



# Building AI Agents with Redis and LangGraph

# Note

This workshop demonstrates AI agents by referencing a classic American video game known as "The Oregon Trail". Originally a text-based adventure game taking place in the mid-1800s USA, the goal of the game was to safely travel from Missouri to Oregon by wagon without succumbing to various threats and diseases.

One of the game's well known lines, "You have died of dysentery," inspired this workshop's original title, "Dodging Dysentery with AI".

# Original Abstract

Are you worn out by all the tedious mental effort of figuring out how to avoid dying of dysentery while on the Oregon trail?

Sure, you could think through some simple heuristics or pay your nephew in skittles to do it for you, but that's just so much work! I mean, it's the AI future now and I was promised that I would no longer have to think. If you too find yourself in this situation, join us and learn how to build your own Agentic RAG app to do all your Oregon trail thinking for you!



# Dodging Dysentery with AI

# Motivation

As you may have noticed, everyone is talking about AI.

And now there's talk of AI agents.

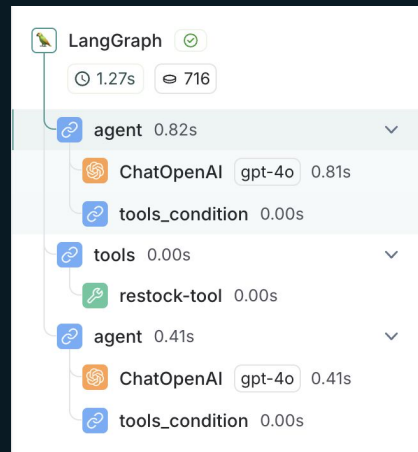
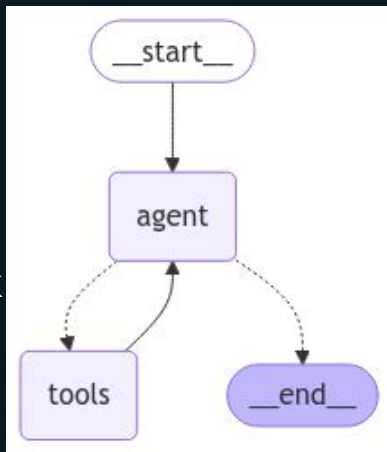
We're going to provide some clarity on these questions:

- What does “agent” mean?
- Why should I care?
- When do AI agents make sense?
- How do I build AI agents?

# What is an agentic workflow?

An **agentic workflow** is an event loop where the next step is determined by an LLM.

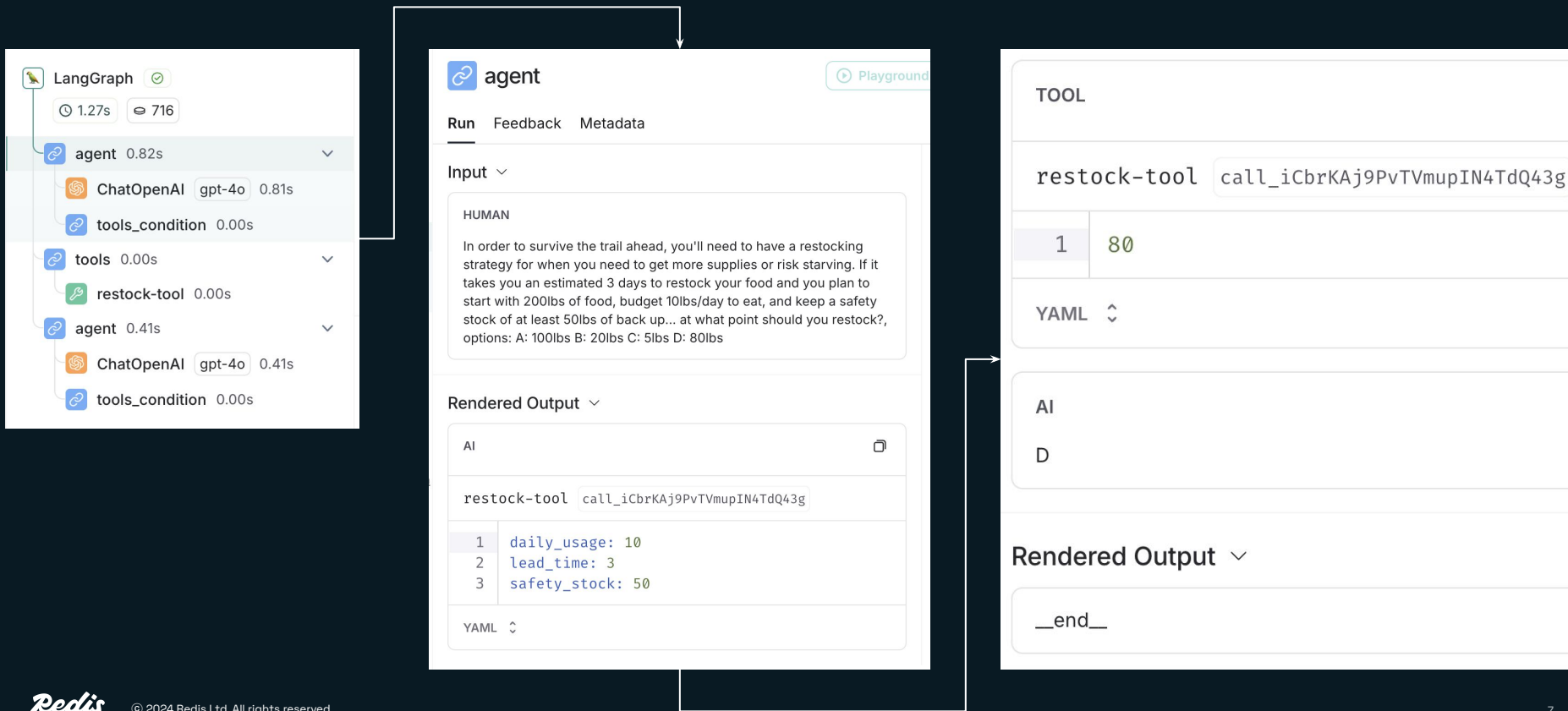
A minimal agentic architecture might look like this:



