Step 0: Install prerequisites

1. **Install Python 3.9+**
   Check:
   `python --version`
2.
   It should say 3.9 or higher.
3. **Install Ollama (local LLM)**
   Go to: https://ollama.com
   Install it for your OS and download the `llama3` model.
   Test in terminal:
   `ollama run llama3`
4.
   If it answers, you are ready.
5. **Install VS Code** (if not already installed)
   https://code.visualstudio.com/

Step 1: Create project folder in VS Code

1. Open VS Code
2. Click **File → Open Folder → New Folder**
3. Name it: `local-ai-agent`
4. Click **Open**

You should now see an empty folder in the Explorer panel.

Step 2: Open terminal in VS Code

1. Press: `Ctrl + `` (backtick)
2. Terminal opens at the folder path
3. Optional: Increase terminal size by dragging

Step 3: Create Python virtual environment

1. In the terminal type:
   `python3 -m venv venv`
2.
3. This creates a folder `venv` for Python dependencies.
   Activate it:
   - **Mac/Linux**:
     `source venv/bin/activate`
   -
   - **Windows**:
     `venv\Scripts\activate`
   -
4. You should see `(venv)` in terminal.
5. Explaination:
   Virtual environments isolate project dependencies so nothing breaks other projects.

Step 4: Install dependencies

1. In terminal type:
   `pip install ollama`
2.
3. Explain:
   - `ollama` lets us talk to a local LLM
   - No API keys required
   - This is the **brain** of the agent

Step 5: Create project files

In VS Code Explorer, **right click → New File** for each:

1. `llm.py` → will contain the LLM interface
2. `agent.py` → will contain agent logic
3. `main.py` → will run the agent

Step 6: Write LLM interface

1. Open `llm.py`
2. Paste:

```
import ollama

def call_llm(prompt):
    """
    This function sends a prompt to the local LLM and returns
the output.
    """
    response = ollama.chat(
        model="llama3",
        messages=[
            {"role": "user", "content": prompt}
        ]
    )
    return response["message"]["content"]
```

3. **Explain line by line**:

- `import ollama` → allows talking to local LLM
- `call_llm(prompt)` → isolates the LLM, keeps agent code clean
- `response = ollama.chat(...)` → sends the prompt to the LLM
- Returns text, which is used by the agent

Step 7: Write the agent logic

1. Open `agent.py`

2. Paste:

```
from llm import call_llm

class AIAgent:
    def __init__(self, goal):
        """
        Initialize agent with a goal
        """
        self.goal = goal
        self.plan = []
        self.memory = []
        self.final_answer = None
```

Explaination:

- `goal` = what user wants
- `plan` = list of steps to achieve goal
- `memory` = keeps track of all work done
- `final_answer` = output at the end

3. Add **plan creation** method:

```
    def create_plan(self):
        """
        Ask the LLM to break the goal into steps
        """
        prompt = f"""
You are an AI planner.

Break this goal into simple steps.
Only list the steps, no explanations.

Goal:
{self.goal}
"""
        response = call_llm(prompt)
        # Split response into list of steps
```


```
        self.plan = [
            step.strip("- ").strip()
            for step in response.split("\n")
            if step.strip()
        ]
```

Explaination:

- `call_llm(prompt)` → uses LLM for reasoning
- LLM generates a **plan first** → planning before acting
- Splitting lines makes it iterable

4. Add **step execution**:

```
    def execute_step(self, step):
        """
        Execute one step using the LLM
        """
        prompt = f"""
You are an AI agent working on this goal:
{self.goal}

Current step:
{step}

Do the work for this step. Be short and concrete.
"""
        result = call_llm(prompt)
        self.memory.append(result)
        return result
```

Explaination:

- Agent handles one step at a time
- Stores result in memory → multi-step thinking
- Memory grows, important for stopping decisions

5. Add **stopping logic**:

```
    def is_finished(self):
```


```
        """
        Ask LLM if the goal is complete
        """
        prompt = f"""
Given the following work:

{self.memory}

Is the original goal fully completed? Answer yes or no.
"""
        answer = call_llm(prompt)
        return "yes" in answer.lower()
```

Explaination:

- Agent evaluates itself
- Stops autonomously
- This is **key agentic behavior**

6. Add **run method**:

```
    def run(self):
        """
        Full agent loop: plan → act → stop
        """
        print("\nCreating plan...")
        self.create_plan()

        for step in self.plan:
            print(f"\nExecuting: {step}")
            self.execute_step(step)

            if self.is_finished():
                break

        self.final_answer = "\n".join(self.memory)
        return self.final_answer
```

Explaination:

- Loops through plan
- Executes each step
- Checks if goal is complete
- Returns final result

Step 8: Run the agent

1. Open `main.py`
2. Paste:

```
from agent import AIAgent

goal = input("Enter a goal for the AI agent: ")

agent = AIAgent(goal)
result = agent.run()

print("\nFinal result:")
print(result)
```

Explaination:

- Takes **user input**
- Creates agent with goal
- Runs agent loop
- Prints final memory

Step 9: Test it

1. In VS Code terminal:

```
python main.py
```

2. Enter a goal, e.g.:

```
Create a 3-day Python study plan
```

3. Output should show:

- Plan creation
- Step execution
- Memory accumulation
- Final answer

Step 10: Teaching notes

1. LLM is **reasoning engine**, not the agent
2. Agent controls:

- Planning
- Step execution
- Memory
- Autonomy

3. Beginners see **how a multi-step AI agent works**
4. This code works **locally** → good foundation

✅ Next steps after this

1. Add **tool calls** (like reading files or APIs)
2. Add **file writing** → agent can create reports
3. Add **GUI or web UI** for interaction
4. Replace Ollama with **Vertex AI** for cloud version

5. Add **multi-agent orchestration**