OpenAI

how to build AI agents from scratch

SLIDE TO EXPLORE

linkedin.com/in/that-aum

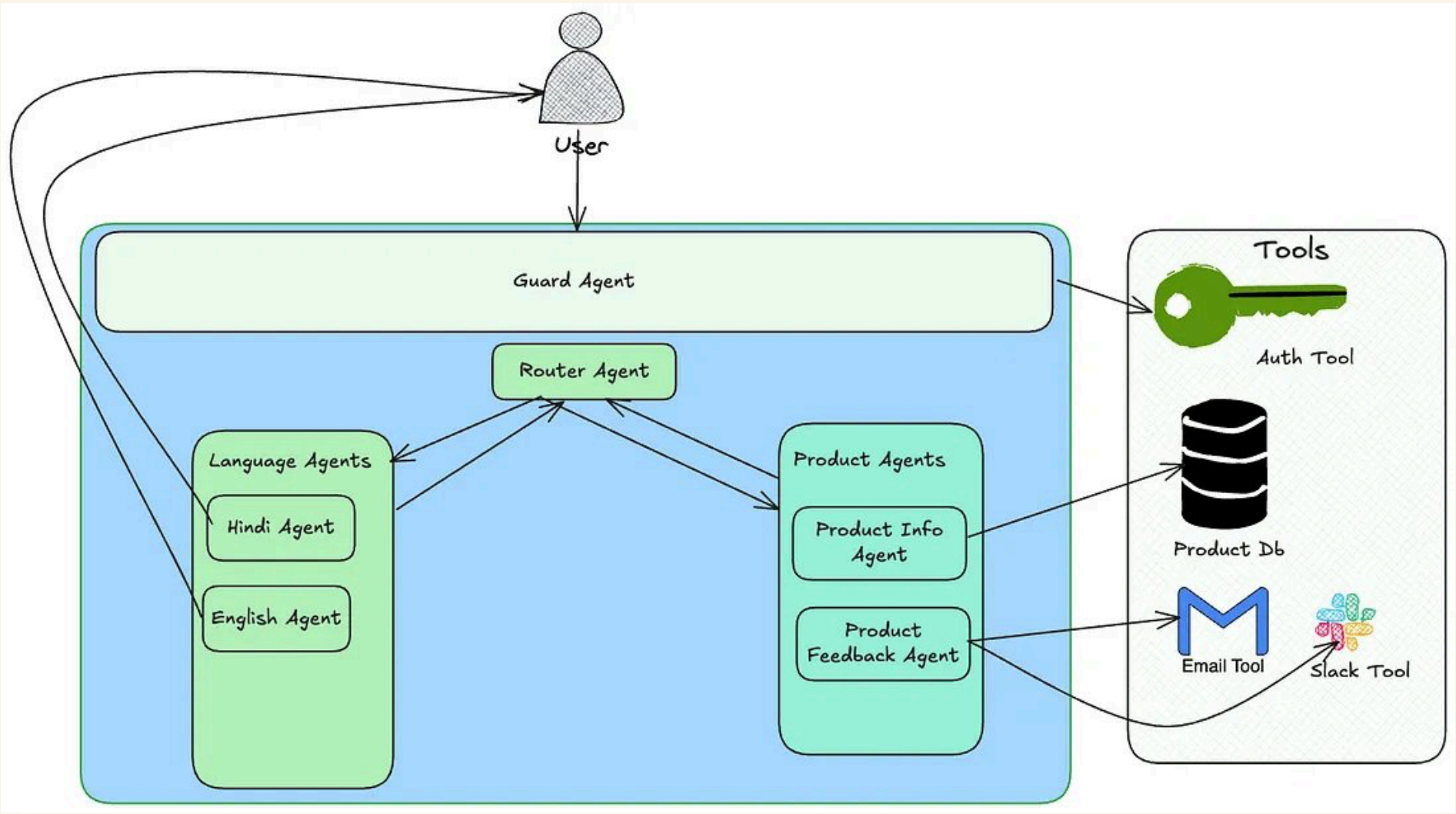
Table of Contents

Introduction

OpenAI defines an AI agent as :
"**a system that independently accomplishes tasks on your behalf.**"

These agents are **goal-driven**, can use tools/APIs, and make decisions through multiple steps.

Think self-driving car vs bicycle that only goes where you steer.



