Blazing the AI Trail:

Using LangGraph to Conquer the Oregon Trail

_robert
_applied ai
engineer

What is an “agent”?

- At its core, an agent is a process loop wherein a LLM decides the next step to take.
- In this way, agents are best understood as a architectural software pattern rather than a mysterious 3rd party.
- In practice, agents are to this point typically implemented as semi-directed graphs of process nodes.

Why does it matter?

The combination of LLMs to parse diverse inputs and the general availability of software tools makes a whole set of previously impossible workflows possible.

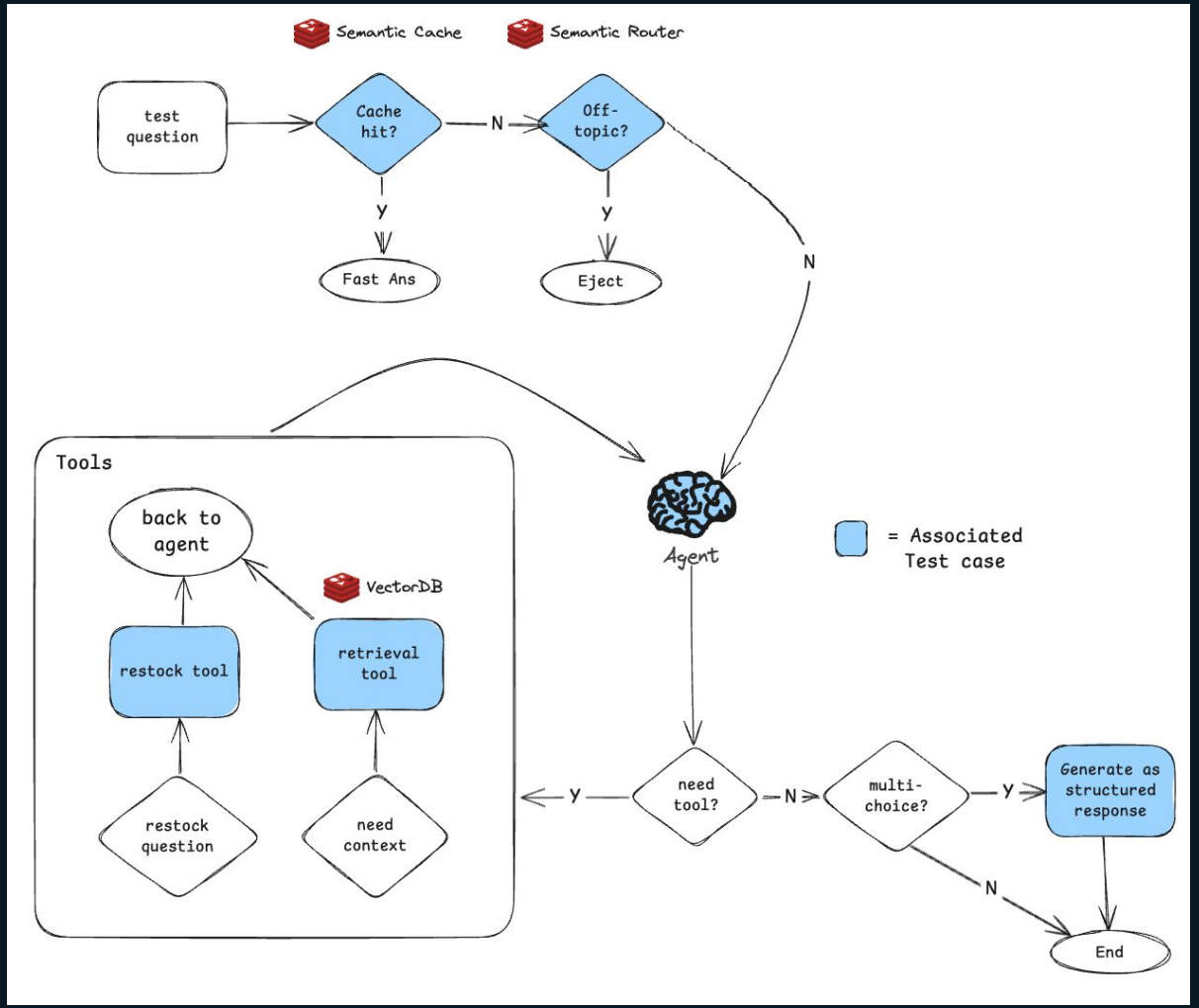
…which is why people won't stop talking about them.

Our goal

- Build an agent application that illustrates all the techniques needed for a real world solution.
- Understand how to structure LangGraph code outside of a notebook.
- Add pragmatic architectural components to optimize performance and usage.
- Develop the right tools for our agent to solve its intended problem.

The architecture.

- Agentic graph
  - Custom tool calling
  - Structured response with Pydantic
  - RAG integration
- Semantic cache
- Allow/block router



Let’s start coding

Step 1 clone the repo:
https://github.com/redis-developer/oregon-trail-agent-workshop

Pre-reqs:

- Docker
- Python (3.12.8)
- OpenAI api key

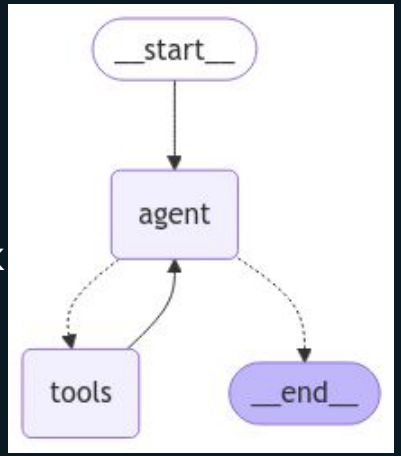
Optional (helpful):

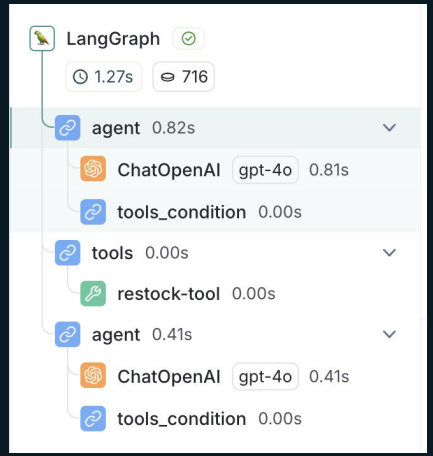
- LangSmith
- LangGraph Studio

LangGraph basics

We will be using LangGraph to build our agent. The code we will write will help us build graphs like this:

