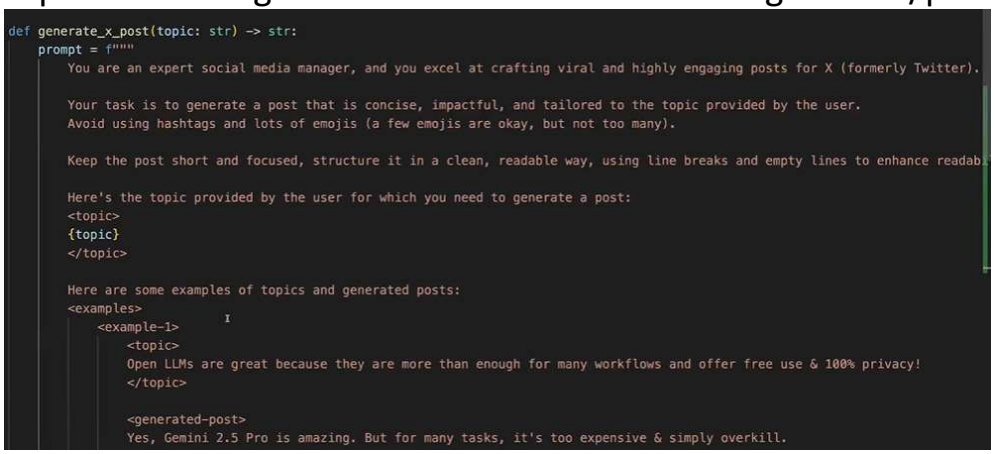
AI Workflow & Agents -2

22 May 2025 10:50

Till now we have taken prompt as the user_input itself but we can add the prompt and get the desired type of output from the models, even we can add the examples, topics to it and generated text from our end to get the o/p as expected.



```
def generate_x_post(topic: str) -> str:
    prompt = f"""
        You are an expert social media manager, and you excel at crafting viral and highly engaging posts for X (formerly Twitter).

        Your task is to generate a post that is concise, impactful, and tailored to the topic provided by the user.
        Avoid using hashtags and lots of emojis (a few emojis are okay, but not too many).

        Keep the post short and focused, structure it in a clean, readable way, using line breaks and empty lines to enhance readabi

        Here's the topic provided by the user for which you need to generate a post:
        <topic>
        {topic}
        </topic>

        Here are some examples of topics and generated posts:
        <examples>
            <example-1>
                <topic>
                Open LLMs are great because they are more than enough for many workflows and offer free use & 100% privacy!
                </topic>

                <generated-post>
                Yes, Gemini 2.5 Pro is amazing. But for many tasks, it's too expensive & simply overkill.
```

We can use multiple models for diff type of data in a single file.

```
print("----------")
print("Extracting core content from the website...")
core_content = extract_core_website_content(html_content)
print("Extracted core content:")
print(core_content)

print("----------")
print("Summarizing the core content...")
summary = summarize_content(core_content)
print("Generated summary:")
print(summary)
```

```
def extract_core_website_content(html: str) -> str:
            {html}
            </html>

            Please extract the core content and return it as plain text.
        """
    )

    return response.output_text


def summarize_content(content: str) -> str:
    response = client.responses.create(
        model="gpt-4o-mini",
        input=f"""
            You are an expert summarizer. Your task is to summarize the provided content into a concise and clear summary.

            Here is the content to summarize:
            <content>
            {content}
            </content>

            Please provide a brief summary of the main points in the content. Prefer bullet points and avoid unncessary explanatio
        """
    )

    return response.output_text
```