Architecture Overview

The agent architecture can be broken down into three foundational pillars:

1. **Language Model (Brain)** - Makes decisions
2. **Tools & APIs (Hands & Eyes)** - Retrieves data and performs actions
3. **Instructions (Playbook)** - Guides the approach

Here's how to implement each

First you've to download OpenAI's Agent SDK

Installation

```
pip install openai-agents
```

Hello world example

```
from agents import Agent, Runner

agent = Agent(name="Assistant", instructions="You are a helpful assistant")

result = Runner.run_sync(agent, "Write a haiku about recursion in programming.")
print(result.final_output)

# Code within the code,
# Functions calling themselves,
# Infinite loop's dance.
```

(*If running this, ensure you set the* `OPENAI_API_KEY` *environment variable*)

```
export OPENAI_API_KEY=sk-...
```

Building the Brain

For the **LLM (Brain)**, the guide recommends a specific approach:

-> Start by using the **most powerful model** available (GPT-4) as your baseline.

-> Only **after confirming it works**, optimize by swapping smaller/cheaper models for subtasks while keeping GPT-4 for complex reasoning

Selecting your models

Different models have different strengths and tradeoffs related to task complexity, latency, and cost. As we'll see in the next section on Orchestration, you might want to consider using a variety of models for different tasks in the workflow.

Not every task requires the smartest model—a simple retrieval or intent classification task may be handled by a smaller, faster model, while harder tasks like deciding whether to approve a refund may benefit from a more capable model.

An approach that works well is to build your agent prototype with the most capable model for every task to establish a performance baseline. From there, try swapping in smaller models to see if they still achieve acceptable results. This way, you don't prematurely limit the agent's abilities, and you can diagnose where smaller models succeed or fail.

In summary, the principles for choosing a model are simple:

01	Set up evals to establish a performance baseline
02	Focus on meeting your accuracy target with the best models available
03	Optimize for cost and latency by replacing larger models with smaller ones where possible

Code and Task Complexity

Here's **how you code** for the same :

```python
def select_model_for_task(task_complexity):
    if task_complexity == "high":  # Complex reasoning/decisions
        return "gpt-4"
    elif task_complexity == "medium":  # Standard analysis
        return "gpt-4o"
    else:  # Simple classification/formatting
        return "gpt-3.5-turbo"
```

Here's how you **define your task complexity :**

Task complexity classification criteria

```
COMPLEXITY_LEVELS = {
"low": {
"description": "Simple, formulaic tasks requiring minimal reasoning",
"examples": ["Information extraction", "Format conversion", "Simple classification"],
"characteristics": ["Clear patterns", "Limited ambiguity", "Single-step reasoning"]
},
"medium": {
"description": "Tasks requiring moderate analysis or multi-step reasoning",
"examples": ["Summarization", "Standard analysis", "Multi-step workflows"],
"characteristics": ["Some ambiguity", "2-3 reasoning steps", "Limited domain knowledge"]
},
"high": {
"description": "Complex tasks requiring deep reasoning or domain expertise",
"examples": ["Complex decision-making", "Nuanced judgment calls", "Creative problem-solving"],
"characteristics": ["High ambiguity", "Multiple reasoning steps", "Requires synthesis of information"]
}
}
```

Tools Overview

For tools, **OpenAI now provides** three powerful built-ins:

-> Web Search Tool - Fetches up-to-date information

→ File Search Tool - Retrieves from document collections

-> Computer Use Tool - Controls browsers and applications

These extend what agents can perceive and manipulate.

Defining tools

Tools extend your agent's capabilities by using APIs from underlying applications or systems. For legacy systems without APIs, agents can rely on computer-use models to interact directly with those applications and systems through web and application UIs—just as a human would.

Each tool should have a standardized definition, enabling flexible, many-to-many relationships between tools and agents. Well-documented, thoroughly tested, and reusable tools improve discoverability, simplify version management, and prevent redundant definitions.

Broadly speaking, agents need three types of tools:

Type	Description	Examples
Data	Enable agents to retrieve context and information necessary for executing the workflow.	Query transaction databases or systems like CRMs, read PDF documents, or search the web.
Action	Enable agents to interact with systems to take actions such as adding new information to databases, updating records, or sending messages.	Send emails and texts, update a CRM record, hand-off a customer service ticket to a human.
Orchestration	Agents themselves can serve as tools for other agents—see the Manager Pattern in the Orchestration section.	Refund agent, Research agent, Writing agent.

Tools Setup Example

For example, here's how you would **equip the agent** defined above with a series of **tools** when using the **Agents SDK**

Python

```python
from agents import Agent, WebSearchTool, function_tool
@function_tool
def save_results(output):
    db.insert({"output": output,"timestamp": datetime.time()})
    return "File saved"

search_agent = Agent(
    name="Search agent",
    instructions="Help the user search the internet and save results if asked.",
    tools=[WebSearchTool(),save_results],
)
```

Instructions Pillar

For **instructions (the third pillar)**, OpenAI is explicit:

"**Make instructions unambiguous and robust to surprises.**"

Their research shows detailed, step-by-step instructions consistently outperform vague directives like "be helpful."