A minimal architecture might look lik this:

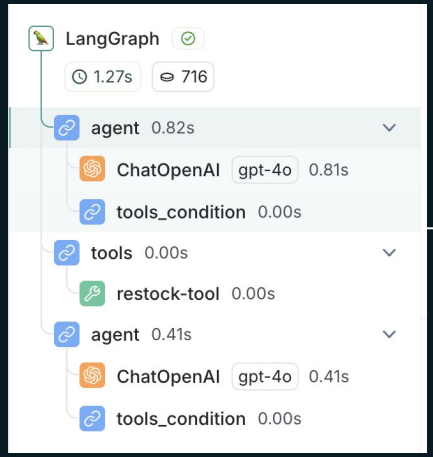


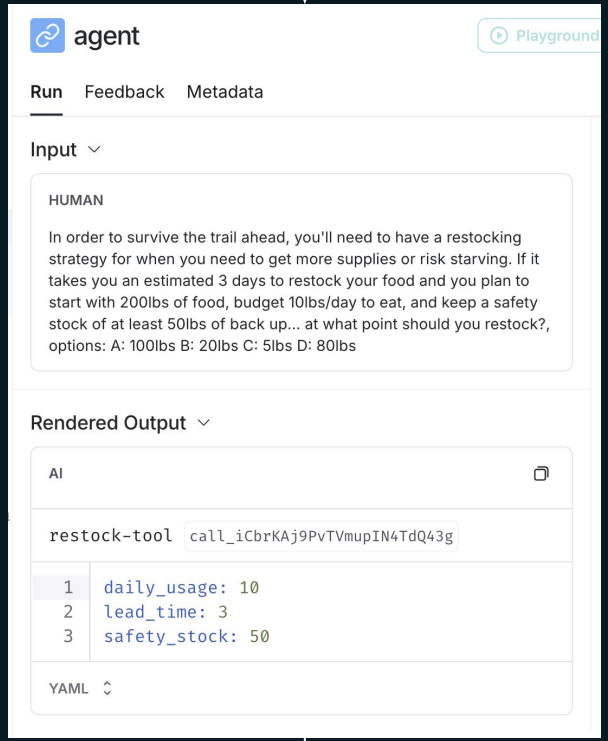


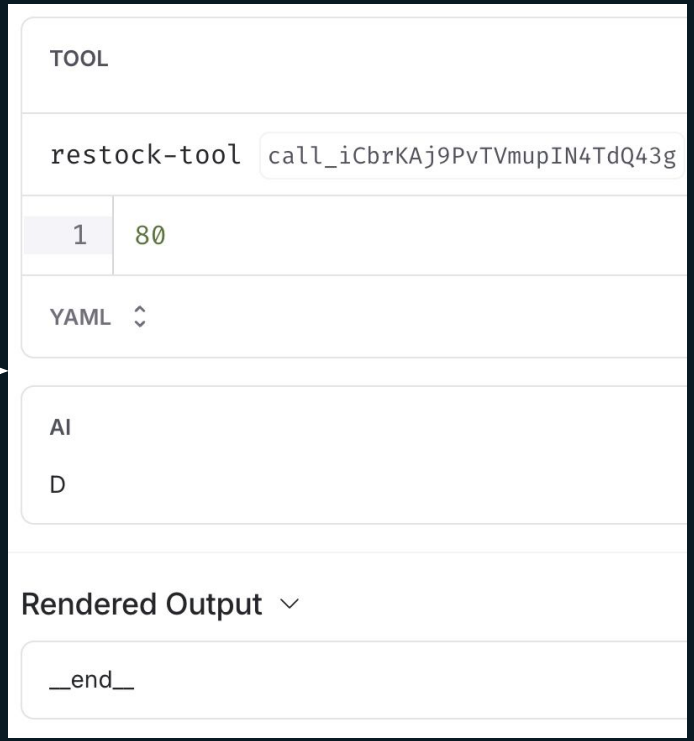
In execution:

- enter flow
- agent decides on tool condition
- use restock-tool
- Back to agent with answer -> end

Concrete visual of execution

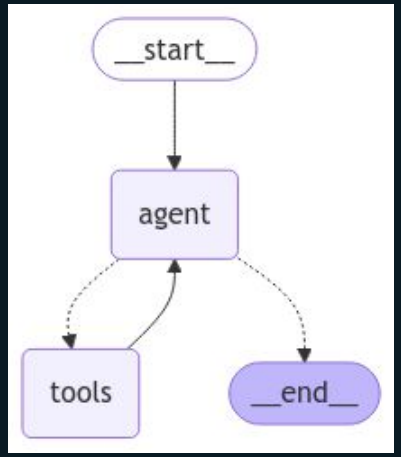






What does a graph look like in code?

...not too bad, right?




```python
llm = ChatOpenAI(model="gpt-4o")
llm_with_tools = llm.bind_tools(tools)

def agent(state: MessagesState):
    print(state["messages"])
    if state['messages'][-1].name == "retrieve_blog_posts":
        return {
            "messages": [
                llm_with_tools.invoke(
                    [SYS_MSG] +
                    [f"{state['messages'][0].content} \
                     consider context: {state['messages'][-1].content}"]
                    )
                ]
            }
    else:
        return {"messages": [llm_with_tools.invoke([SYS_MSG] + state['messages'])]}

builder = StateGraph(MessagesState)

# Add nodes
builder.add_node("agent", agent)
builder.add_node("tools", ToolNode(tools))  # for the tools

# Add edges
builder.add_edge(START, "agent")
builder.add_conditional_edges(
    "agent",
    # If the latest message (result)
    # from node agent is a tool call -> tools_condition routes to tools
    # If the latest message (result)
    # from node agent is a not a tool call -> tools_condition routes to END
    tools_condition,
)
builder.add_edge("tools", "agent")
graph = builder.compile()
```

LangGraph fundamentals

The graph we will compose consists of a few fundamental units:

- Agent (aka reasoner aka LLM)
  - An LLM model that decides based on the human input whether to use tools or not.
- Tools
  - A tool is a developer defined function that the Agent is aware of.
- Node
  - A node defines the execution loop within our graph.
  - We will create an agent node and a tools node.
  - The node syntax is how we define the relationship between our agent, tools, and other aspects of our graph.
- State
  - Set of messages passed between nodes to preserve context

A note on probabilistic systems

In the LLM age, there has been a shift from building largely deterministic systems to probabilistic systems. Agentic systems due to their probabilistic nature can be extremely powerful however within this new development paradigm come different flavors of problems.