In execution:

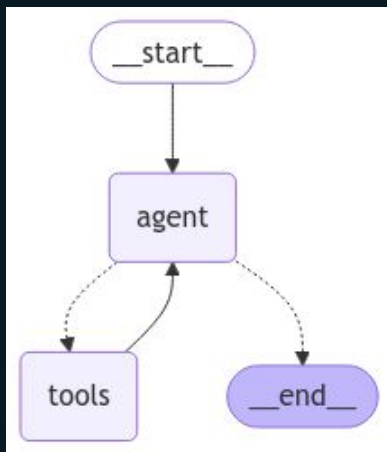
- enter flow
- agent decides on tool condition
- Invokes tool
- Back to agent with answer from tool
- End

# Concrete visual of execution



# What does a graph look like in code?

...not too bad, right?



```
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 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**
  - The set of messages passed between nodes to preserve context

## Why should we care?

The software market has shifted towards AI as a major focus\*.

Agents show significant industry potential and will be a large development focus in the coming year and beyond.

They're pretty cool.

\*\*standard hype rates apply

# When should we implement an Agentic workflow?

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, with this new development paradigm come different flavors of issues.

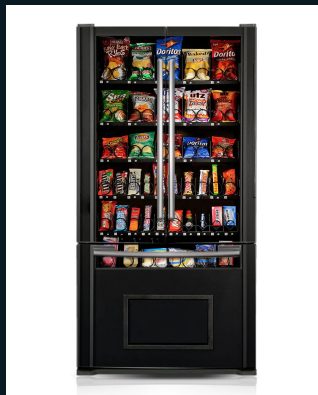
## Deterministic

### Pros:

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