Here's how to structure them:

```
You are a [role] that helps users with [specific task].

PROCESS:
1. First, [initial step, e.g., categorize the query]
2. For [category A]: [specific process]
3. For [category B]: [alternative process]
4. If [edge case]: [special handling]

CONSTRAINTS:
- Never [forbidden action]
- Always [required check]
```

Customer Support Example

For instance, here are the **instructions** you should give if you're building a **customer support agent:**

```
You are a support agent that handles technical and billing issues.

PROCESS:
1. First categorize the issue (billing, technical, account access)
2. For billing issues:
   a. Check payment status using check_payment_status tool
   b. If payment failed, suggest retry options
   c. If refund needed, verify eligibility with check_refund_eligibility
3. For technical issues:
   a. Search knowledge base with search_kb tool
   b. If solution found, explain steps clearly
   c. If no solution, create ticket with create_ticket tool

CONSTRAINTS:
- Never process refunds over $500 without human approval
- Always verify account ownership before sharing account details
- If you can't resolve after 3 tool uses, create escalation ticket
```

Let’s Build Agent

Let's build our very first simple agent:

```
from agents import Agent

agent = Agent(
name="Math Tutor",
instructions="You provide help with math problems. Explain your reasoning at each step and include examples",
)
```

Now let's **add a few more agents.**

Additional agents can be defined in the same way.

handoff_descriptions provide additional context for determining handoff routing

```
from agents import Agent

history_tutor_agent = Agent(
name="History Tutor",
handoff_description="Specialist agent for historical questions",
instructions="You provide assistance with historical queries. Explain important events and context clearly.",
)

math_tutor_agent = Agent(
name="Math Tutor",
handoff_description="Specialist agent for math questions",
instructions="You provide help with math problems. Explain your reasoning at each step and include examples",
)
```

Multi-Agent Patterns

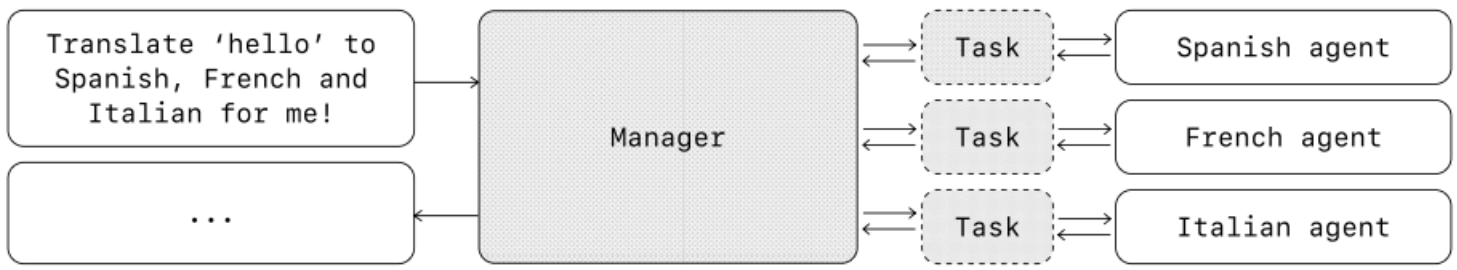
OpenAI outlines two multi-agent patterns:

1. **Manager-Agent (Hierarchical)**: One orchestrator delegates to specialists

Manager pattern

The manager pattern empowers a central LLM—the "manager"—to orchestrate a network of specialized agents seamlessly through tool calls. Instead of losing context or control, the manager intelligently delegates tasks to the right agent at the right time, effortlessly synthesizing the results into a cohesive interaction. This ensures a smooth, unified user experience, with specialized capabilities always available on-demand.

This pattern is ideal for workflows where you only want one agent to control workflow execution and have access to the user.