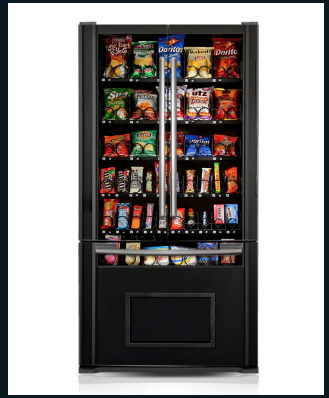
Deterministic

Pros:

- Direct relationship between Input (A20) and output (Doritos)
- Relatively cheap, fast, easy to maintain

Cons:

- Brittle to unexpected input
- Limit UI flexibility



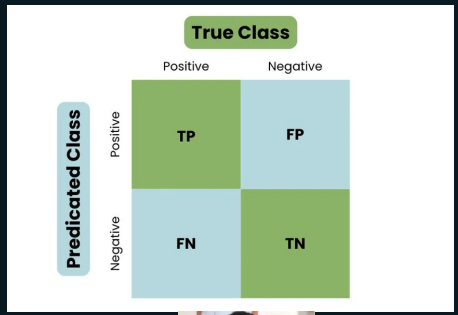
Probabilistic

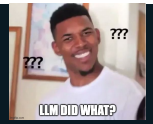
Pros:

- Can handle robust inputs and respond accordingly
- “Intelligence” reduces complex logic

Cons:

- Expected output not guaranteed
- Relatively expensive and slow





Confusion matrix?

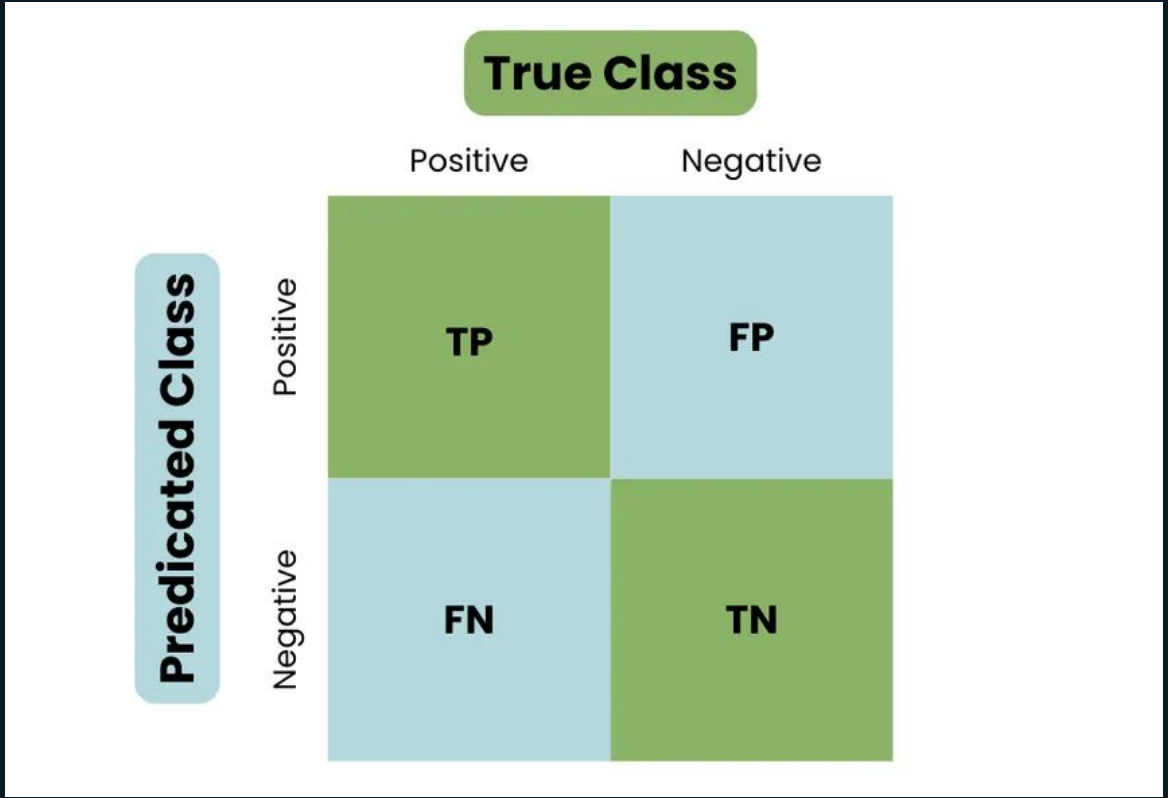
A confusion matrix maps the potential outcomes of a prediction.

TP = True positive
FP = False positive

FN = False negative
TN = True negative

Positive: prediction aligns with label.

Negative: prediction doesn't align with label.



What the heck does this have to do with the Oregon Trail?!

Many application in practice are like the classic Oregon Trail game. In that you need to:

- Use tools when necessary
- Gather additional relevant information
- Format and respond to inquiries appropriately

General workshop flow

- We will be working to pass all 5 test scenarios by updating code in the partcipant_agent/ folder.
- **All** steps are in the README.md so don't worry if you get ahead or fall behind.
- We will go step by step as a group through the various stage and try to keep to the median pace of the group
- If you get **extremely lost** there is a completed example in the example_agent/ folder.

And so it begins...

