DAY9 (23/04/2025) [Refrence](https://cloud.google.com/discover/what-are-ai-agents)

What is an AI agent?

AI agents are software systems that use AI to pursue goals and complete tasks on behalf of users. They show reasoning, planning, and memory and have a level of autonomy to make decisions, learn, and adapt.

Their capabilities are made possible in large part by the multimodal capacity of generative AI and AI foundation models. AI agents can process multimodal information like text, voice, video, audio, code, and more simultaneously; can converse, reason, learn, and make decisions. They can learn over time and facilitate transactions and business processes. Agents can work with other agents to coordinate and perform more complex workflows.

How do AI agents work?

Every agent defines its role, personality, and communication style, including specific instructions and descriptions of available tools.

1. **Persona**: A well defined persona allows an agent to maintain a consistent character and behave in a manner appropriate to its assigned role, evolving as the agent gains experience and interacts with its environment.
2. **Memory**: The agent is equipped in general with short term, long term, consensus, and episodic memory. Short term memory for immediate interactions, long-term memory for historical data and conversations, episodic memory for past interactions, and consensus memory for shared information among agents. The agent can maintain context, learn from experiences, and improve performance by recalling past interactions and adapting to new situations.
3. **Tools**: Tools are functions or external resources that an agent can utilize to interact with its environment and enhance its capabilities. They allow agents to perform complex tasks by accessing information, manipulating data, or controlling external systems, and can be categorized based on their user interface, including physical, graphical, and program-based interfaces. Tool learning involves teaching agents how to effectively use these tools by understanding their functionalities and the context in which they should be applied.
4. **Model**: Large language models (LLMs) serve as the foundation for building AI agents, providing them with the ability to understand, reason, and act. LLMs act as the "brain" of an agent, enabling them to process and generate language, while other components facilitate reason and action.

What are Large Language Models (LLMs)?

* Large Language Models (LLMs) are advanced AI models trained to understand and generate human language.
* They’re called "large" because:
* They’re trained on huge datasets (like books, websites, and code).
* They have billions (or even trillions) of parameters (the parts of the model that learn patterns in language).

Examples of LLMs:

* GPT-4 (by OpenAI – what you're chatting with now )
* BERT (by Google)
* LLaMA (by Meta)
* Gemini, Claude, etc.

**Why are LLMs Used?**

LLMs are used for **understanding, generating, and interacting in natural human language**. Here's **what they can do**:

| **Use Case** | **Description** |
| --- | --- |
| * Text Generation | Writing articles, blogs, emails, essays, etc. |
| * Chatbots | Powering intelligent virtual assistants (like ChatGPT, Alexa, etc.) |
| * Question Answering | Answering questions from documents, websites, or general knowledge |
| * Search Enhancement | Making search engines smarter with natural understanding |
| * Code Generation | Writing and debugging code (e.g., GitHub Copilot) |
| * Summarization | Summarizing long texts, legal docs, or research papers |
| * Translation | Translating text between different languages |
| * Classification / Sentiment Analysis | Classifying emails, reviews, or detecting emotions in text |

How Do They Work (Simple Explanation)?

* LLMs use a type of AI called transformers. Here’s how they generally work:
* Read huge amounts of text (Wikipedia, books, articles, etc.).
* Learn patterns in language — like grammar, facts, and reasoning.
* When you ask something, the model predicts what to say next, like auto-complete on steroids.

Open ai API Key = sk-proj-tKe39ojMuTmzKF2NpUibSW0F424IQVlmOAzC8\_j-Cn3oKKX-PbYnAW\_0oHPp88OWMYjJZg\_ZBrT3BlbkFJlbBrLuTRJRRPe6TTFJlzGKz19C6eyJT2G2toG8\_c9NPSLbs5FsmYhJphHjaKr6YOOxOHh1N7kA

Tasks Completed:

Initially implemented the recipe generation agent using OpenAI’s GPT-4 API.

