Chapter 5. Orchestration

Now that your agent has a set of tools that can be used, it's time to orchestrate them to solve real tasks. Orchestration involves more than just deciding which tools to call and when—it also requires constructing the right context for each model invocation to ensure effective, grounded actions. While simple tasks may only need a single tool and minimal context, more complex workflows demand careful planning, memory retrieval, and dynamic context assembly to perform each step accurately. In this chapter, we'll cover orchestration strategies, context engineering, tool selection, execution, and planning topologies to build agents capable of handling realistic, multistep tasks efficiently and reliably. As we can see in Figure 5-1, orchestration is how the system utilizes the resources at its disposal to address the user query effectively.

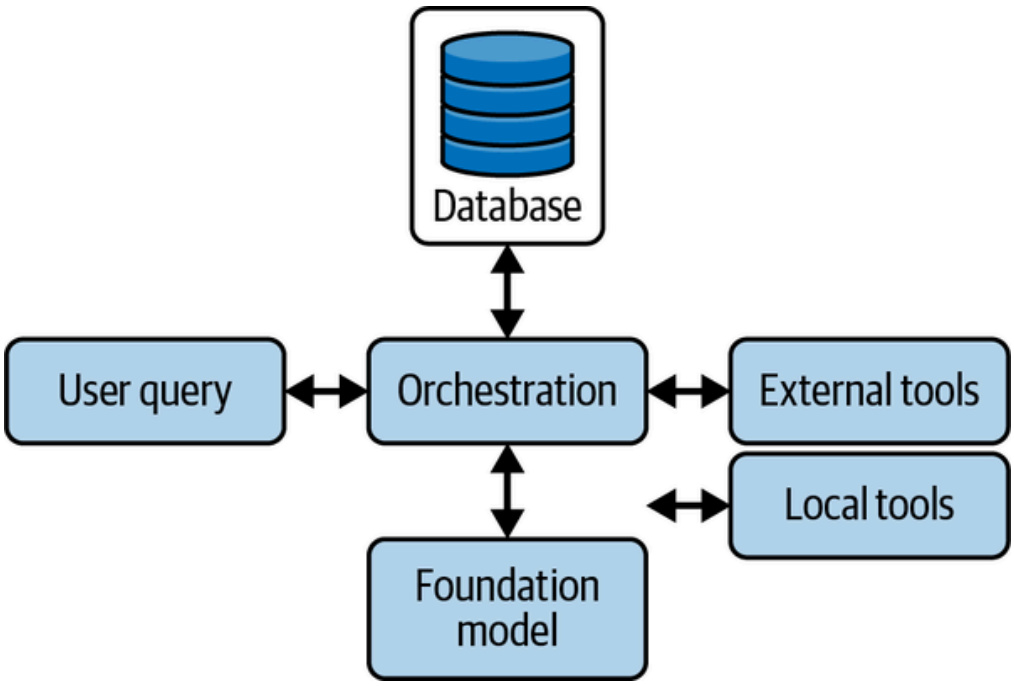

Figure 5-1. Orchestration as the core logic that handles user queries and coordinates calls to the foundation models, external and local tools, and to various databases to retrieve additional information.

Agent Types

Before diving into specific orchestration strategies, it's important to understand the different types of agents you can build. Each agent type embodies a distinct approach to reasoning, planning, and action, shaping how

tasks are decomposed and executed. Some agents respond instantly with preprogrammed mappings, while others iteratively reason and reflect to handle complex, open-ended goals. The choice of agent type directly influences your system's performance, cost, and capabilities. In this section, we will explore the spectrum: from reflex agents that provide lightning-fast responses, to deep research agents that tackle multistage investigations with adaptive plans and synthesis. Understanding these archetypes will help design agents aligned with your application needs and constraints and will illuminate how orchestration patterns, tool selection, and context construction come together within each type to achieve effective, reliable outcomes.