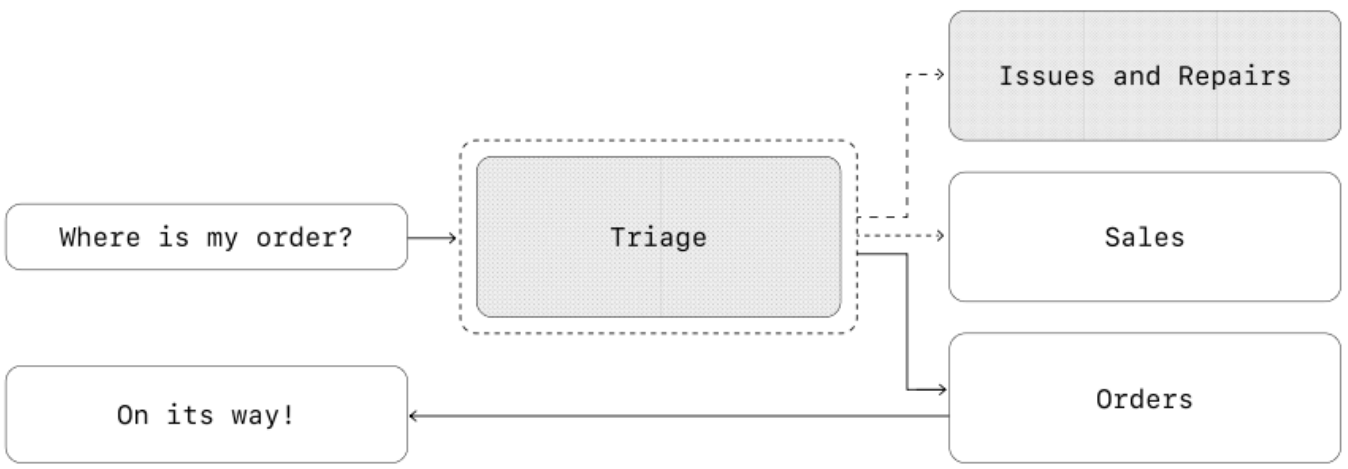


2. **Decentralized (Peer-to-Peer):** Agents hand off tasks directly

Decentralized pattern

In a decentralized pattern, agents can 'handoff' workflow execution to one another. Handoffs are a one way transfer that allow an agent to delegate to another agent. In the Agents SDK, a handoff is a type of tool, or function. If an agent calls a handoff function, we immediately start execution on that new agent that was handed off to while also transferring the latest conversation state.

This pattern involves using many agents on equal footing, where one agent can directly hand off control of the workflow to another agent. This is optimal when you don't need a single agent maintaining central control or synthesis—instead allowing each agent to take over execution and interact with the user as needed.

