Orchestrating Agents: Routines and Handoffs


Ilan Bigio
Oct 10, 2024

Open in Github

When working with language models, quite often all you need for solid performance is a good prompt and the right tools. However, when dealing with many unique flows, things may get hairy. This cookbook will walk through one way to tackle this.

We'll introduce the notion of **routines** and **handoffs**, then walk through the implementation and show how they can be used to orchestrate multiple agents in a simple, powerful, and controllable way.

Finally, we provide a sample repo, Swarm, that implements these ideas along with examples.

Let's start by setting up our imports.

```
from openai import OpenAI
from pydantic import BaseModel
from typing import Optional
import json


client = OpenAI()
```

Routines

The notion of a "routine" is not strictly defined, and instead meant to capture the idea of a set of steps. Conretely, let's define a routine to be a list of instructions in natural langauge (which we'll repreesnt with a system prompt), along with the tools necessary to complete them.

Let's take a look at an example. Below, we've defined a routine for a customer service agent instructing it to triage the user issue, then either suggest a fix or provide a refund. We've also defined the necessary functions `execute_refund` and `look_up_item`. We can call this a

customer service routine, agent, assistant, etc – however the idea itself is the same: a set of steps and the tools to execute them.

```
# Customer Service Routine

system_message = (
    "You are a customer support agent for ACME Inc."
    "Always answer in a sentence or less."
    "Follow the following routine with the user:"
    "1. First, ask probing questions and understand the user's problem deeper.\n"
    " - unless the user has already provided a reason.\n"
    "2. Propose a fix (make one up).\n"
    "3. ONLY if not satesfied, offer a refund.\n"
    "4. If accepted, search for the ID and then execute refund."
    ""
)

def look_up_item(search_query):
    """Use to find item ID.
    Search query can be a description or keywords."""

    # return hard-coded item ID - in reality would be a lookup
    return "item_132612938"


def execute_refund(item_id, reason="not provided"):

    print("Summary:", item_id, reason) # lazy summary
    return "success"
```

The main power of routines is their simplicity and robustness. Notice that these instructions contain conditionals much like a state machine or branching in code. LLMs can actually handle these cases quite robustly for small and medium sized routine, with the added benefit of having "soft" adherance – the LLM can naturally steer the conversation without getting stuck in dead-ends.

Executing Routines

To execute a routine, let's implement a simple loop that:

1. Gets user input.
2. Appends user message to `messages`.
3. Calls the model.
4. Appends model response to `messages`.

```
def run_full_turn(system_message, messages):
    response = client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[{"role": "system", "content": system_message}] + messages,
```

```
    )
    message = response.choices[0].message
    messages.append(message)

    if message.content: print("Assistant:", message.content)

    return message


messages = []
while True:
    user = input("User: ")
    messages.append({"role": "user", "content": user})

    run_full_turn(system_message, messages)
```

As you can see, this currently ignores function calls, so let's add that.

Models require functions to be formatted as a function schema. For convenience, we can define a helper function that turns python functions into the corresponding function schema.

```
import inspect

def function_to_schema(func) -> dict:
    type_map = {
        str: "string",
        int: "integer",
        float: "number",
        bool: "boolean",
        list: "array",
        dict: "object",
        type(None): "null",
    }

    try:
        signature = inspect.signature(func)
    except ValueError as e:
        raise ValueError(
            f"Failed to get signature for function {func.__name__}: {str(e)}"
        )

    parameters = {}
    for param in signature.parameters.values():
        try:
            param_type = type_map.get(param.annotation, "string")
        except KeyError as e:
            raise KeyError(
                f"Unknown type annotation {param.annotation} for parameter {param.name}: {str(e)}"
            )
        parameters[param.name] = {"type": param_type}

    required = [
        param.name
        for param in signature.parameters.values()
        if param.default == inspect._empty
    ]

    return {
        "type": "function",
        "function": {
            "name": func.__name__,
```

```
            "description": (func.__doc__ or "").strip(),
            "parameters": {
                "type": "object",
                "properties": parameters,
                "required": required,
            },
        },
    }
```

For example:

```
def sample_function(param_1, param_2, the_third_one: int, some_optional="John Doe"):
    """
    This is my docstring. Call this function when you want.
    """
    print("Hello, world")

schema =  function_to_schema(sample_function)
print(json.dumps(schema, indent=2))
```

```
{
  "type": "function",
  "function": {
    "name": "sample_function",
    "description": "This is my docstring. Call this function when you want.",
    "parameters": {
      "type": "object",
      "properties": {
        "param_1": {
          "type": "string"
        },
        "param_2": {
          "type": "string"
        },
        "the_third_one": {
          "type": "integer"
        },
        "some_optional": {
          "type": "string"
        }
      },
      "required": [
        "param_1",
        "param_2",
        "the_third_one"
      ]
    }
  }
}
```