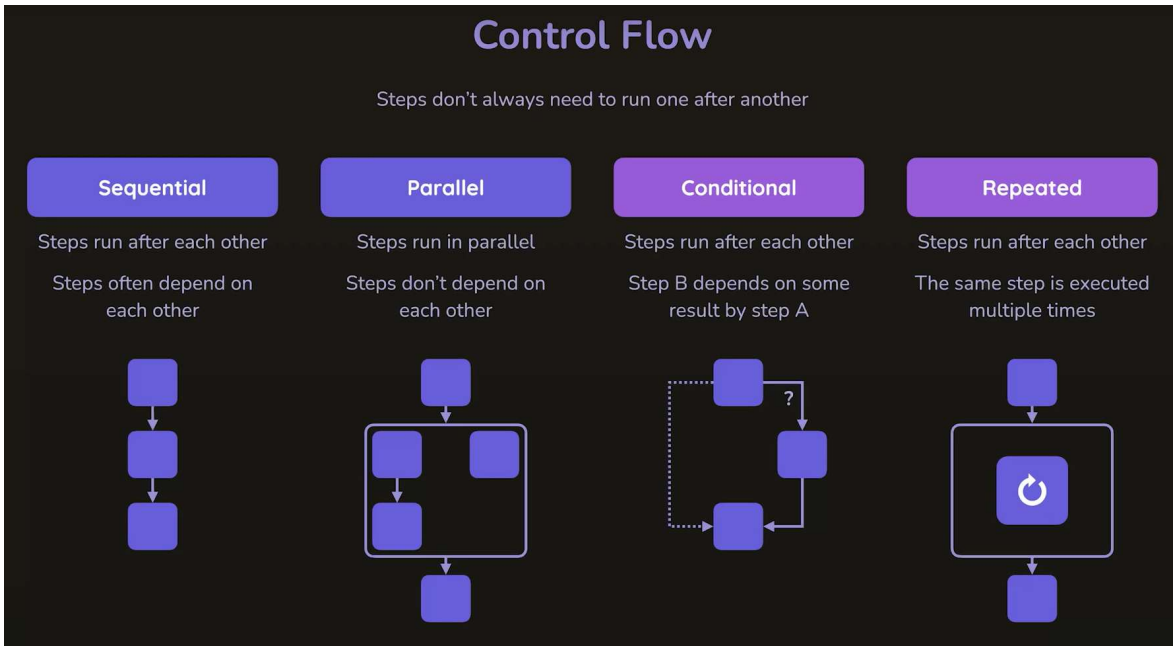
We can create images from the text based on the API. For image generation we can use the ChatGpt API dev.
But for using image generation we need to be verified by organization and Organization ID

```
def generate_thumbnail(article: str) -> bytes:
    print("Generating thumbnail...")

    response = client.images.generate(
        model="gpt-image-1",
        prompt=f"Generate a thumbnail for the following blog post: {article}",
        n=1,
        output_format="jpeg",
        size="1536x1024"
    )

    image_bytes = base64.b64decode(response.data[0].b64_json)
    return image_bytes
```

Control Flow
==========



Building AI Agents

==========

Core Components of an AI Agent

Here's what goes into building one:

Component	Role
LLM (Large Language Model)	The brain that reasons and generates responses
Memory System	Stores context and history across interactions
Tool Integration	Connects to APIs, databases, or external systems to take action
Planner/Controller	Decides what steps to take and in what order
Feedback Loop	Evaluates results and iterates for improvement

Design Patterns You Can Use

According to Anthropic and other experts2, successful agents often use these patterns:

- **Prompt Chaining**: Break tasks into sequential steps
- **Routing**: Direct inputs to specialized tools or models
- **Parallelization**: Run subtasks concurrently
- **Orchestrator-Worker**: One agent delegates to others
- **Evaluator-Optimizer**: Generate, evaluate, and refine outputs

Multi Agent System

==========

A Multi-Agent System (MAS) is a network of intelligent agents that interact within a shared environment to solve complex problems collaboratively or competitively. Each agent operates autonomously but contributes to a larger goal through coordination, communication, and sometimes negotiation.

```python
class Tool:
    """
    The base class for a tool that can be used by an agent.
    """

    def __init__(
        self,
        name: str,
        description: str,
        parameters: Dict[str, Any],
    ):
        self.name = name
        self.description = description
        self.parameters = parameters

    def get_schema(self) -> Dict[str, Any]:
```

```python
class ResearchPlannerAgent(Agent):
    """
    A research planner agent that uses the tools to plan a research project.
    """

    def __init__(self):
        super().__init__()
        self.register_tool(StoreResearchPlanTool())
        self.register_tool(GetResearchPlansTool())
        self.register_tool(DeleteResearchPlanTool())
        self._set_initial_prompt()

    def _set_initial_prompt(self):
        """
        Sets the initial prompt for the agent.
        """
        self.messages = [
            {
                "role": "developer",
                "content": """
                You are a research planner agent. You are tasked with helping the user plan a web research project.
                The user will provide you with a research task and your job is to create a research plan together with the user.
                Your job is NOT to answer the user's question. Instead, you MUST help them build a good research plan that can then be
                The research plan should include things like:
                    - Core topics to be researched
```

```python
class Agent:
    def register_tool(self, tool: Tool):
        """
        Registers a tool with the agent.
        """
        self.tools[tool.name] = tool

    def _get_tool_schemas(self) -> list[Dict[str, Any]]:
        """
        Returns the list of tool schemas.
        """
        return [tool.get_schema() for tool in self.tools.values()]

    def execute_tool_call(self, tool_call: Any) -> str:
        """
        Executes a tool call and returns the output.
        """
        fn_name = tool_call.name
        fn_args = json.loads(tool_call.arguments)

        if fn_name in self.tools:
            tool_to_call = self.tools[fn_name]
            try:
                print(f"Calling {fn_name} with arguments: {fn_args}")
                # The return value of the function is converted to a string to be compatible with the API.
                return str(tool_to_call.execute(tool_call.arguments))
            except Exception as e:
                return f"Error calling {fn_name}: {e}"

        return f"Unknown tool: {fn_name}"

    def run(self):
        """
        Runs the agent. This method should be implemented by subclasses.
```