Session – 4&5

Date: October 11, 2025

Detailed Course Notes: Agentic AI

Session 1: Intro & Agentic Frameworks

Initial Recap (N8N Flow)

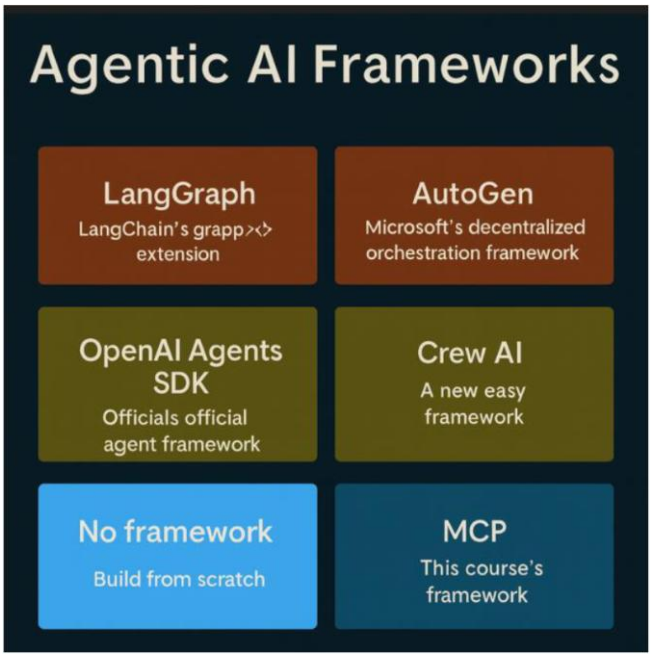
- We briefly looked at an N8N workflow diagram.
- It showed a flow: Gmail Trigger -> Code -> AI Agent -> Code1 -> Redis.
- The AI Agent used tools like OpenAI Chat Model, gmail label tool, and gmail draft tool. This was a good visual of what we're about to build manually.

Topic 0: What are our options? (Agentic AI Frameworks)

This is about *how* to build agents. There's a spectrum from easy/manual to complex/powerful.

- **Complexity 1: (The "Do It Yourself" way)**
    - **No Framework:** Just using raw API calls to LLMs (like OpenAI). This is what we did in **Lab 1 & 2**.
        - *Pro:* Full control.
        - *Con:* You have to build *everything* (state management, tool calling logic) yourself.
    - **MCP (Model Context Protocol):** From Anthropic. It's a standard for models to connect *to each other* without needing as much "glue code" from us.
- **Complexity 2: (The "Managed" way)**
    - **OpenAI Agents SDK:** The "official" toolkit from OpenAI for building agents.
    - **Crew AI:** A newer, "easy" framework. It's low-code and uses .yaml files for configuration. Seems good for simple agent teams.
- **Complexity 3: (The "Heavy-Duty" way)**
    - **LangGraph:** This is an extension of LangChain. It's graph-based, which means it's really good for complex, cyclical workflows where agents might need to loop or make decisions.

- **AutoGen:** Microsoft's big framework. It's for "decentralized multi-agent orchestration." This is for when you have many specialized agents all collaborating.



Lab Setup & Core Concepts

Step-by-Step: Getting the Labs Running

1. **Clone Repo:** git clone https://github.com/TEJAPS/agentic-notebooks.git
2. **Get IDE:** Download **Cursor** (https://cursor.com/). It's an AI-assisted VS Code fork.
3. **Install uv:** This is a new, super-fast Python package manager. We *must* use this.
    - Run this in PowerShell: powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"
4. **Create Environment:** cd into the repo folder and run:
    - uv sync
    - This reads the pyproject.toml file, creates an isolated virtual environment (.venv folder), and installs all the packages.
5. **Get API Key:** Go to https://platform.openai.com/settings/organization/api-keys and create a new secret key.
6. **Set Environment Variable:** Create a .env file in the project folder and add your key:
    - OPENAI_API_KEY=sk-proj-YOUR_KEY_HERE

Key Definitions

This is the most important part for understanding the course.

- **Agent:**

An **autonomous decision-maker**. It's a program where the LLM's output *controls the workflow*. It can **reason**, maintain **state** (memory, goals), and dynamically decide *what to do next* (e.g., call a tool, talk to another agent).

Example: A ResearchAgent that decides whether to use Google or search a database based on the query.

- **Tool:**

A **non-intelligent function** or API that the agent can *call* to act in the world. It doesn't reason; it just *does one specific thing*.

Example: GoogleSearchTool, PythonREPLTool, DatabaseTool, gmail_draft_tool.

- **Resource:**

External data or a persistent store. It's *not* executable logic; it's just information that an agent or tool can **read from** or **write to**.

Example: A Vector DB (Pinecone), a SQL database, or just a summary.txt file (like in Lab 3).