Reflex Agents

Reflex agents implement a direct mapping from input to action without any internal reasoning trace. Simple reflex agents follow "if-condition, then-action" rules, calling the appropriate tool immediately upon detecting predefined triggers. Because they bypass intermediate thought steps, reflex agents deliver responses with minimal latency and predictable performance, making them well suited for use cases like keyword-based routing, single-step data lookups, or basic automations (e.g., "If X, call tool Y"). However, their limited expressiveness means they cannot handle tasks requiring multistep reasoning or context beyond the immediate input.

ReAct Agents

ReAct agents interleave Reasoning and Action in an iterative loop: the model generates a *thought*, selects and invokes a tool, observes the result, and repeats as needed. This pattern enables the agent to break complex tasks into manageable steps, updating its plan based on intermediate observations:

- `ZERO_SHOT_REACT_DESCRIPTION` (LangChain) presents tools and instructions in a single prompt, relying on the LLM's innate reasoning to select and call tools without example traces.
- `CHAT_ZERO_SHOT_REACT_DESCRIPTION` extends this by incorporating conversational history, enabling the agent to use past exchanges when deciding on its next action.

ReAct agents excel in exploratory scenarios—dynamic data analysis, multisource aggregation, or troubleshooting—where the ability to adapt

midstream outweighs the additional latency and computational overhead. Their looped structure also provides transparency ("chain of thought") that aids debugging and auditability, though it can increase API costs and response times.

Planner-Executor Agents

Planner-executor agents split a task into two distinct phases: planning, where the model generates a multistep plan; and execution, where each planned step is carried out via tool calls. This clear separation lets the planner focus on long-horizon reasoning while executors invoke only the necessary tools, reducing redundant LLM calls. Because the plan is explicit, debugging and monitoring become straightforward—you can inspect the generated plan, track which step failed, and replan if needed. This approach has multiple advantages:

Clear decomposition

Complex tasks break down into manageable subtasks.

Debuggability

Explicit plans reveal where and why errors occur.

Cost efficiency

Smaller models or fewer LLM calls handle execution, reserving large models for planning.

Query-Decomposition Agents

Query-decomposition agents tackle a complex question by iteratively breaking it into subquestions, invoking search or other tools for each, and then synthesizing a final answer. This pattern—often called "self-ask with search"—prompts the model: "What follow-up question do I need?" → call search → "What's the next question?" → … → "What's the final answer?"

Example: `SELF_ASK_WITH_SEARCH`

Ask: "Who lived longer, X or Y?"

Self-ask: "What's X's lifespan?" → search tool

Self-ask: “What’s Y’s lifespan?” → search tool

Synthesize: “X lived 85 years, Y lived 90 years, so Y lived longer”

This approach excels when external knowledge retrieval is needed, ensuring each fact is grounded in tool output before composing the final response.

Reflection Agents

Reflection and metareasoning agents extend the ReAct paradigm by not only interleaving thought and action but also reviewing past steps to identify and correct mistakes before proceeding. In this approach—exemplified by the recently proposed ReflAct framework—the agent continuously grounds its reasoning in goal-state reflections, measuring its current state against the intended outcome and adjusting its plan when misalignments arise. Reflection prompts encourage the model to critique its own chain of thought, correct logical errors, and reinforce successful strategies, effectively simulating human-style self-assessment during complex problem-solving.

This pattern shines in high-stakes workflows where early errors can cascade into costly failures—such as financial transaction orchestration, medical diagnosis support, or critical incident response. By pairing each action with a reflection step, agents detect when tool outputs deviate from expectations and can replan or roll back before committing to irreversible operations. The added metareasoning overhead does incur extra latency and compute, but for tasks where correctness and reliability outweigh speed, reflection agents offer a powerful guardrail against error propagation and help maintain alignment with overarching goals.