### Cons:

- Brittle in the face of unexpected input
- Limited UI flexibility



## Probabilistic

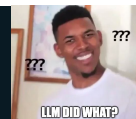
### Pros:

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

### Cons:

- Expected output not guaranteed
- Relatively expensive and slow

		True Class	
		Positive	Negative
Predicated Class	Positive	TP	FP
	Negative	FN	TN



# Confusion matrix?

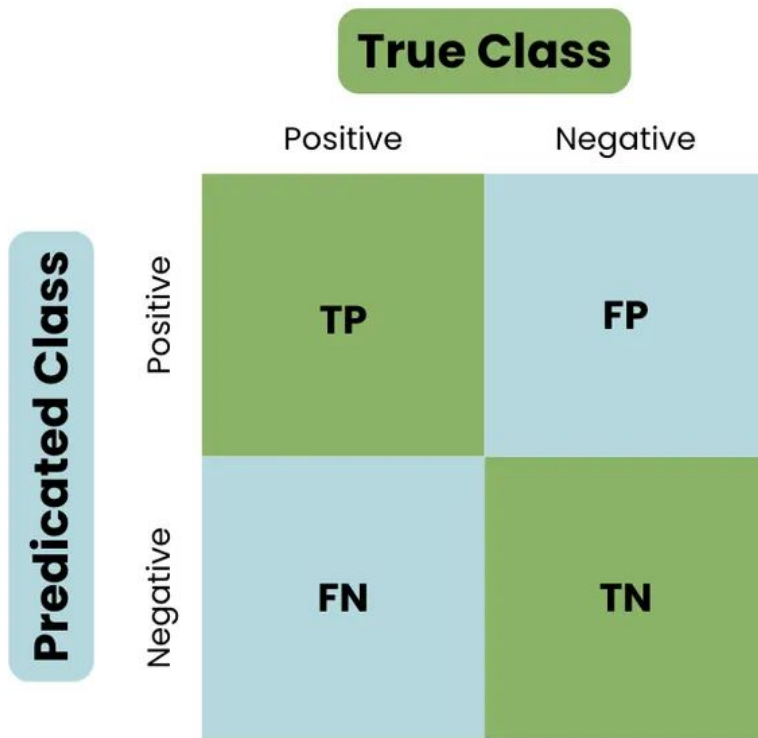
A confusion matrix maps the potential outcomes of a prediction.

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 Dysentery?!

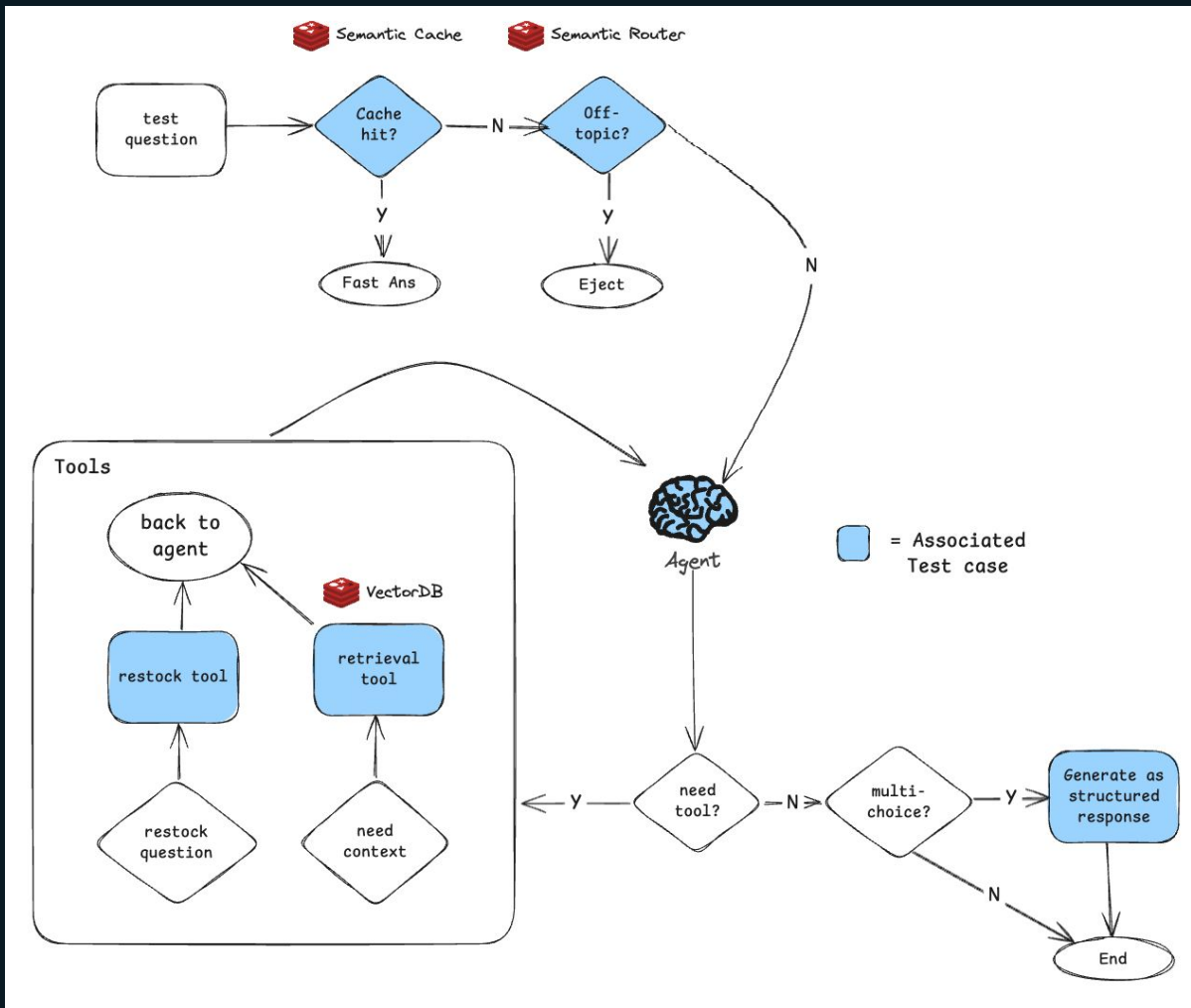
In practice, many applications are like the classic Oregon Trail game. You need to:

- Use tools when necessary
- Hunt for additional relevant information
- Format and respond to inquiries appropriately
- Do most of your work through a command line

To survive the Oregon Trail and to make a prod ready app, we need more than just an LLM call.

We will implement:

- Agentic graph
- Semantic cache
- Router





And so it begins...

# 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](#)



# General workshop flow

- You will be working to pass all 5 test scenarios by updating code in the `participant_agent/` folder.
- **All** steps are in 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 stages and try to keep to the median pace of the group.
- 🙋 Raise your hand if you need help!
- If you get **extremely lost**, there is a completed example in the `example_agent/` folder.

# 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

# Setup Demo

# 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 are in README.md**

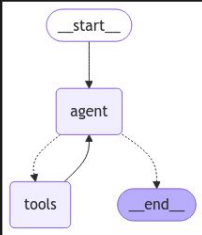


pause

# 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!

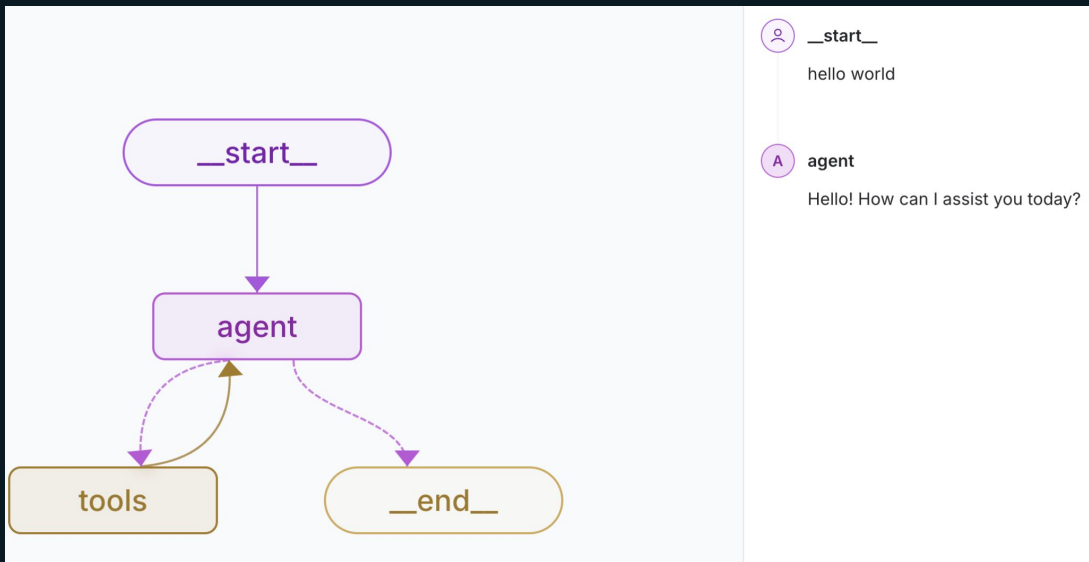
```
from participant_agent.graph import graph
from IPython.display import Image, display
display(Image(graph.get_graph(xray=True).draw_mermaid_png()))
✓ 1.3s
```