- **LLM Node (Non-Agent):**

A simple, "dumb" LLM call. It performs a **fixed subtask** (like summarizing or extracting keywords) as part of a predefined path. It has *no decision-making power*.

Workflow Patterns & Lab 1-2

5 Agentic Workflow Design Patterns

These are the "plays" we can run to build complex systems.

1. **Prompt Chaining:**
    - The simplest. Decompose a goal into fixed, sequential steps.
    - IN -> LLM1 -> Gate -> LLM2 -> OUT
2. **Routing:**
    - An "LLM Router" classifies the input and directs it to the right "expert" LLM or workflow.
    - IN -> LLM Router -> (Path A, Path B, Path C) -> OUT

3. **Parallelization:**
    - A "Coordinator" breaks a task into independent parts, runs them at the same time (concurrently), and an "Aggregator" merges the results.
    - IN -> Coordinator -> (LLM1, LLM2, LLM3) -> Aggregator -> OUT
4. **Orchestrator-Worker:**
    - A "master" Orchestrator agent *dynamically* assigns complex subtasks to "worker" LLMs and then synthesizes their outputs. This is more flexible than Parallelization.
    - IN -> Orchestrator -> (LLM1, LLM2, LLM3) -> Synthesizer -> OUT
5. **Evaluator-Optimizer (Used in Lab 3!)**
    - A quality-control loop. One LLM generates a solution, and a *second* LLM evaluates it. If it's rejected, it sends feedback to the generator to *rerun* and improve the answer.
    - IN -> LLM Generator --(Solution)--> LLM Evaluator --(Accepted)--> OUT
    - ^ |--(Rejected w/ Feedback)--|

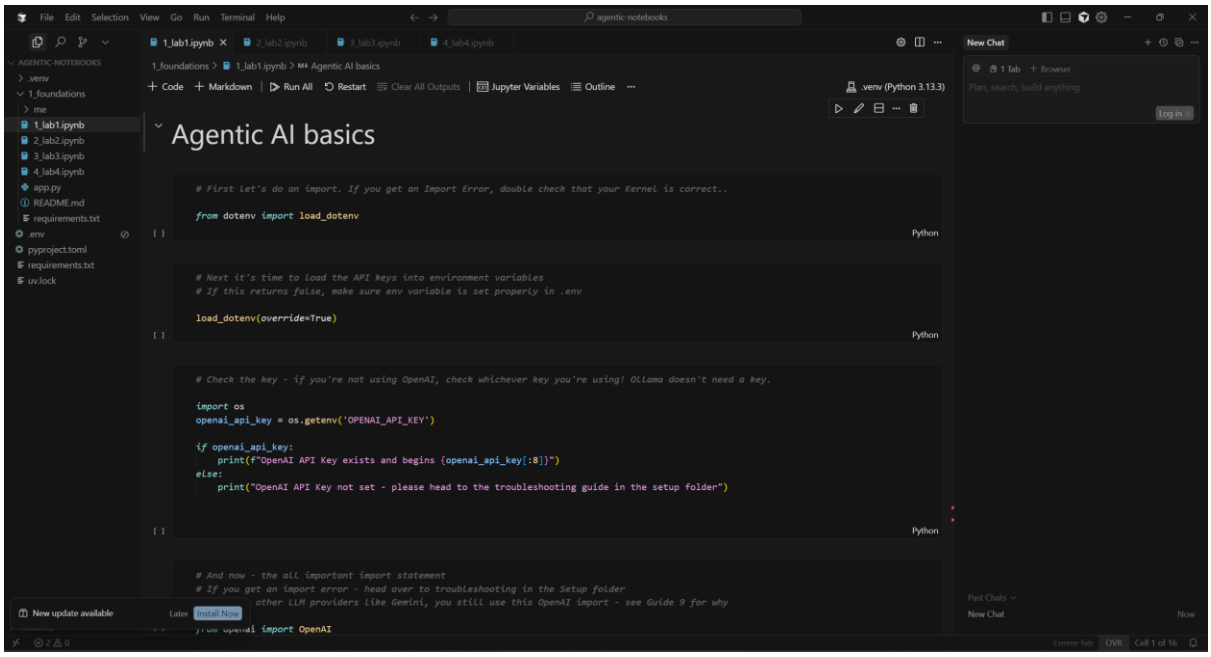
Lab 1 : "No Framework" Agent

- Build an agentic workflow using *only* raw OpenAI API calls.
- Showed us how to manually construct the messages array, make the API call (openai.chat.completions.create(...)), and parse the response. We built the "agent" logic ourselves.