Deep Research Agents

Deep research agents specialize in tackling open-ended, highly complex investigations that require extensive external knowledge gathering, hypothesis testing, and synthesis—think literature reviews, scientific discovery, or strategic market analysis. They combine multiple patterns: a planner-executor phase to chart research workflows; query-decomposition to break down big questions into targeted searches; and ReAct loops to iteratively refine hypotheses based on new findings. In a typical cycle, a deep research agent will:

1. Plan the overall research agenda (e.g., identify key subtopics or data sources).
2. Decompose each subtopic into concrete queries (via `SELF_ASK` or similar).
3. Invoke tools—from academic search APIs to domain-specific databases—and reflect on the relevance and reliability of each result.
4. Synthesize the insights into an evolving report or set of recommendations, using LLM-driven summarization and critique at each step.

Strengths

Capability

It can handle high-complexity, multistage investigations that lean on specialized databases and cross-disciplinary sources.

Adaptive

Research direction is adjusted as new evidence emerges.

Transparent

Explicit plans and decomposition steps make it easier to audit methodology.

Weaknesses

High cost

Extensive foundation model use and multiple API calls inflate compute and token expenses.

Latency

Each layer of planning, decomposition, and reflection adds delay.

Fragility

It is reliant on quality and availability of external data sources and needs careful error handling and fallback strategies.

The best use cases are long-form, expert-level tasks—academic literature surveys, technical due diligence, competitive intelligence—where depth and rigor trump speed.

Table 5-1 offers a snapshot of today's most common agent archetypes—each with its own trade-offs in speed, flexibility, and complexity. However, this landscape is evolving at breakneck speed. New hybrid patterns, metareasoning frameworks, and planning strategies are emerging all the time, and the classification of agent types will only grow more nuanced. Consider this list a starting point rather than a definitive taxonomy: as the field advances, you'll see fresh approaches built on these foundations, so stay curious, experiment often, and be ready to adapt your orchestration strategies as the research and tooling continue to mature.

Table 5-1. Common agent archetypes

Agent type	Strength	Weakness	Best use case
Reflex	Millisecond responses	No multistep reasoning	Keyword routing, simple lookups
ReAct	Flexible, on-the-fly adaptation	Higher latency and cost	Exploratory workflows, troubleshooting
Plan-execute	Clear task breakdown	Planning overhead	Complex, multistep processes
Query-decomposition	Grounded retrieval accuracy	Multiple tool calls	Research, fact-based Q&A
Reflection	Early error detection	Added compute and latency	High-stakes, safety-critical tasks
Deep research	Management of multistage, adaptive investigations	High compute costs and very high latency	Long-form literature reviews

Tool Selection

Before we get to orchestration, we will start with tool selection, because it is the foundation for more advanced planning. Different approaches to tool selection offer unique advantages and considerations, meeting different requirements and environments. We assume a set of tools have already been developed, so if you need a refresher, go back to Chapter 4.

Table 5-2. Tool selection strategies

Technique	Pros	Cons
Standard tool selection	Simple to implement	Scales poorly to high numbers of tools
Semantic tool selection	• Very scalable to large numbers of tools • Typically low latency to implement	Often worse selection accuracy due to semantic collisions
Hierarchical tool selection	Very scalable to large numbers of tools	Slower because it requires multiple sequential foundation model calls

Standard Tool Selection

The simplest approach is standard tool selection. In this case, the tool, its definition, and its description are provided to a foundation model, and the model is asked to select the most appropriate tool for the given context. The output from the foundation model is then compared with the toolset, and the closest one is chosen. This approach is easy to implement, and requires no additional training, embedding, or a toolset hierarchy to use. The main drawback is latency, as it requires another foundation model call, which can add seconds to the overall response time. It can also benefit from in-context learning, where few-shot examples can be provided to boost predictive accuracy for your problem without the challenge of training or fine-tuning a model.