```
graph TD
    start([__start__]) --> agent[agent]
    agent --> tools[tools]
    tools -.-> agent
    agent --> end([__end__])
```

```
graph.invoke({"messages": [{"content": "hello world"}]})
✓ 1.4s
```

```
{'messages': [HumanMessage(content='hello world', additional_kwargs={}),
               AIMessage(content='Hello! How can I assist you today?', additional_kwargs={})]}
```



# Scenario 1 Demo

# Expected

```
# TODO: define the graph to be used in testing
workflow = StateGraph(AgentState, config_schema=GraphConfig)

# node 1
workflow.add_node("agent", call_tool_model)
# node 2
workflow.add_node("tools", tool_node)

# entry
workflow.set_entry_point("agent")

# Conditional edge
workflow.add_conditional_edges("agent", tools_condition)

# We now add a normal edge.
workflow.add_edge("tools", "agent")

# Compiled graph will be picked up by workflow
graph = workflow.compile()
```

# Run tests - only one test will pass at this point

In command line:

```
_alias test_trail_agent="pytest --disable-warnings -vv -rP  
test_participant_oregon_trail.py"
```

```
_test_trail_agent
```

```
test_participant_oregon_trail.py::test_1_wagon_leader PASSED  
test_participant_oregon_trail.py::test_2_restocking_tool FAILED  
test_participant_oregon_trail.py::test_3_retrieval_tool FAILED  
test_participant_oregon_trail.py::test_4_semantic_cache FAILED  
test_participant_oregon_trail.py::test_5_router FAILED
```



## Scenario 1: Defining a more advanced tool

**Question:** In order to survive the trail ahead, you'll need to have a restocking strategy to get more supplies and avoid starving. If it takes you an estimated 3 days to restock your food, and you plan to start with 200 lbs. of food, budget 10 lbs./day to eat, and keep a safety stock of at least 50 lbs. of reserves... 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:  $(\text{daily\_usage} * \text{lead\_time}) + \text{safety\_stock}$
- 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`



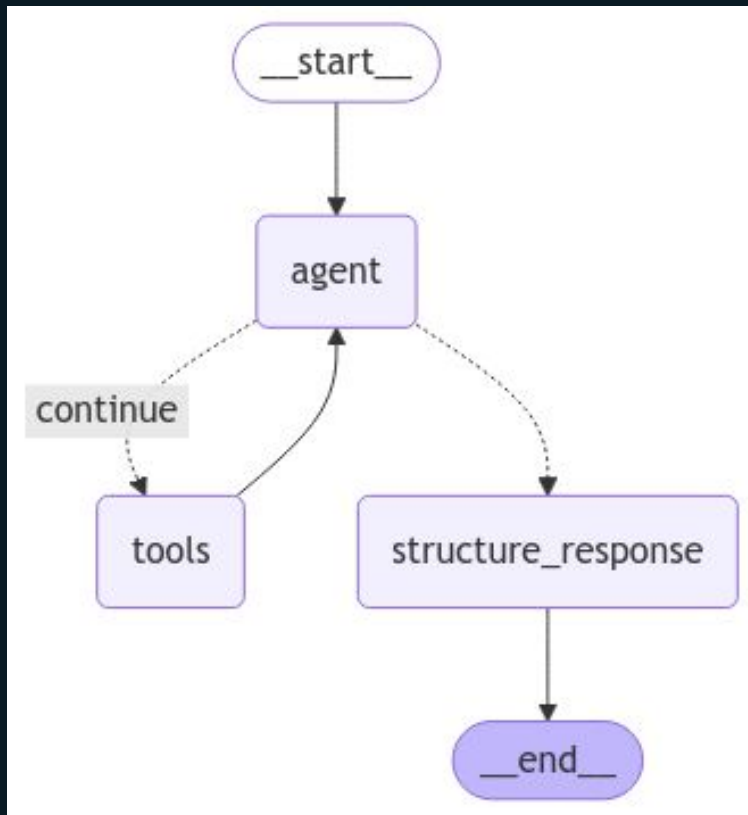
# Defining structured output

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



# Scenario 2 Demo

# Expected graph

```
# 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("tools", tool_node)
workflow.add_node("structure_response", structure_response)

# 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 between `agent` and `tools`.
workflow.add_conditional_edges(
    "agent",
    should_continue,
    {"continue": "tools", "structure_response": "structure_response"},
)