Code time 💻

Reference

Backup notebook

Notebook in redis-ai-resources

Pre-reqs:

- Docker
- Python (3.12.8)
- OpenAI api key

Optional (helpful):

- LangSmith
- LangGraph Studio

Setup

1. Create .env file and update OPENAI_API_KEY: `cp dot.env .env`
2. Create python virtual environment and `pip install -r requirements.txt`
3. Run redis-stack instance: `docker run -d --name redis -p 6379:6379 -p 8001:8001 redis/redis-stack:latest`
4. Test setup worked: `python test_setup.py`

🕰️ pause for people to try

Scenario 1: Name of the Wagon Leader

Question: What is the first name of the wagon leader?

Answer: Art

Goal:

- Finalize all LangGraph boilerplate
- Update system prompt
- Run test script for the first time
- Reminder: if you get lost all commands in the README.md

 pause

Review agent graph setup

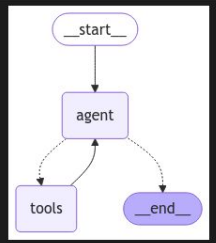
- Now we will test the setup open `workshop/participant_test.ipynb` or use LangGraph studio and confirm.
- If this works you've defined the core of your graph!

```
from participant_agent.graph import graph

from IPython.display import Image, display

display(Image(graph.get_graph(xray=True).draw_mermaid_png()))
```

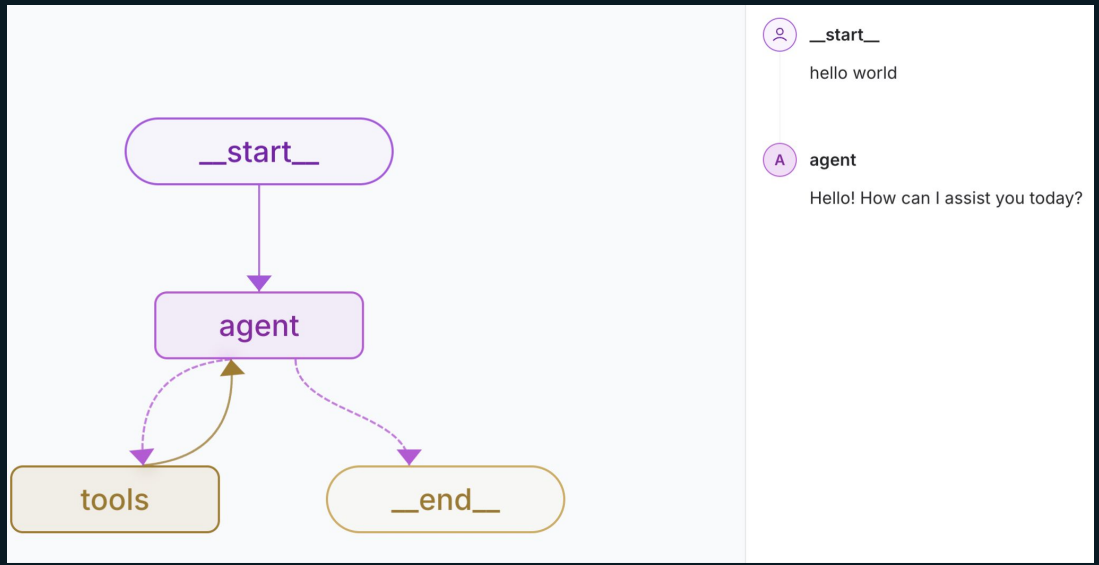
✓ 1.3s




```
graph.invoke({"messages": ["hello world"]})
```

✓ 1.4s

```
{'messages': [HumanMessage(content='hello world', additional_kwar
  AIMessage(content='Hello! How can I assist you today?', additio
```



Scenario 1 Demo 

Expected

```python
# Define a new graph
workflow = StateGraph(AgentState, config_schema=GraphConfig)

# Define the two nodes we will cycle between
workflow.add_node("agent", call_tool_model)
# workflow.add_node("respond", respond)
workflow.add_node("tools", tool_node)

# Set the entrypoint as `agent`
# This means that this node is the first one called
workflow.set_entry_point("agent")

# We now add a conditional edge
workflow.add_conditional_edges(
    "agent",
    tools_condition,
)

# We now add a normal edge from `tools` to `agent`.
# This means that after `tools` is called, `agent` node is called next.
workflow.add_edge("tools", "agent")
# workflow.add_edge("respond", END)

# Finally, we compile it!
# This compiles it into a LangChain Runnable,
# meaning you can use it as you would any other runnable
graph = workflow.compile()
```

Reminder: goal pass all tests

```
collected 5 items

test_example_oregon_trail.py::test_1_wagon_leader PASSED                  [ 20%]
test_example_oregon_trail.py::test_2_restocking_tool PASSED               [ 40%]
test_example_oregon_trail.py::test_3_retrieval_tool PASSED                [ 60%]
test_example_oregon_trail.py::test_4_semantic_cache PASSED                [ 80%]
test_example_oregon_trail.py::test_5_router PASSED                        [100%]

=================================== PASSES ===================================
_____________________________ test_1_wagon_leader _____________________________
----------------------------- Captured stdout call -----------------------------

 What is the first name of the wagon leader?


 response: Artificial
____________________________ test_2_restocking_tool ____________________________
----------------------------- Captured stdout call -----------------------------

 question: In order to survive the trail ahead, you'll need to have a restocking strategy for when you need to get more supplies or risk starving. If it takes you an estimated 3 days to restock your food and you plan to start with 2
00lbs of food, budget 10lbs/day to eat, and keep a safety stock of at least 50lbs of back up... at what point should you restock?


 Using restock tool!: daily_usage=10, lead_time=3, safety_stock=50


 response: D
____________________________ test_3_retrieval_tool ____________________________
----------------------------- Captured stdout call -----------------------------

 You’ve encountered a dense forest near the Blue Mountains, and your party is unsure how to proceed. There is a fork in the road, and you must choose a path. Which way will you go?


 response: B
____________________________ test_4_semantic_cache ____________________________
----------------------------- Captured stdout call -----------------------------

 There's a deer. You're hungry. You know what you have to do...


 response: bang
________________________________ test_5_router ________________________________
----------------------------- Captured stdout call -----------------------------

 Tell me about the S&P 500?

you shall not pass
```

Scenario 1: Defining a more advanced tool

Question: In order to survive the trail ahead, you'll need to have a restocking strategy for when you need to get more supplies or risk starving. If it takes you an estimated 3 days to restock your food and you plan to start with 200lbs of food, budget 10lbs/day to eat, and keep a safety stock of at least 50lbs of back up... at what point should you restock?