Effective tool selection often comes down to how you describe each capability. Start by giving every tool a concise, descriptive name (e.g., `calculate_sum` instead of `process_numbers`) and follow it with a one-sentence summary that highlights its unique purpose (e.g., “Returns the sum of two numbers”). Include an example invocation in the description—showing typical inputs and outputs—to ground the model’s understanding in concrete terms rather than abstract language. Finally, enforce input constraints by specifying types and ranges (e.g., “x and y must be integers between 0 and

1,000”), which reduces ambiguous matches and helps the foundation model rule out irrelevant tools. By iteratively testing with representative prompts and refining each description for clarity and specificity, you’ll see significant gains in selection accuracy without any extra training or infrastructure. This sounds simple enough, but as the number of tools you register with your agent grows, overlap in the tool descriptions frequently becomes a problem and a source of mistakes in tool selection. Here we define another tool that is capable of computing mathematical expressions and evaluating formulas, something foundation models tend to not be good at:

```
from langchain_core.tools import tool
import requests

@tool
def query_wolfram_alpha(expression: str) -> str:
    """
    Query Wolfram Alpha to compute expressions or retri
    Args: expression (str): The mathematical expressior
    Returns: str: The result of the computation or the
        """

    api_url = f'''https://api.wolframalpha.com/v1/resul
        i={requests.utils.quote(expression)}&
        appid=YOUR_WOLFRAM_ALPHA_APP_ID'''

    try:
        response = requests.get(api_url)
        if response.status_code == 200:
            return response.text
        else: raise ValueError(f"Wolfram Alpha API Erro
            {response.status_code} - {response.text}")
    except requests.exceptions.RequestException as e:
        raise ValueError(f"Failed to query Wolfram Alph


@tool
def trigger_zapier_webhook(zap_id: str, payload: dict)
    """ Trigger a Zapier webhook to execute a predefine
    Args:
    zap_id (str): The unique identifier for the Zap to
    payload (dict): The data to send to the Zapier webh
    Returns:
    str: Confirmation message upon successful triggerir
    Raises: ValueError: If the API request fails or ret
```

```
        """

        zapier_webhook_url = f"https://hooks.zapier.com/hoc
        try:
            response = requests.post(zapier_webhook_url, js
            if response.status_code == 200:
                return f"Zapier webhook '{zap_id}' successf

            else:
            raise ValueError(f'''Zapier API Error: {respons
                             {response.text}''')
        except requests.exceptions.RequestException as e:
            raise ValueError(f"Failed to trigger Zapier web
```

Here's another example of a tool you might want to register with your agent to notify a particular channel when your task is completed or needs attention for a human-in-the-loop pattern:

```
@tool
def send_slack_message(channel: str, message: str) -> s
    """ Send a message to a specified Slack channel.
    Args:
    channel (str): The Slack channel ID or name where t
    message (str): The content of the message to send.
    Returns:
    str: Confirmation message upon successful sending c
    Raises: ValueError: If the API request fails or ret
    """

    api_url = "https://slack.com/api/chat.postMessage"
        headers = { "Authorization": "Bearer YOUR_SLACK
                    "Content-Type": "application/json"
    payload = { "channel": channel, "text": message }
    try:
        response = requests.post(api_url, headers=heade
        response_data = response.json()
        if response.status_code == 200 and response_dat
            return f"Message successfully sent to Slack
        else:
            error_msg = response_data.get("error", "Unk
            raise ValueError(f"Slack API Error: {error_
        except requests.exceptions.RequestException as
            raise ValueError(f'''Failed to send message
```

```
                                "{channel}": {e}''')
```

Now that we've defined our tools, we bind them to the model client and allow the model to pick which tools to invoke to best address the input:

```
# Initialize the LLM with GPT-4o and bind the tools
llm = ChatOpenAI(model_name="gpt-4o")
llm_with_tools = llm.bind_tools([get_stock_price,
    send_slack_message, query_wolfram_alpha])

messages = [HumanMessage("What is the stock price of Ap

ai_msg = llm_with_tools.invoke(messages)
messages.append(ai_msg)

for tool_call in ai_msg.tool_calls:
   tool_msg = get_stock_price.invoke(tool_call)

final_response = llm_with_tools.invoke(messages)
print(final_response.content)
```

In summary, standard tool selection offers a fast, intuitive way to integrate tools into your agent system without additional infrastructure or training overhead. While it scales well for small toolsets, careful description engineering becomes essential as your tool library grows to maintain accuracy and avoid misselection. By combining thoughtful descriptions with iterative prompt testing, you can achieve robust performance using this simple yet powerful approach.