# 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("structure_response", 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()
```

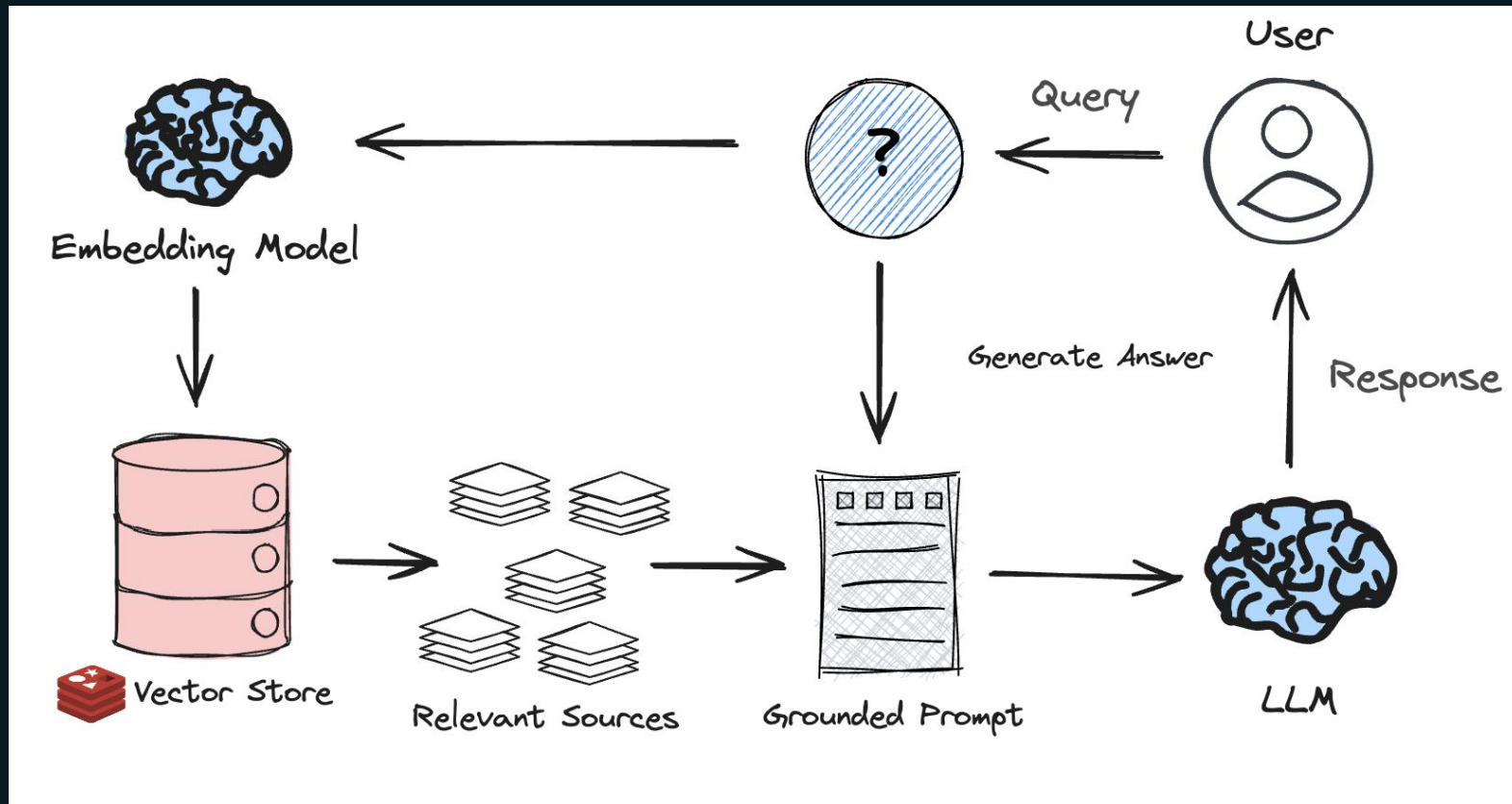
Take a breath

## 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

```
3 def get_vector_store():
4     try:
5         config.from_existing = True
6         vector_store = RedisVectorStore(OpenAIEmbeddings(), config=config)
7     except:
8         print("Init vector store with document")
9         config.from_existing = False
10        vector_store = RedisVectorStore.from_documents(
11            [doc], OpenAIEmbeddings(), config=config
12        )
13    return vector_store
```