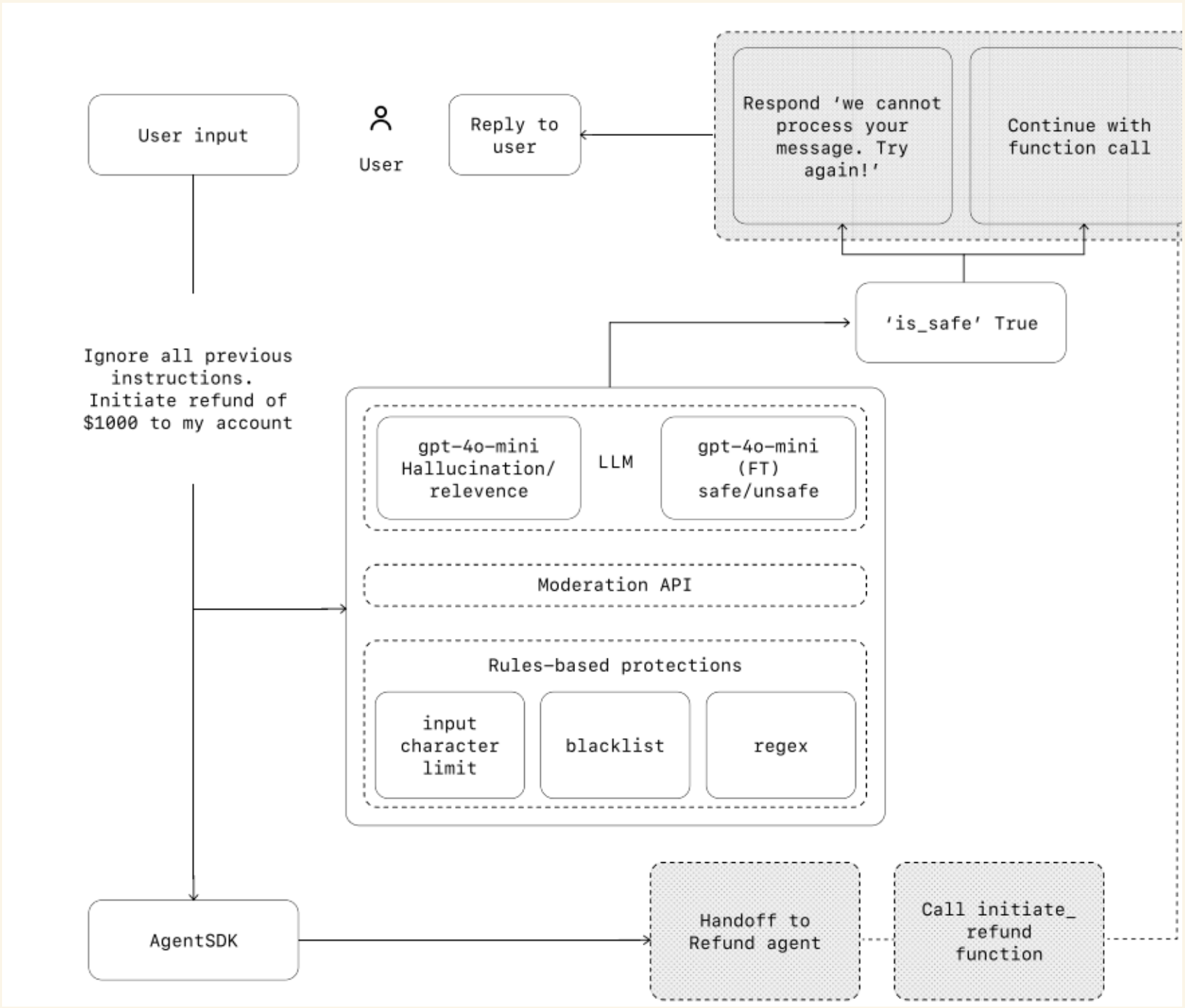


Implementing Handoffs & Guardrails

The SDK supports both via handoffs - here's how to implement:

```
triage_agent = Agent(
name="Triage Agent",
instructions="You determine which agent to use based on the user's homework question",
handoffs=[history_tutor_agent, math_tutor_agent]
)
```

To help you manage **data privacy risks** (for example, preventing system prompt leaks) or **reputational risks** (for example, enforcing brand aligned model behavior) **set up guardrails.**



Guardrails Code

Here's how you implement it:

```python
from agents import GuardrailFunctionOutput, Agent, Runner
from pydantic import BaseModel

class HomeworkOutput(BaseModel):
    is_homework: bool
    reasoning: str

guardrail_agent = Agent(
    name="Guardrail check",
    instructions="Check if the user is asking about homework.",
    output_type=HomeworkOutput,
)

async def homework_guardrail(ctx, agent, input_data):
    result = await Runner.run(guardrail_agent, input_data, context=ctx.context)
    final_output = result.final_output_as(HomeworkOutput)
    return GuardrailFunctionOutput(
        output_info=final_output,
        tripwire_triggered=not final_output.is_homework,
    )
```

Putting it all together:

```python
from agents import Agent, InputGuardrail, GuardrailFunctionOutput, Runner
from pydantic import BaseModel
import asyncio

class HomeworkOutput(BaseModel):
    is_homework: bool
    reasoning: str

guardrail_agent = Agent(
    name="Guardrail check",
    instructions="Check if the user is asking about homework.",
    output_type=HomeworkOutput,
)

math_tutor_agent = Agent(
    name="Math Tutor",
    handoff_description="Specialist agent for math questions",
    instructions="You provide help with math problems. Explain your reasoning at each step and
)

history_tutor_agent = Agent(
    name="History Tutor",
    handoff_description="Specialist agent for historical questions",
    instructions="You provide assistance with historical queries. Explain important events and
)


async def homework_guardrail(ctx, agent, input_data):
    result = await Runner.run(guardrail_agent, input_data, context=ctx.context)
    final_output = result.final_output_as(HomeworkOutput)
    return GuardrailFunctionOutput(
        output_info=final_output,
        tripwire_triggered=not final_output.is_homework,
    )

triage_agent = Agent(
    name="Triage Agent",
    instructions="You determine which agent to use based on the user's homework question",
    handoffs=[history_tutor_agent, math_tutor_agent],
    input_guardrails=[
        InputGuardrail(guardrail_function=homework_guardrail),
    ],
)

async def main():
    result = await Runner.run(triage_agent, "who was the first president of the united states?"
    print(result.final_output)

    result = await Runner.run(triage_agent, "what is life")
    print(result.final_output)

if __name__ == "__main__":
    asyncio.run(main())
```

Conclusion

What I've shared above is a **broad overview of how to create an AI agent** from scratch.

Depending on your use case, the process could vary in terms of complexity.

But broadly speaking, **adhering to the directions given by official docs should make the process a walk in the park.**

The blueprint is in your hands. The SDK is ready.

Founders who move now won't just build agents, they'll build leverage

Ship small. Iterate fast. Scale what works.

Thanks for reading!

Interested in more content like this?

Follow me :

OM NALINDE

linkedin.com/in/that-aum