Semantic Tool Selection

Another approach, semantic tool selection, uses semantic representations to index all of the available tools and semantic search to retrieve the most relevant tools. This reduces the number of tools to choose from and then relies on the foundation model to choose the correct tool and parameters from this much smaller set. Ahead of time, each tool definition and description is embedded using an encoder-only model—such as OpenAI's Ada model, Amazon's Titan model, Cohere's Embed model, ModernBERT, or others—which represents the tool name and description as a vector of numbers. This

process is illustrated in Figure 5-2, which shows how each tool is embedded into a vector representation for efficient retrieval based on semantic similarity to the task query.

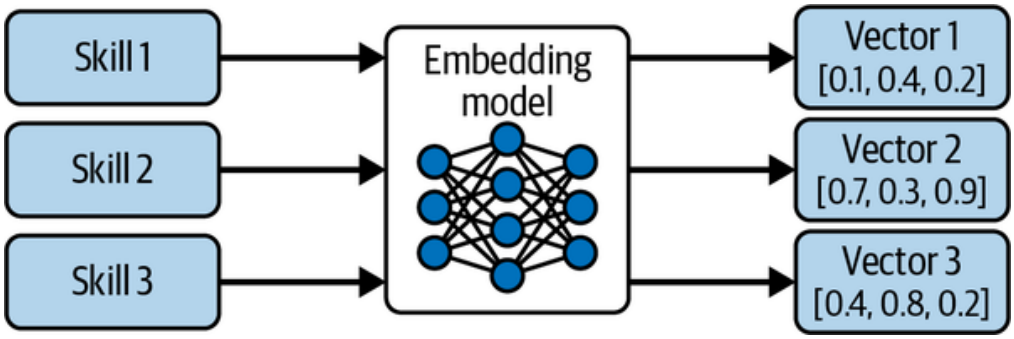

Figure 5-2. Semantic tool embedding for retrieval-based selection. Each tool or skill is encoded into a dense vector representation using an embedding model. These vectors are then stored for efficient semantic search, enabling the system to retrieve the most relevant tools based on the task query.

These tools are then indexed in a lightweight vector database. At runtime, the current context is embedded using the same embedding model, a search is performed on the database, and the top tools are selected and retrieved. These tools are then passed to the foundation model, which can then choose to invoke a tool and choose the parameters. The tool is then invoked, and the response is used to compose the response for the user. This process is illustrated in Figure 5-3, which shows how the system retrieves relevant tools and uses the foundation model to select and invoke the appropriate tool with its parameters to generate the final response.