Due to the subscription requirement of GPT-4, we shifted to using the open-source GPT4All model to run locally and avoid dependency on external APIs.

Set up the environment for GPT4All and began configuring model loading and path setup.

Explored multiple local models and researched compatibility with the GPT4All library.

Challenges & Errors Faced (GPT4All Integration):

While attempting to run GPT4All locally, we faced several roadblocks:

Model Path Errors

Missing Model Directory

Unsupported File Format

Failed DLL Load

Actions Taken:

Replaced incompatible .bin model with .gguf format.

Updated model path handling to avoid syntax issues on Windows.

Installed missing dependencies and upgraded the gpt4all Python package to its latest version.

Verified new model loading and began basic response testing.

Request:

I wanted to reach out to share that I am currently facing some challenges in implementing the agent functionality for our project. As I am still very new to this field, I find myself overwhelmed with the wide range of concepts, tools, and terminologies related to AI agent development.

At this point, I feel a bit stuck and unsure about how to proceed effectively. I believe having a clear and simple roadmap or step-by-step guidance would really help me move forward confidently and build the required functionality in a structured manner.

If possible, I would sincerely appreciate it if you could provide:

A basic outline of steps to follow for implementing a Python-based AI agent.

Key components or libraries I should focus on first.

Any recommendations on best practices or reference material.

I am committed to learning and making consistent progress on the task, and your guidance at this stage would be extremely valuable to me.

Thank you for your time and support.

Best regards,

DAY11 (24/04/2025)

MCP - Model Context Protocol

Recipe.csv

Dish Name,Ingredients,Instructions

Pav Bhaji,"Potatoes, Tomatoes, Onions, Capsicum, Butter, Pav Bhaji Masala","Boil and mash vegetables. Saute onions and tomatoes, add spices, mix mashed vegetables, and serve with buttered pav."

Biryani,"Rice, Chicken, Yogurt, Onion, Spices, Saffron","Marinate chicken in yogurt and spices. Layer with cooked rice. Steam and serve hot."

Masala Dosa,"Rice, Urad Dal, Potato, Onion, Mustard Seeds, Curry Leaves","Prepare dosa batter. Make potato filling. Cook dosa, add filling, fold, and serve."

Chole Bhature,"Chickpeas, Onion, Tomato, Ginger, Garlic, Spices, Maida","Cook chole with spices. Knead dough and fry bhature. Serve hot."

Rajma Chawal,"Rajma (Kidney Beans), Onion, Tomato, Spices, Rice","Soak and boil rajma. Cook with masala base of onion, tomato. Serve with steamed rice."

Palak Paneer,"Spinach, Paneer, Onion, Tomato, Garlic, Spices","Blanch spinach, blend to puree. Cook with spices, add paneer cubes, and simmer."

Aloo Paratha,"Wheat Flour, Potato, Onion, Coriander, Spices, Ghee","Make dough and prepare potato filling. Roll, stuff, cook on tawa with ghee."

Streamlit\_recipe\_agent.py

import streamlit as st

import pandas as pd

from fuzzywuzzy import process

# Load the dataset

df = pd.read\_csv("veg\_recipes.csv")

st.title("🍽️ AI Recipe Agent")

# Initialize session state

if "dish\_input" not in st.session\_state:

    st.session\_state.dish\_input = ""

# Reset button (comes before text\_input)

if st.button(" Reset"):

    st.session\_state.dish\_input = ""

    st.rerun()  #  This is the correct method now

# Input field

dish\_name = st.text\_input("Enter the name of the dish:", key="dish\_input")

# Recipe search logic

if dish\_name:

    matches = process.extractOne(dish\_name, df["Dish Name"])

    if matches and matches[1] > 60:

        result = df[df["Dish Name"] == matches[0]].iloc[0]

        st.success(f"\*\*Dish:\*\* {result['Dish Name']}")

        st.write(f"\*\*Ingredients:\*\* {result['Ingredients']}")

        st.write(f"\*\*Instructions:\*\* {result['Instructions']}")

    else:

        st.warning(" Sorry, no matching recipe found,we will update soon.")