Answer: "D" (80lbs)

Scenario 2: steps to complete

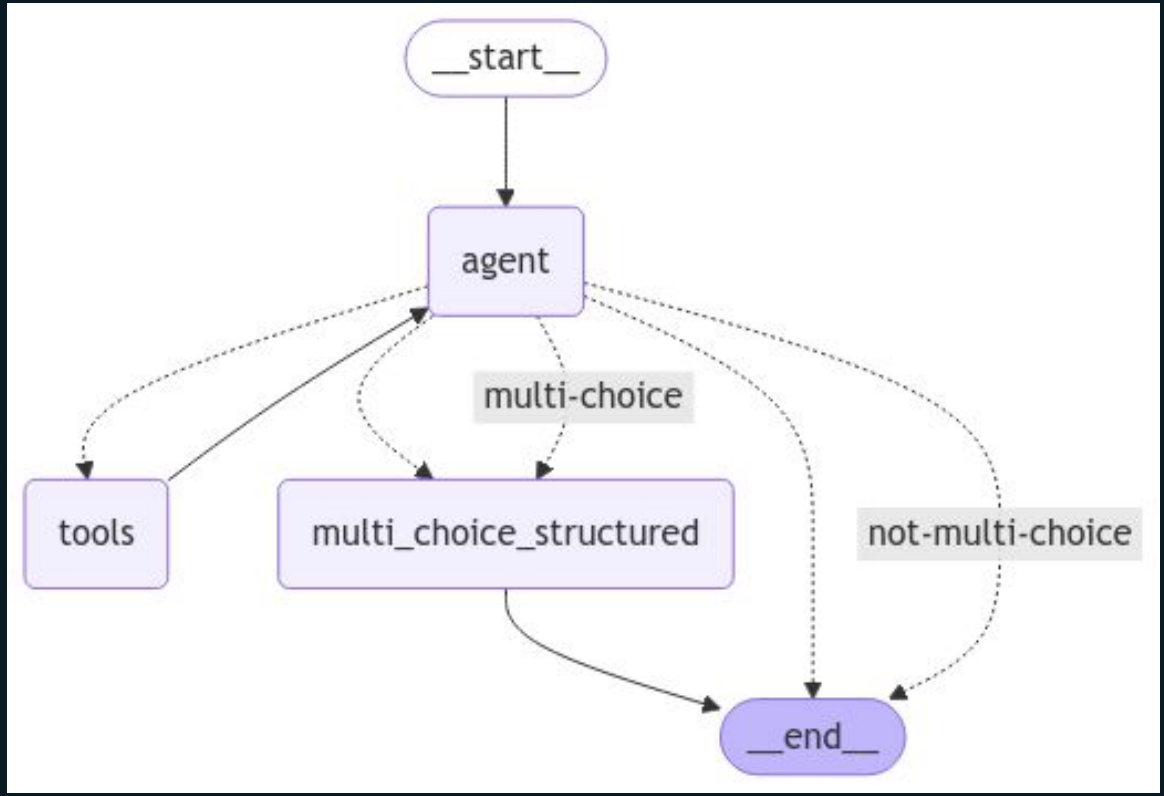
- Update the `restock-tool` description with a meaningful doc_string that provides context for the LLM.
- Implement the restock formula: `(daily_usage * lead_time) + safety_stock`
- Update the `RestockInput` class such that it receives the correct variables
- Pass the restock_tool to the exported tools list.
- Update graph to use `structured output`

Defining structured output

In production, agents will be expected to work with existing systems that will require specific schemas.

For this reason, LangChain supports an LLM call `with_structured_output` so that responses will be returned in a predictable structure.

We will modify our graph to support answering multiple choice questions vs. free-form.