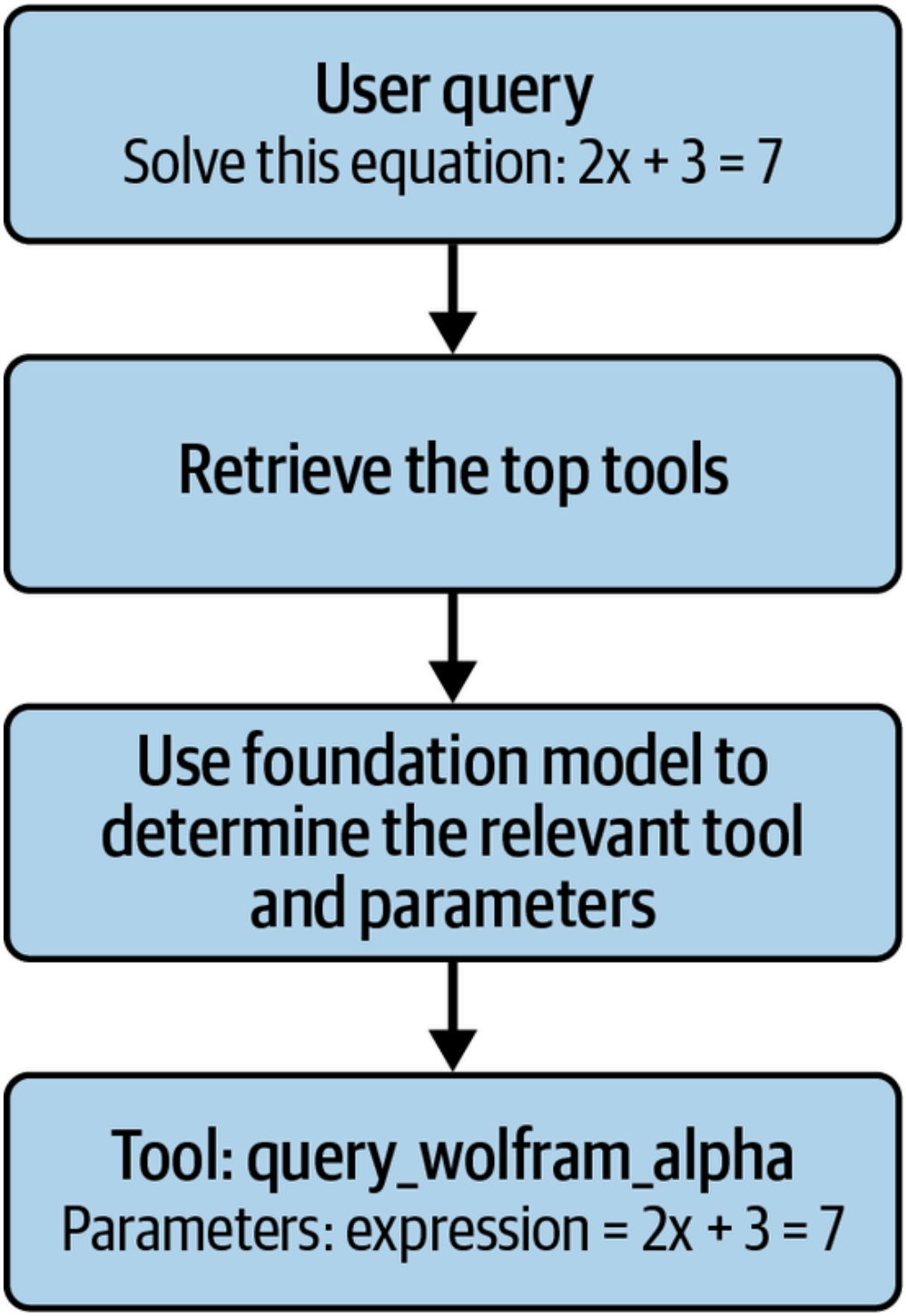

Figure 5-3. Semantic tool retrieval and invocation workflow. At runtime, the user query is embedded and used to retrieve the top relevant tools from the vector database. The foundation model then selects the appropriate tool and determines its parameters, invokes the tool, and integrates the tool's output to generate the final user response.