Now, we can use this function to pass the tools to the model when we call it.

```
messages = []

tools = [execute_refund, look_up_item]
tool_schemas = [function_to_schema(tool) for tool in tools]

response = client.chat.completions.create(
```

```
            model="gpt-4o-mini",
            messages=[{"role": "user", "content": "Look up the black boot."}],
            tools=tool_schemas,
        )
message = response.choices[0].message

message.tool_calls[0].function
```

```
Function(arguments='{"search_query":"black boot"}', name='look_up_item')
```

Finally, when the model calls a tool we need to execute the corresponding function and provide the result back to the model.

We can do this by mapping the name of the tool to the python function in a `tool_map`, then looking it up in `execute_tool_call` and calling it. Finally we add the result to the conversation.

```
tools_map = {tool.__name__: tool for tool in tools}

def execute_tool_call(tool_call, tools_map):
    name = tool_call.function.name
    args = json.loads(tool_call.function.arguments)

    print(f"Assistant: {name}({args})")

    # call corresponding function with provided arguments
    return tools_map[name](**args)

for tool_call in message.tool_calls:
            result = execute_tool_call(tool_call, tools_map)

            # add result back to conversation
            result_message = {
                "role": "tool",
                "tool_call_id": tool_call.id,
                "content": result,
            }
            messages.append(result_message)
```

```
Assistant: look_up_item({'search_query': 'black boot'})
```

In practice, we'll also want to let the model use the result to produce another response. That response might *also* contain a tool call, so we can just run this in a loop until there are no more tool calls.

If we put everything together, it will look something like this:

```
tools = [execute_refund, look_up_item]


def run_full_turn(system_message, tools, messages):

    num_init_messages = len(messages)
```

```
    messages = messages.copy()

    while True:

        # turn python functions into tools and save a reverse map
        tool_schemas = [function_to_schema(tool) for tool in tools]
        tools_map = {tool.__name__: tool for tool in tools}

        # === 1. get openai completion ===
        response = client.chat.completions.create(
            model="gpt-4o-mini",
            messages=[{"role": "system", "content": system_message}] + messages,
            tools=tool_schemas or None,
        )
        message = response.choices[0].message
        messages.append(message)

        if message.content:  # print assistant response
            print("Assistant:", message.content)

        if not message.tool_calls:  # if finished handling tool calls, break
            break

        # === 2. handle tool calls ===

        for tool_call in message.tool_calls:
            result = execute_tool_call(tool_call, tools_map)

            result_message = {
                "role": "tool",
                "tool_call_id": tool_call.id,
                "content": result,
            }
            messages.append(result_message)

    # ==== 3. return new messages =====
    return messages[num_init_messages:]


def execute_tool_call(tool_call, tools_map):
    name = tool_call.function.name
    args = json.loads(tool_call.function.arguments)

    print(f"Assistant: {name}({args})")

    # call corresponding function with provided arguments
    return tools_map[name](**args)


messages = []
while True:
    user = input("User: ")
    messages.append({"role": "user", "content": user})

    new_messages = run_full_turn(system_message, tools, messages)
    messages.extend(new_messages)
```

Now that we have a routine, let's say we want to add more steps and more tools. We can up to a point, but eventually if we try growing the routine with too many different tasks it may start to struggle. This is where we can leverage the notion of multiple routines – given a user request, we can load the right routine with the appropriate steps and tools to address it.

Dynamically swapping system instructions and tools may seem daunting. However, if we view "routines" as "agents", then this notion of **handoffs** allow us to represent these swaps simply – as one agent handing off a conversation to another.

Handoffs

Let's define a **handoff** as an agent (or routine) handing off an active conversation to another agent, much like when you get transfered to someone else on a phone call. Except in this case, the agents have complete knowledge of your prior conversation!