streamlit\_shopping\_agent.py

import streamlit as st

import pandas as pd

from fuzzywuzzy import process

# Load recipe data

df = pd.read\_csv("recipes.csv")

st.title("🛒 Shopping Assistant")

# User input for dish name

dish\_input = st.text\_input("Enter the dish name:", key="dish\_input")

# Button to generate shopping list

if st.button("Generate Shopping List"):

    # Fuzzy match the dish

    match = process.extractOne(dish\_input, df["Dish Name"].tolist())

    if match and match[1] >= 70:

        selected\_recipe = df[df["Dish Name"] == match[0]].iloc[0]

        st.subheader(f"Ingredients for: {selected\_recipe['Dish Name']}")

        # Split and display as bullet points

        ingredients = selected\_recipe["Ingredients"].split(",")

        for item in ingredients:

            st.markdown(f"- {item.strip()}")

    else:

        st.error("Dish not found. Please try a different name.")

Let’s implement MCP in above project as:

Recipe Section:

* The user enters a dish name (e.g., "Pav Bhaji").
* The app returns the recipe details, including the ingredients and instructions.
* A shopping list is automatically generated based on the ingredients.

Shopping Assistant:

* For now, it generates a shopping list based on ingredients. Later, we can extend it to interact with an online shopping API to order ingredients.

What's Model Context Protocol (MCP)?

MCP is a pattern where agents or components share a common context or memory, allowing:

* Persistent knowledge exchange
* Task-specific coordination
* Modular communication (just like your Recipe Agent → Shopping Agent)

We can build your own simple MCP-like pattern using Python in a Streamlit session or file-based communication.

MCP-Like Flow Between Our Agents:

1.Recipe Agent:

* User enters dish name
* Fuzzy-matched recipe is found
* Store dish name or ingredient list in a shared context
* Shared Context (MCP layer):
* Could be st.session\_state, or a JSON/text file like context.json

2.Shopping Agent:

* Reads the shared context
* Auto-populates shopping list based on last selected recipe

1.session\_state: if you're using st.session\_state, then both agents must be part of the same Streamlit app (same .py)

st.session\_state Scope:

* Session-based: Data stored in st.session\_state exists only during that user session and within the same app process.
* Not shared across different apps or .py files running separately.
* If Recipe Agent is in recipe\_agent.py and Shopping Agent is in shopping\_agent.py (and run independently), then session\_state won’t carry over.

Alternatives if Agents Are in Different Apps:

1.Use a Shared File (like context.json)

* Let Recipe Agent write to a file, and Shopping Agent read from it.

2.Use a Local SQLite Database

* Store context (selected recipe/ingredients), and both agents can read/write.

3.Use a Tiny Flask API Backend

* A central server stores the shared context, and both agents call endpoints

**Let’s implement recipe & shopping agent using CMP (context.json):**

implement shared communication between the Recipe Agent and the Shopping Agent using a context.json file. This file will act like a simple shared memory to store the selected recipe.

context.json:

{

  "dish\_name": "",

  "ingredients": []

}

recipe\_agent.py:

import streamlit as st

import pandas as pd

import json

from fuzzywuzzy import process

# Load recipe data from CSV

df = pd.read\_csv("recipes.csv")

dish\_names = df['Dish Name'].tolist()

st.title("🍛 Recipe Agent")

# User input

user\_input = st.text\_input("Enter the dish name", key="dish\_input")

if st.button("Get Recipe") and user\_input:

    best\_match, score = process.extractOne(user\_input, dish\_names)

    if score >= 70:

        # Fetch recipe row

        recipe\_row = df[df["Dish Name"] == best\_match].iloc[0]