Scenario 2 Demo 💻

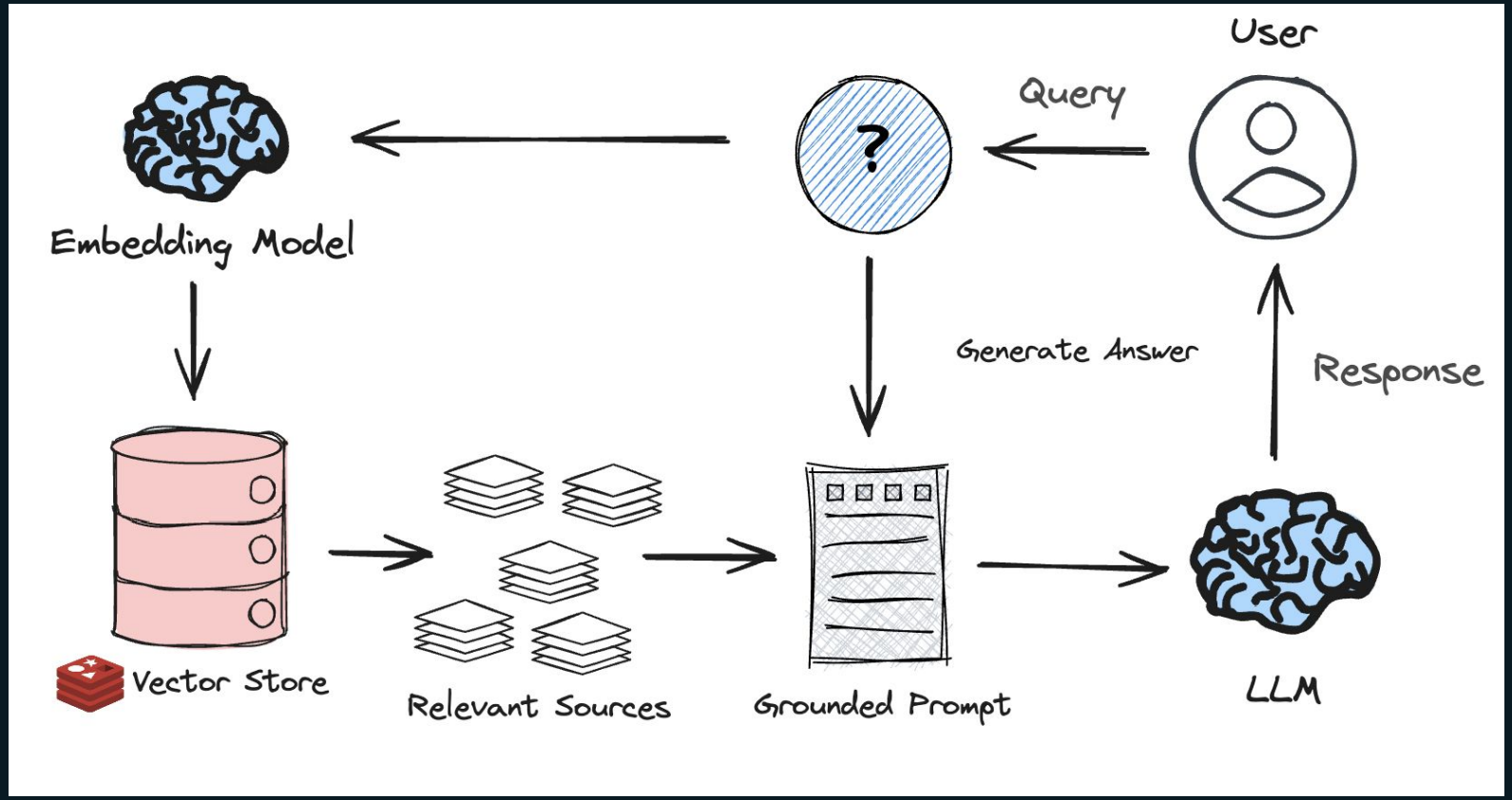
Take a breath

Scenario 3: Creating a retrieval tool

Question: You've encountered a dense forest near the Blue Mountains, and your party is unsure how to proceed. There is a fork in the road, and you must choose a path. Which way will you go?

Answer: "B" (Take the southern trail)

Retrieval Augmented Generation (RAG)



Scenario 3: Steps to complete

- Open `participant_agent/utils/vector_store.py`
- Where `vector_store=None` update to `vector_store = RedisVectorStore.from_documents(<docs>, <embedding_model>, config=<config>)` with the appropriate variables.
- Open `participant_agent/utils/tools.py`
  - Uncomment code for retrieval tool
  - Update the `create_retriever_tool` to take the correct params. Ex: `create_retriever_tool(vector_store.as_retriever(), "get_directions", "meaningful doc string")`
- Make sure the retriever tool is included in the list of tools

Scenario 3 Demo 💻

Solution

```python
def get_vector_store():
    try:
        config.from_existing = True
        vector_store = RedisVectorStore(OpenAIEmbeddings(), config=config)
    except:
        print("Init vector store with document")
        config.from_existing = False
        vector_store = RedisVectorStore.from_documents(
            [doc], OpenAIEmbeddings(), config=config
        )
    return vector_store
```

```python
## retriever tool
# see .vector_store for implementation logic
vector_store = get_vector_store()

retriever_tool = create_retriever_tool(
    vector_store.as_retriever(),
    "get_directions",
    "Search and return information related to which routes/paths/trails to take along your journey.",
)

tools = [retriever_tool, restock_tool]
```

Scenario 4: Semantic Caching

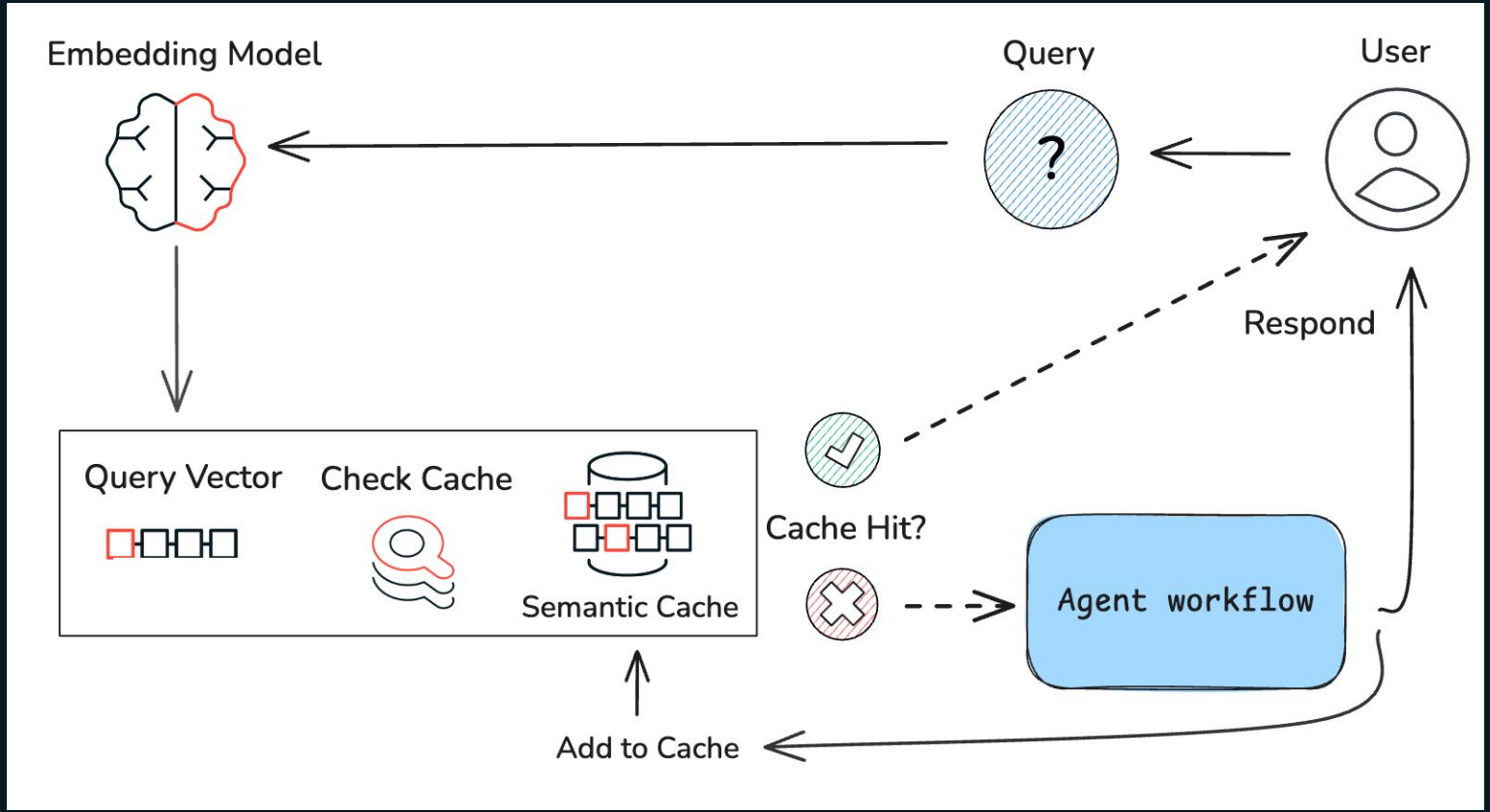
Question: ”There’s a deer. You’re hungry. You know what you have to do...”