```
3 ## retriever tool
4 # see .vector_store for implementation logic
5 vector_store = get_vector_store()
6
7 retriever_tool = create_retriever_tool(
8     vector_store.as_retriever(),
9     "get_directions",
10    "Search and return information related to which routes/paths/trails to take along your journey.",
11 )
12
13 tools = [retriever_tool, restock_tool]
```

## Scenario 4: Semantic Caching

**Question:** "There's a deer. You're hungry. You know what you have to do..."

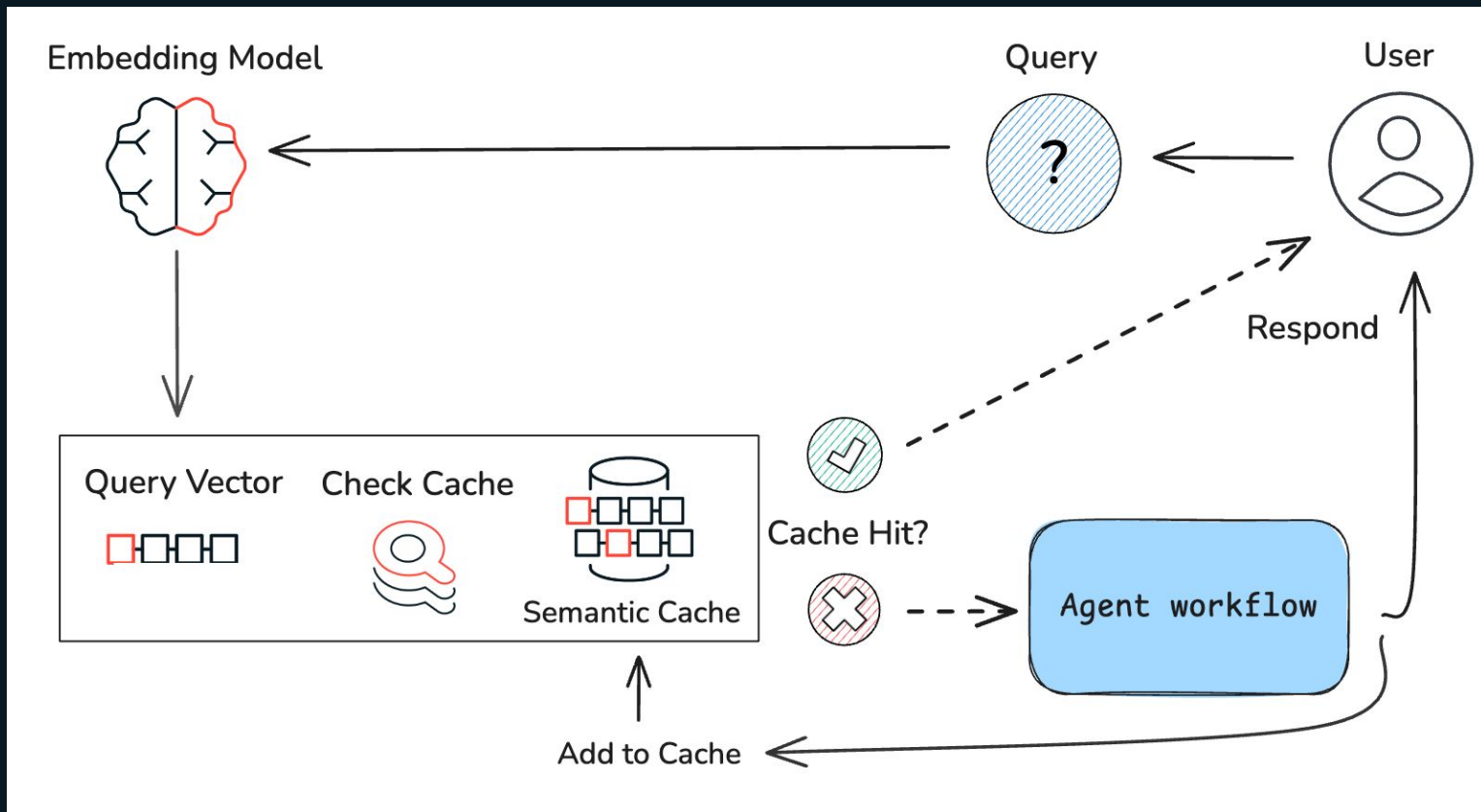
**Answer:** "Bang" (must respond in sub second latency)

Steps to complete:

- Open `participant_agent/app.py`. Here you will see the starting code for a semantic cache.
- A semantic cache allows us to skip the expensive and timely agent flow altogether in situations when 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 expensive, round trip 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?"

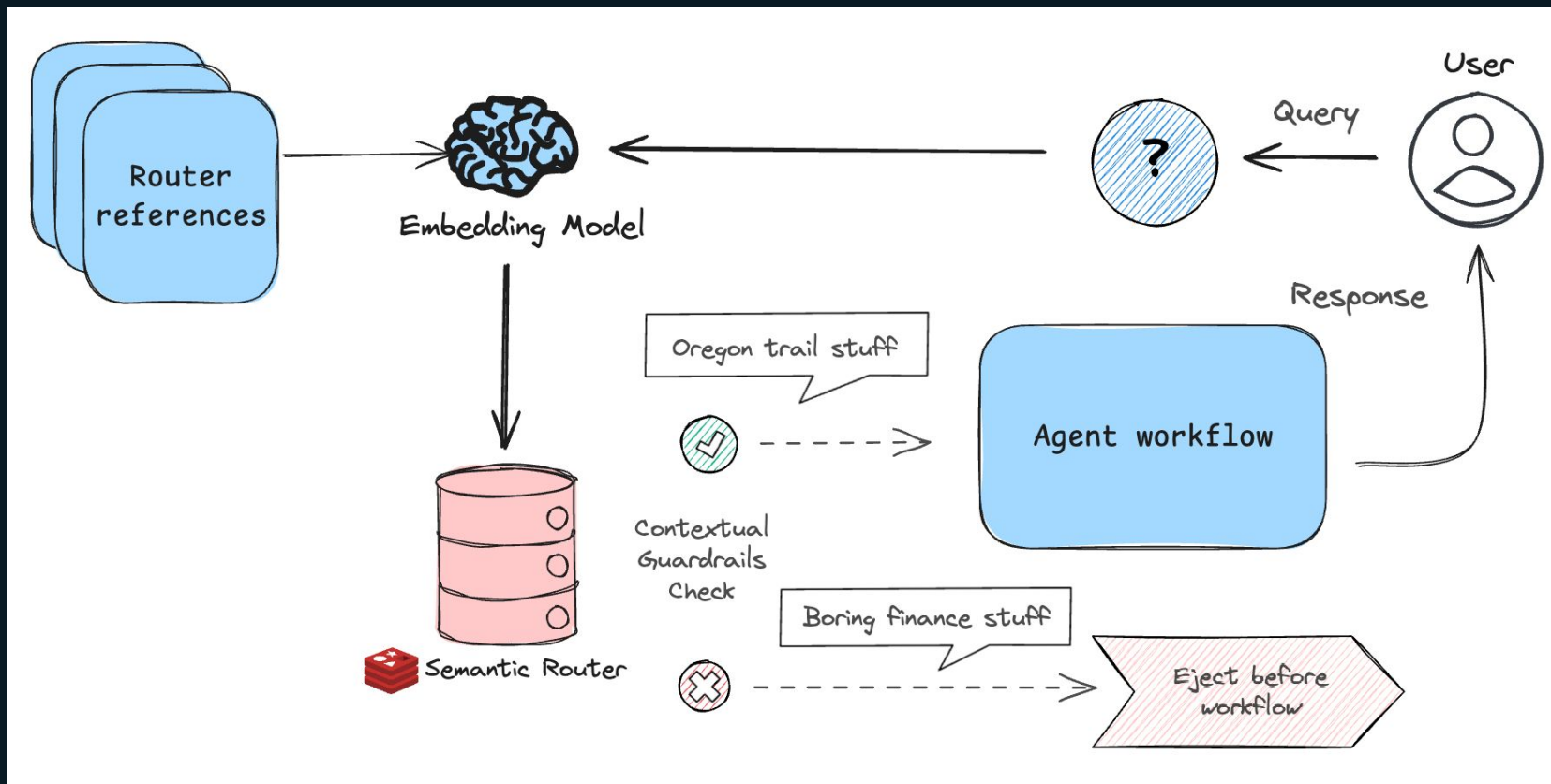
**Answer:** "you shall not pass"

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, 4 days ago | 1 author (You)
1 import os
2
3 from dotenv import load_dotenv
4 from redisvl.extensions.router import Route, SemanticRouter
5 from redisvl.utils.vectorize import HFTextVectorizer
6
7 load_dotenv()
8
9 REDIS_URL = os.environ.get("REDIS_URL", "redis://host.docker.internal:6379/0")
10
11 # Semantic router
12 blocked_references = [
13     "thinks about aliens",
14     "corporate questions about agile",
15     "anything about the S&P 500",
16 ]
17
18 blocked_route = Route(name="block_list", references=blocked_references)
19
20 router = SemanticRouter(
21     name="bouncer",
22     vectorizer=HFTextVectorizer(),
23     routes=[blocked_route],
24     redis_url=REDIS_URL,
25     overwrite=True,
26 )
27
```

# 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 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?

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



Congratulations! You have  
made it to Oregon! Let's  
see how many points you have  
received.

Willamette Valley  
September 29, 1848

Press SPACE BAR to continue

# 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 with a certain form.
- You defined a tool that implements Retrieval Augmented Generation, a.k.a., RAG
- You created a semantic cache to 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

Check out (and star 🤔):

- [redis-ai-resources](#) if you're interested in more AI use cases
- [redisvl](#) for a Python library to simplify the Redis vector database
- [this workshop](#) anytime you'd like to review

Get in touch:

[robert.shelton@redis.com](mailto:robert.shelton@redis.com)

[LinkedIn](#) | [GitHub](#)