Lab 2 : Multi-Model Agent

- Use different LLMs within the same workflow.
- **Models We Used:**
    1. **OpenAI:** gpt-4o-mini
    2. **Google:** gemini-2.0-flash
    3. **Ollama:** For running models *locally* (like llama3.1 or qwen).
- **Ollama Setup:**
    - Install from https://ollama.com/.
    - It runs locally at http://localhost:11434/.
    - *Need to restart Cursor/VS Code* for it to detect the ollama app.

- **Key Takeaway:** Showed how to abstract the LLM call so we can easily switch providers. We saw big differences in speed, cost, and response quality.



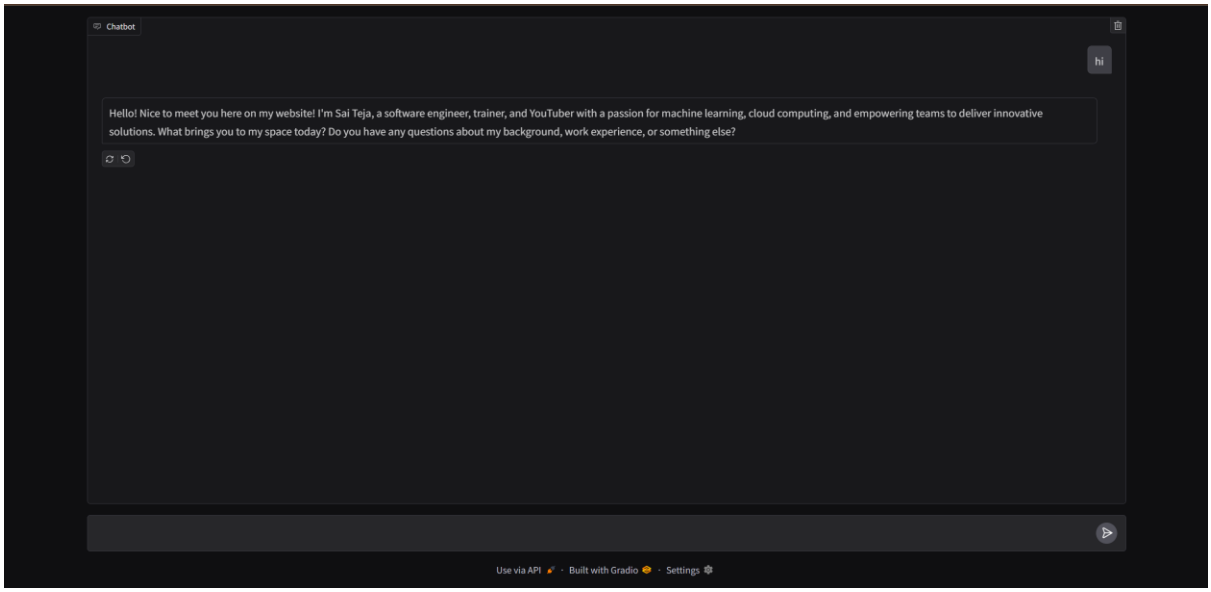
Resources, Tools, & Lab 3-4

Lab 3 - Resources & Evaluator Pattern

- Build a *personalized* chatbot using **Resources**.
- **Resources Used:**
    1. Our **LinkedIn profile** (downloaded as PDF and put in the /me folder).
    2. A summary.txt file we wrote about ourselves.
- The agent was prompted to read these files to get context before answering questions "as us."
- **Pattern Used: Evaluator-Optimizer**.
    - We created a Pydantic model for the evaluation:
    - # Create a Pydantic model for the Evaluation
    - from pydantic import BaseModel
    -
    - class Evaluation(BaseModel):
    - is_acceptable: bool
    - feedback: str

    - An evaluator LLM would check the generator's answer. If is_acceptable == False, the agent would re-run the request using the feedback.

- **UI:** We used **Gradio** to create a simple web chatbot interface.
    - gr.ChatInterface(chat, type="messages").launch()



Lab 4 - Tools & Notifications

- Give our agent **Tools** to perform actions.
- **Tools We Built:**
    1. A tool to **record unknown questions** (to capture queries the LLM couldn't answer).
    2. A tool to **record user details** (for context/personalization).
    3. A **Pushover** integration tool.
- **Pushover Setup (For Notifications):**
    1. Go to https://pushover.net/ and create an account.
    2. Copy your **User Key** -> save as PUSHOVER_USER in .env.
    3. Create a new "Application" -> get an **API Token** -> save as PUSHOVER_TOKEN in .env.
- **Key Takeaway:** This was a full end-to-end pipeline: **Personalization (Resources) + Action (Tools) + Notifications.**

Assignments

1. **Complete all 4 labs.**

2. **Personalized LLM Use Case (Team Project):**
    - Each team member must choose **one (1) tool** and **one (1) resource**.
    - Design and implement a *real-world solution* using them with your personalized LLM.
3. **Finish all previous pending assignments.**