To see handoffs in action, let's start by defining a basic class for an Agent.

```
class Agent(BaseModel):
    name: str = "Agent"
    model: str = "gpt-4o-mini"
    instructions: str = "You are a helpful Agent"
    tools: list = []
```

Now to make our code support it, we can change `run_full_turn` take an `Agent` instead of separate `system_message` and `tools`:

```
def run_full_turn(agent, messages):

    num_init_messages = len(messages)
    messages = messages.copy()

    while True:

        # turn python functions into tools and save a reverse map
        tool_schemas = [function_to_schema(tool) for tool in agent.tools]
        tools_map = {tool.__name__: tool for tool in agent.tools}

        # === 1. get openai completion ===
        response = client.chat.completions.create(
            model=agent.model,
            messages=[{"role": "system", "content": agent.instructions}] + messages,
            tools=tool_schemas or None,
        )
        message = response.choices[0].message
        messages.append(message)

        if message.content:  # print assistant response
            print("Assistant:", message.content)

        if not message.tool_calls:  # if finished handling tool calls, break
            break

        # === 2. handle tool calls ===

        for tool_call in message.tool_calls:
            result = execute_tool_call(tool_call, tools_map)
```

```
            result_message = {
                "role": "tool",
                "tool_call_id": tool_call.id,
                "content": result,
            }
            messages.append(result_message)

    # ==== 3. return new messages =====
    return messages[num_init_messages:]


def execute_tool_call(tool_call, tools_map):
    name = tool_call.function.name
    args = json.loads(tool_call.function.arguments)

    print(f"Assistant: {name}({args})")

    # call corresponding function with provided arguments
    return tools_map[name](**args)
```

We can now run multiple agents easily:

```
def execute_refund(item_name):
    return "success"

refund_agent = Agent(
    name="Refund Agent",
    instructions="You are a refund agent. Help the user with refunds.",
    tools=[execute_refund],
)

def place_order(item_name):
    return "success"

sales_assistant = Agent(
    name="Sales Assistant",
    instructions="You are a sales assistant. Sell the user a product.",
    tools=[place_order],
)


messages = []
user_query = "Place an order for a black boot."
print("User:", user_query)
messages.append({"role": "user", "content": user_query})

response = run_full_turn(sales_assistant, messages) # sales assistant
messages.extend(response)


user_query = "Actually, I want a refund." # implitly refers to the last item
print("User:", user_query)
messages.append({"role": "user", "content": user_query})
response = run_full_turn(refund_agent, messages) # refund agent
```

```
User: Place an order for a black boot.
Assistant: place_order({'item_name': 'black boot'})
Assistant: Your order for a black boot has been successfully placed! If you need anything else, feel fre
User: Actually, I want a refund.
Assistant: execute_refund({'item_name': 'black boot'})
Assistant: Your refund for the black boot has been successfully processed. If you need further assistanc
```

Great! But we did the handoff manually here – we want the agents themselves to decide when to perform a handoff. A simple, but surprisingly effective way to do this is by giving them a `transfer_to_XXX` function, where `XXX` is some agent. The model is smart enough to know to call this function when it makes sense to make a handoff!

Handoff Functions

Now that agent can express the *intent* to make a handoff, we must make it actually happen. There's many ways to do this, but there's one particularly clean way.

For the agent functions we've defined so far, like `execute_refund` or `place_order` they return a string, which will be provided to the model. What if instead, we return an `Agent` object to indate which agent we want to transfer to? Like so:

```
refund_agent = Agent(
    name="Refund Agent",
    instructions="You are a refund agent. Help the user with refunds.",
    tools=[execute_refund],
)

def transfer_to_refunds():
    return refund_agent

sales_assistant = Agent(
    name="Sales Assistant",
    instructions="You are a sales assistant. Sell the user a product.",
    tools=[place_order],
)
```

We can then update our code to check the return type of a function response, and if it's an `Agent`, update the agent in use! Additionally, now `run_full_turn` will need to return the latest agent in use in case there are handoffs. (We can do this in a `Response` class to keep things neat.)

```
class Response(BaseModel):
    agent: Optional[Agent]
    messages: list
```