Answer: “Bang” (must respond in sub second latency)

Steps to complete:

- Open `participant_agent/app.py` here you will see the beginner code for a semantic cache.
- A semantic cache allows us to skip the expensive and timely agent flow all together for situations where we already know the answer.

Semantic Caching



Scenario 4 Demo 

Semantic Caching - Solution

```
REDIS_URL = os.environ.get("REDIS_URL", "redis://host.docker.internal:6379/0")

# Semantic cache
hunting_example = "There's a deer. You're starving. You know what you have to do..."

semantic_cache = SemanticCache(
    name="oregon_trail_cache",
    redis_url=REDIS_URL,
    distance_threshold=0.1,
)

semantic_cache.store(prompt=hunting_example, response="bang")
```

With semantic caching in place

- Skip round trip expensive LLM calls for questions we already “know” the answer to
- Respond with sub-second latency from RAM

Scenario 5: Allow/Block List with Router

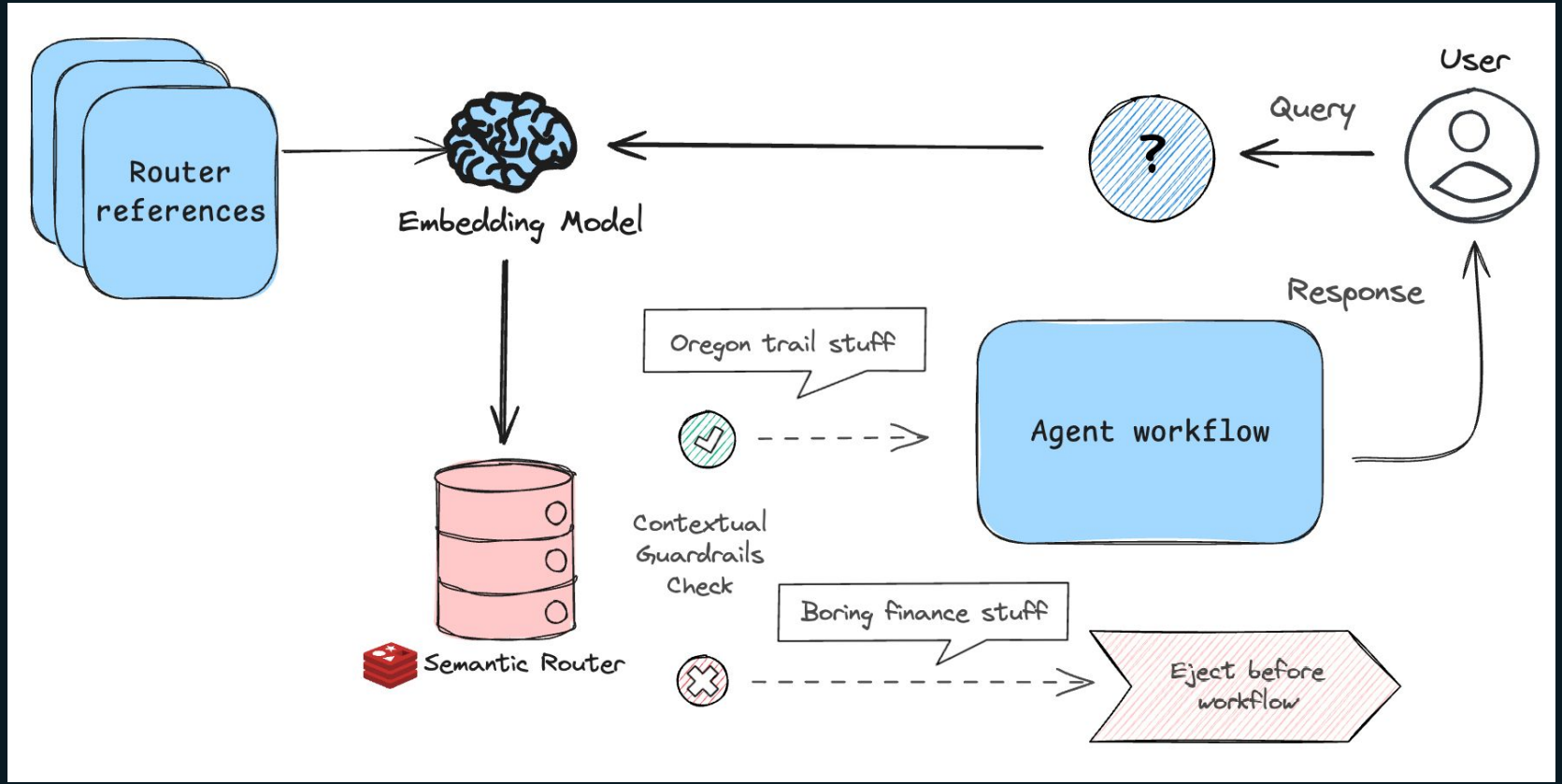
Question: ”Tell me about the S&P 500?”