        # Display ingredients

        ingredients\_str = recipe\_row["Ingredients"]

        ingredients = [item.strip() for item in ingredients\_str.split(",")]

        st.subheader(f"Ingredients for {best\_match}:")

        for ing in ingredients:

            st.write(f"- {ing}")

        # Display instructions

        instructions = recipe\_row["Instructions"]

        st.subheader(" How to Make It:")

        st.write(instructions)

        # Save context for Shopping Agent

        with open("context.json", "w") as f:

            json.dump({

                "dish\_name": best\_match,

                "ingredients": ingredients

            }, f)

        st.success(" Recipe saved for Shopping Assistant!")

    else:

        st.error(" Recipe not found. Please try another dish name.")

shopping\_agent.py:

import streamlit as st

import json

st.title("🛍️ Shopping Agent")

try:

    with open("context.json", "r") as f:

        context = json.load(f)

except FileNotFoundError:

    context = {"dish\_name": "", "ingredients": []}

if context["dish\_name"]:

    st.write(f"### Suggested Ingredients for \*\*{context['dish\_name']}\*\*:")

    for item in context["ingredients"]:

        st.write(f"- {item}")

    if st.button("Order Ingredients"):

        st.success("🛒 Ingredients ordered successfully!")

else:

    st.warning("No recipe selected yet. Go to Recipe Agent first.")

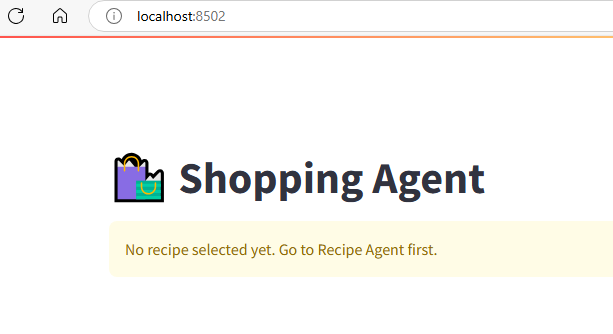
Run the Apps => In two terminals or tabs:

streamlit run recipe\_agent.py

streamlit run shopping\_agent.py

Output:

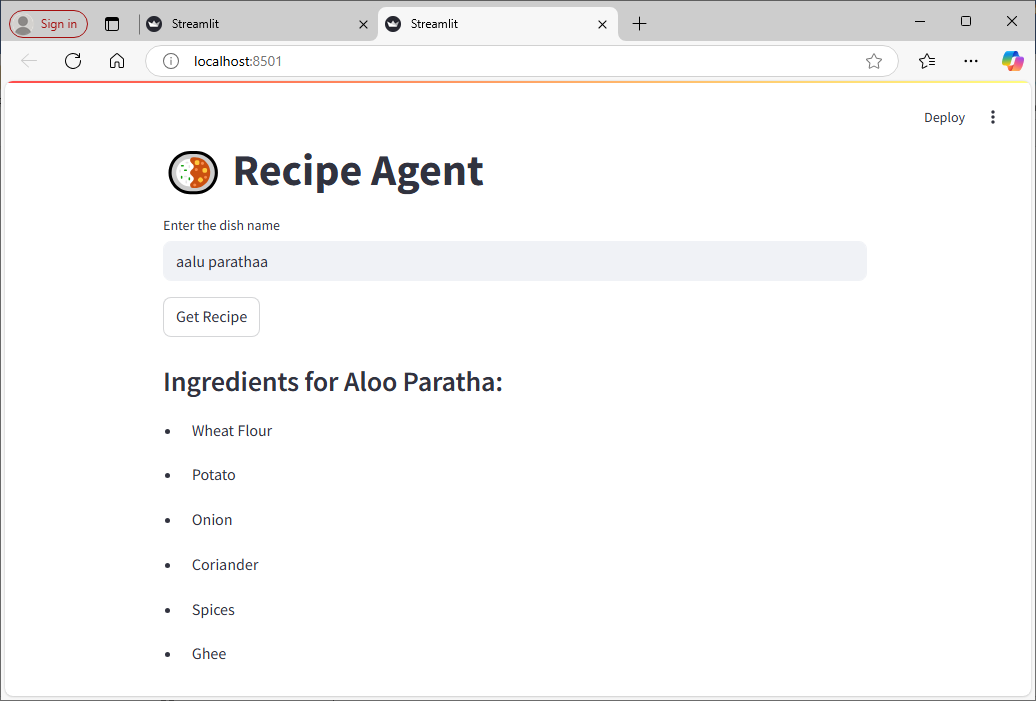
shooping\_agent



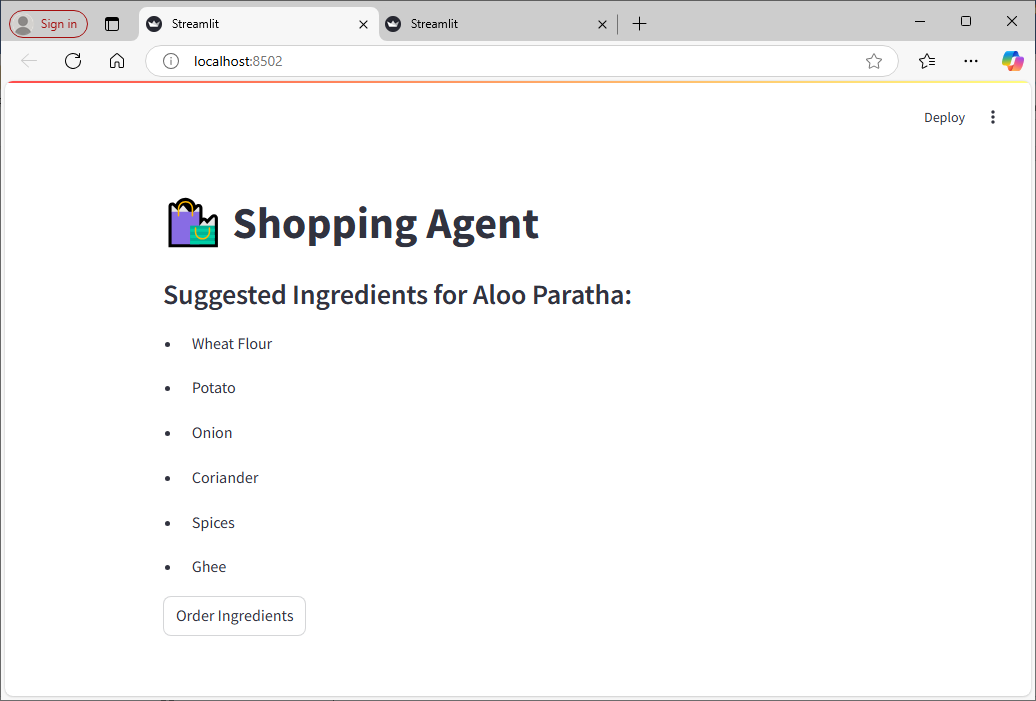
recipe\_agent.py



When user enter dish name:



Shopping\_agent get reflected:



And context.json file also get reflected:

{"dish\_name": "Aloo Paratha", "ingredients": ["Wheat Flour", "Potato", "Onion", "Coriander", "Spices", "Ghee"]}

What is CMP in this scenario?

CMP (Context Management Protocol) here is a method to store and share context (data) between multiple AI agents (apps) using a shared file – in this case, a simple context.json file.

Why This Works

* Memory Without Database: No need for a DB or cloud sync — it's just a local JSON file acting as memory.
* Loose Coupling: Agents are independent; they don’t need to run in the same Streamlit app or session.
* Lightweight Protocol: JSON is human-readable, portable, and fast.
* Multi-Agent Communication: Multiple AI agents can read/write this file — it's the "language" they speak through.