Now for the updated `run_full_turn`:

```
def run_full_turn(agent, messages):

    current_agent = agent
    num_init_messages = len(messages)
    messages = messages.copy()
```

```
    while True:

        # turn python functions into tools and save a reverse map
        tool_schemas = [function_to_schema(tool) for tool in current_agent.tools]
        tools = {tool.__name__: tool for tool in current_agent.tools}

        # === 1. get openai completion ===
        response = client.chat.completions.create(
            model=agent.model,
            messages=[{"role": "system", "content": current_agent.instructions}]
            + messages,
            tools=tool_schemas or None,
        )
        message = response.choices[0].message
        messages.append(message)

        if message.content:  # print agent response
            print(f"{current_agent.name}:", message.content)

        if not message.tool_calls:  # if finished handling tool calls, break
            break

        # === 2. handle tool calls ===

        for tool_call in message.tool_calls:
            result = execute_tool_call(tool_call, tools, current_agent.name)

            if type(result) is Agent:  # if agent transfer, update current agent
                current_agent = result
                result = (
                    f"Transfered to {current_agent.name}. Adopt persona immediately."
                )

            result_message = {
                "role": "tool",
                "tool_call_id": tool_call.id,
                "content": result,
            }
            messages.append(result_message)

    # ==== 3. return last agent used and new messages =====
    return Response(agent=current_agent, messages=messages[num_init_messages:])


def execute_tool_call(tool_call, tools, agent_name):
    name = tool_call.function.name
    args = json.loads(tool_call.function.arguments)

    print(f"{agent_name}:", f"{name}({args})")

    return tools[name](**args)  # call corresponding function with provided arguments
```

Let's look at an example with more Agents.

```
def escalate_to_human(summary):
    """Only call this if explicitly asked to."""
    print("Escalating to human agent...")
    print("\n=== Escalation Report ===")
    print(f"Summary: {summary}")
    print("=========================\n")
    exit()
```

```python
def transfer_to_sales_agent():
    """User for anything sales or buying related."""
    return sales_agent


def transfer_to_issues_and_repairs():
    """User for issues, repairs, or refunds."""
    return issues_and_repairs_agent


def transfer_back_to_triage():
    """Call this if the user brings up a topic outside of your purview,
    including escalating to human."""
    return triage_agent


triage_agent = Agent(
    name="Triage Agent",
    instructions=(
        "You are a customer service bot for ACME Inc. "
        "Introduce yourself. Always be very brief. "
        "Gather information to direct the customer to the right department. "
        "But make your questions subtle and natural."
    ),
    tools=[transfer_to_sales_agent, transfer_to_issues_and_repairs, escalate_to_human],
)


def execute_order(product, price: int):
    """Price should be in USD."""
    print("\n\n=== Order Summary ===")
    print(f"Product: {product}")
    print(f"Price: ${price}")
    print("=================\n")
    confirm = input("Confirm order? y/n: ").strip().lower()
    if confirm == "y":
        print("Order execution successful!")
        return "Success"
    else:
        print("Order cancelled!")
        return "User cancelled order."


sales_agent = Agent(
    name="Sales Agent",
    instructions=(
        "You are a sales agent for ACME Inc."
        "Always answer in a sentence or less."
        "Follow the following routine with the user:"
        "1. Ask them about any problems in their life related to catching roadrunners.\n"
        "2. Casually mention one of ACME's crazy made-up products can help.\n"
        " - Don't mention price.\n"
        "3. Once the user is bought in, drop a ridiculous price.\n"
        "4. Only after everything, and if the user says yes, "
        "tell them a crazy caveat and execute their order.\n"
        ""
    ),
    tools=[execute_order, transfer_back_to_triage],
)


def look_up_item(search_query):
    """Use to find item ID.
    Search query can be a description or keywords."""
    item_id = "item_132612938"
    print("Found item:", item_id)
```

```
    return item_id


def execute_refund(item_id, reason="not provided"):
    print("\n\n=== Refund Summary ===")
    print(f"Item ID: {item_id}")
    print(f"Reason: {reason}")
    print("=================\n")
    print("Refund execution successful!")
    return "success"


issues_and_repairs_agent = Agent(
    name="Issues and Repairs Agent",
    instructions=(
        "You are a customer support agent for ACME Inc."
        "Always answer in a sentence or less."
        "Follow the following routine with the user:"
        "1. First, ask probing questions and understand the user's problem deeper.\n"
        " - unless the user has already provided a reason.\n"
        "2. Propose a fix (make one up).\n"
        "3. ONLY if not satesfied, offer a refund.\n"
        "4. If accepted, search for the ID and then execute refund."
        ""
    ),
    tools=[execute_refund, look_up_item, transfer_back_to_triage],
)
```

Finally, we can run this in a loop (this won't run in python notebooks, so you can try this in a separate python file):

```
agent = triage_agent
messages = []

while True:
    user = input("User: ")
    messages.append({"role": "user", "content": user})

    response = run_full_turn(agent, messages)
    agent = response.agent
    messages.extend(response.messages)
```

Swarm

As a proof of concept, we've packaged these ideas into a sample library called Swarm. It is meant as an example only, and should not be directly used in production. However, feel free to take the ideas and code to build your own!