Answer: ”you shall not pass”

Steps to complete:

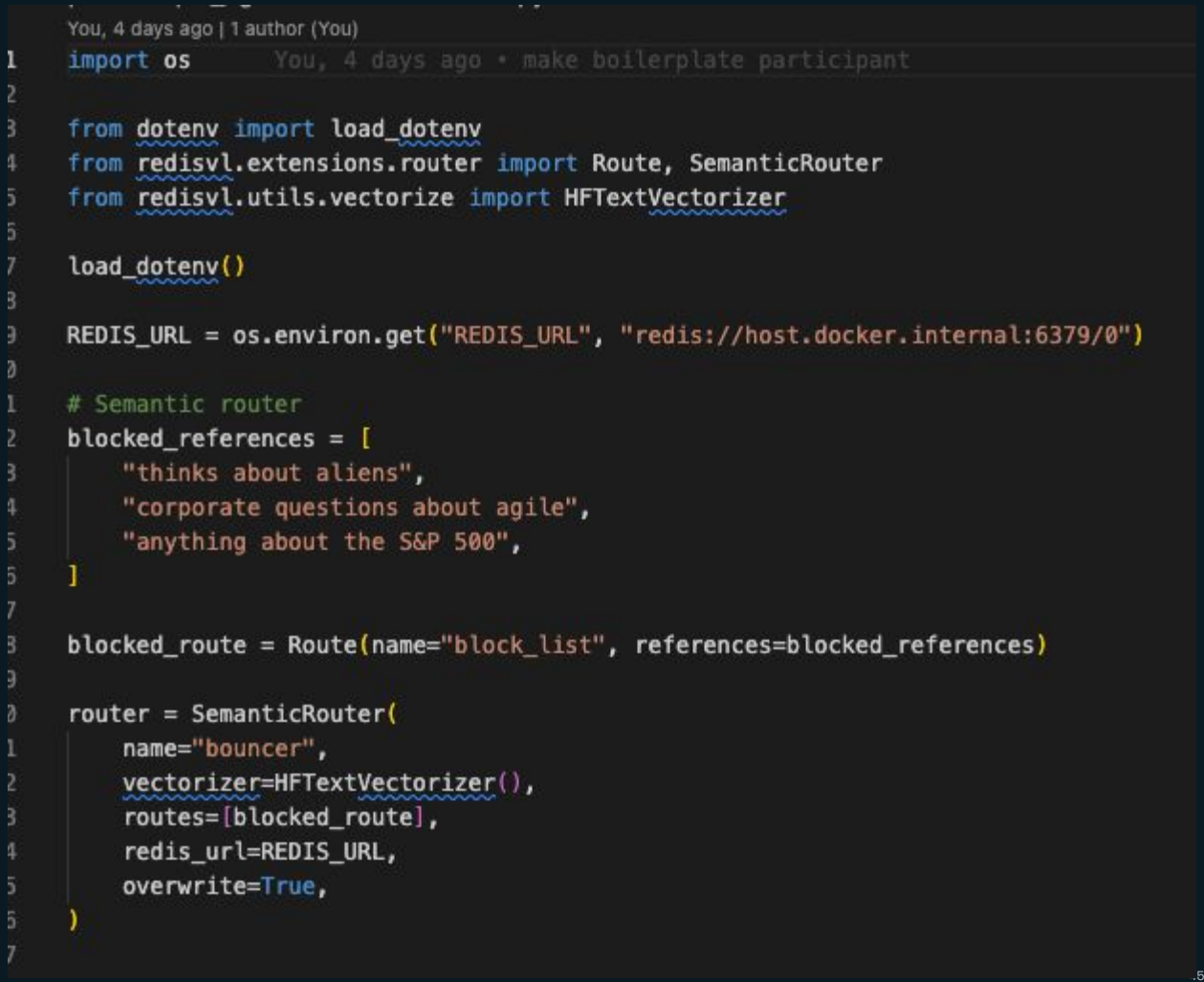
- open `participant_agent/app.py` here you will see the beginner code for a semantic router.
- A semantic router allows us to filter out topics that our agent isn’t meant to interact with such as the S&P 500 when we’re focused on the Oregon Trail.

Allow/Block List with Router



Scenario 5 Demo 

Solution



You should now be passing all test scenarios!

```
collected 5 items

test_example_oregon_trail.py::test_1_wagon_leader PASSED                                   [ 20%]
test_example_oregon_trail.py::test_2_restocking_tool PASSED                                [ 40%]
test_example_oregon_trail.py::test_3_retrieval_tool PASSED                                 [ 60%]
test_example_oregon_trail.py::test_4_semantic_cache PASSED                                 [ 80%]
test_example_oregon_trail.py::test_5_router PASSED                                         [100%]

================================== PASSES ==================================
___________________________ test_1_wagon_leader ___________________________
--------------------------- Captured stdout call ---------------------------

 What is the first name of the wagon leader?


 response: Artificial
_________________________ test_2_restocking_tool _________________________
--------------------------- Captured stdout call ---------------------------

 question: In order to survive the trail ahead, you'll need to have a restocking strategy for when you need to get more supplies or risk starving. If it takes you an estimated 3 days to restock your food and you plan to start with 2
00lbs of food, budget 10lbs/day to eat, and keep a safety stock of at least 50lbs of back up... at what point should you restock?


 Using restock tool!: daily_usage=10, lead_time=3, safety_stock=50


 response: D
__________________________ test_3_retrieval_tool __________________________
--------------------------- Captured stdout call ---------------------------

 You've encountered a dense forest near the Blue Mountains, and your party is unsure how to proceed. There is a fork in the road, and you must choose a path. Which way will you go?


 response: B
__________________________ test_4_semantic_cache __________________________
--------------------------- Captured stdout call ---------------------------

 There's a deer. You're hungry. You know what you have to do...


 response: bang
______________________________ test_5_router ______________________________
--------------------------- Captured stdout call ---------------------------

 Tell me about the S&P 500?

you shall not pass
```

Review

- You created a tool calling AI Agent
- You defined a custom tool for mathematical operations (restocking)
- You added structured output for when a system requires answers within a certain form.
- You defined a tool that implements Retrieval Augmented Generation aka RAG (retrieval tool)
- You created a semantic cache that can increase the speed and cost effectiveness of your agent workflow by short circuiting for known inputs/outputs.
- You implemented a router to protect your system from wasting time/resources on unrelated topics.

More cool stuff

Checkout (and star 😉):

- redis-ai-resources if you're interested in more AI use cases
- redisvl for the latest and greatest with the redis vector database
- this workshop anytime you'd like to review

Get in touch:

robert.shelton@redis.com

LinkedIn | GitHub