__":
    demo.launch(server_name="127.0.0.1", server_port=5000)
```

How It All Works Together

1. **User Interaction**:
   - The user uploads an image, specifies dietary restrictions, and selects a workflow type.
2. **Backend Processing**:
   - The `analyze_food` function processes the inputs and initializes the corresponding crew.
   - The crew's workflow generates the result.
3. **Output Formatting**:
   - The result is passed to a formatting function (`format_recipe_output` or `format_analysis_output`) to create a user-friendly display.
4. **Display**:
   - The formatted output is displayed on the Gradio interface.

Click the button below to see the fully updated `app.py` and copy-paste the code into your starter file.

▶ Click to see solution

Explanation of Key Design Choices

Using Markdown for Results

- Markdown allows for a structured and visually appealing display of results, including tables and headings.

Separate Formatting Functions

- Splitting formatting into `format_recipe_output` and `format_analysis_output` ensures clean and maintainable code.

Why Save Uploaded Images Locally?

- Saving images temporarily allows the backend to process them easily without dealing with in-memory operations, which can be more error-prone.

Custom CSS and JavaScript

- These enhancements make the interface more user-friendly and visually engaging.

Output Formatting and JSON Models

In `analyze_food`, the final output from the workflow is converted into a dictionary using the `.to_dict()` method. This structured output aligns with the Pydantic models defined in `models.py` and allows us to:

- Extract relevant fields.
- Display them in a clear, user-friendly format.

Congratulations on completing the implementation of app.py! 🎉 You've now built a functional interface for the AI NourishBot app, enabling users to analyze food images or generate personalized recipe ideas through a seamless and user-friendly Gradio interface. In the next step, we will launch the application and test it out!

Launching the application

Step 1: Set up the files

Return to the terminal and verify that the virtual environment `venv` label appears at the start of the line. This means that you are in the `venv` environment that you just created. Next, run the following command to pull the complete code of the application.

```
git reset --hard
git checkout 5-final
git pull
```

Step 2: Run the Gradio app in the terminal

```
python app.py
```

Note: After it runs successfully, you will see a message similar to the following example in the terminal:

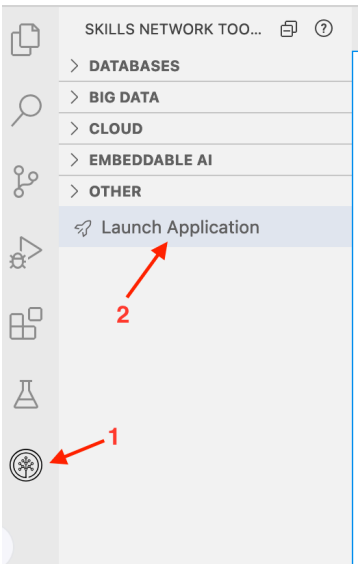
```
(venv) theiagithia-multiyquick:/home/project/nutritouch$ python app.py
INFO:root:Extracting ingredients from image...
* Running on local URL:  http://127.0.0.1:5000
INFO:httpx:HTTP Request: GET http://127.0.0.1:5000/gradio_api/startup-events "HTTP/1.1 200 OK"
INFO:httpx:HTTP Request: HEAD http://127.0.0.1:5000/ "HTTP/1.1 200 OK"

To create a public link, set `share=True` in `launch()`.
INFO:httpx:HTTP Request: GET https://api.gradio.app/pkg-version "HTTP/1.1 200 OK"
```

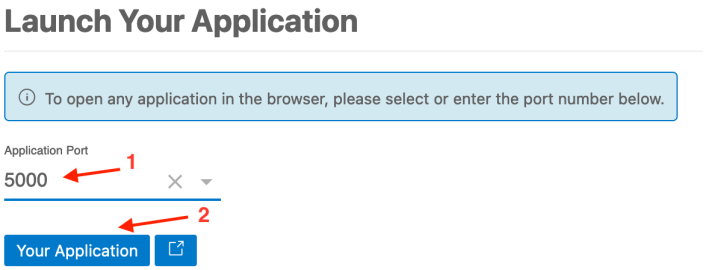
Step 3: Congrats! You can now launch the web application

Since the web application is hosted locally on port 5000, click on the following button to view the application we've developed.

Web Application

Note: If this "Web Application" button does not work, follow the following picture instructions to launch the application.





Once you launch your application, a window opens and you should be able to see the application view similar to the following example:

Upload an image and test out the application! After you hit the `Analyze` button, it takes around 20-30 seconds for the dietary crew (our multi-agent system) to analyze the uploaded image and provide output according to the selected workflow. **If the application fails, refresh it and try again.**

To stop execution of `app.py` in addition to closing the application tab, hit `Ctrl+C` in terminal.

Conclusion and next steps

Congratulations on completing the AI NourishBot project! By building this application, you've successfully explored the powerful capabilities of the Llama 3.2 90B vision-instruct Model and integrated it with AI-driven technologies to deliver practical, real-world solutions for dietary management. This marks the culmination of integrating all the components of the project—tools, agents, tasks, and workflows—into a cohesive application. With the Gradio interface, users can interact with your app intuitively and experience the power of AI-driven food analysis in real time.

As a final step, feel free to:

- Test the app with different images and dietary restrictions.
- Explore Gradio's documentation to customize the UI further if desired.
- Share your app with others or deploy it on a cloud platform to make it accessible to a broader audience.

This project showcases how advanced AI technologies like LLMs and computer vision can be combined with intuitive design to create impactful applications. Keep pushing boundaries, experimenting, and exploring new use cases for generative AI! 🚀

HAPPY LEARNING!

Author(s)

[Hailey Quach]

Other Contributor(s)

[Ricky Shi]
[Karan Goswami]
[Wojciech "Victor" Fulmyk]