This is the most common pattern and is recommended for most use cases. It's typically faster than standard tool selection, performant, and reasonably scalable. First, the tool database is set up by embedding the tool descriptions:

```
import os
import requests
import logging
from langchain_core.tools import tool
from langchain_openai import ChatOpenAI, OpenAIEmbeddir
from langchain_core.messages import HumanMessage, AIMes
```

```
from langchain.vectorstores import FAISS
import faiss
import numpy as np

# Initialize OpenAI embeddings
embeddings = OpenAIEmbeddings(openai_api_key=OPENAI_API

# Tool descriptions
tool_descriptions = {
       "query_wolfram_alpha": '''Use Wolfram Alpha to c
                                expressions or retriev
       "trigger_zapier_webhook": '''Trigger a Zapier we
                                   predefined automate
       "send_slack_message": '''Send messages to specif
                                communicate with team m
}

# Create embeddings for each tool description
tool_embeddings = []
tool_names = []

for tool_name, description in tool_descriptions.items()
   embedding = embeddings.embed_text(description)
   tool_embeddings.append(embedding)
   tool_names.append(tool_name)

# Initialize FAISS vector store
dimension = len(tool_embeddings[0])
index = faiss.IndexFlatL2(dimension)

# Normalize embeddings for cosine similarity
faiss.normalize_L2(np.array(tool_embeddings).astype('fl

# Convert list to FAISS-compatible format
tool_embeddings_np = np.array(tool_embeddings).astype('
index.add(tool_embeddings_np)

# Map index to tool functions
index_to_tool = {
   0: query_wolfram_alpha,
   1: trigger_zapier_webhook,
   2: send_slack_message
}
```

Those embeddings for your tool catalog only need to be computed once, and now they're ready to be quickly retrieved. To choose your tool, you embed your query using the same embedding model, perform a quick database lookup, choose the parameters, and invoke our tool:

```
def select_tool(query: str, top_k: int = 1) -> list:
    """
    Select the most relevant tool(s) based on the user's
    vector-based retrieval.

    Args:
        query (str): The user's input query.
        top_k (int): Number of top tools to retrieve.

    Returns:
        list: List of selected tool functions.
    """
    query_embedding = embeddings.embed_text(query).astyp
    faiss.normalize_L2(query_embedding.reshape(1, -1))
    D, I = index.search(query_embedding.reshape(1, -1),
    selected_tools = [index_to_tool[idx] for idx in I[0]
    return selected_tools

def determine_parameters(query: str, tool_name: str) ->
    """
    Use the LLM to analyze the query and determine the p
    to be invoked.

    Args:
        query (str): The user's input query.
        tool_name (str): The selected tool name.

    Returns:
        dict: Parameters for the tool.
    """
    messages = [
        HumanMessage(content=f'''Based on the user's que
            parameters should be used for the tool '{toc
    ]

    # Call the LLM to extract parameters
    response = llm(messages)

    # Example logic to parse response from LLM
    parameters = {}
```

```
    if tool_name == "query_wolfram_alpha":
        parameters["expression"] = response['expression'
        # Extract mathematical expression
    elif tool_name == "trigger_zapier_webhook":
        parameters["zap_id"] = response.get('zap_id', "1
        parameters["payload"] = response.get('payload',
    elif tool_name == "send_slack_message":
        parameters["channel"] = response.get('channel',
        parameters["message"] = response.get('message',

    return parameters

# Example user query
user_query = "Solve this equation: 2x + 3 = 7"

# Select the top tool
selected_tools = select_tool(user_query, top_k=1)
tool_name = selected_tools[0] if selected_tools else Nc

if tool_name:
    # Use LLM to determine the parameters based on the c
    args = determine_parameters(user_query, tool_name)

    # Invoke the selected tool
    try:
        # Assuming each tool has an `invoke` method to e
        tool_result = globals()[tool_name].invoke(args)
        print(f"Tool '{tool_name}' Result: {tool_result}
    except ValueError as e:
        print(f"Error invoking tool '{tool_name}': {e}")
else:
    print("No tool was selected.")
```

Hierarchical Tool Selection

If your scenario involves a large number of tools, however, you might need to consider hierarchical tool selection. This is especially true if many of those tools are semantically similar and you are looking to improve tool selection accuracy at the price of higher latency and complexity. In this pattern, you organize your tools into groups and provide a description for each group. Your tool selection (either generative or semantic) first selects a group and then performs a secondary search only among the tools in that group. Figure 5-4

visualizes this two-stage process, showing how a query is first routed to the appropriate tool group and then refined to a single tool within that group.

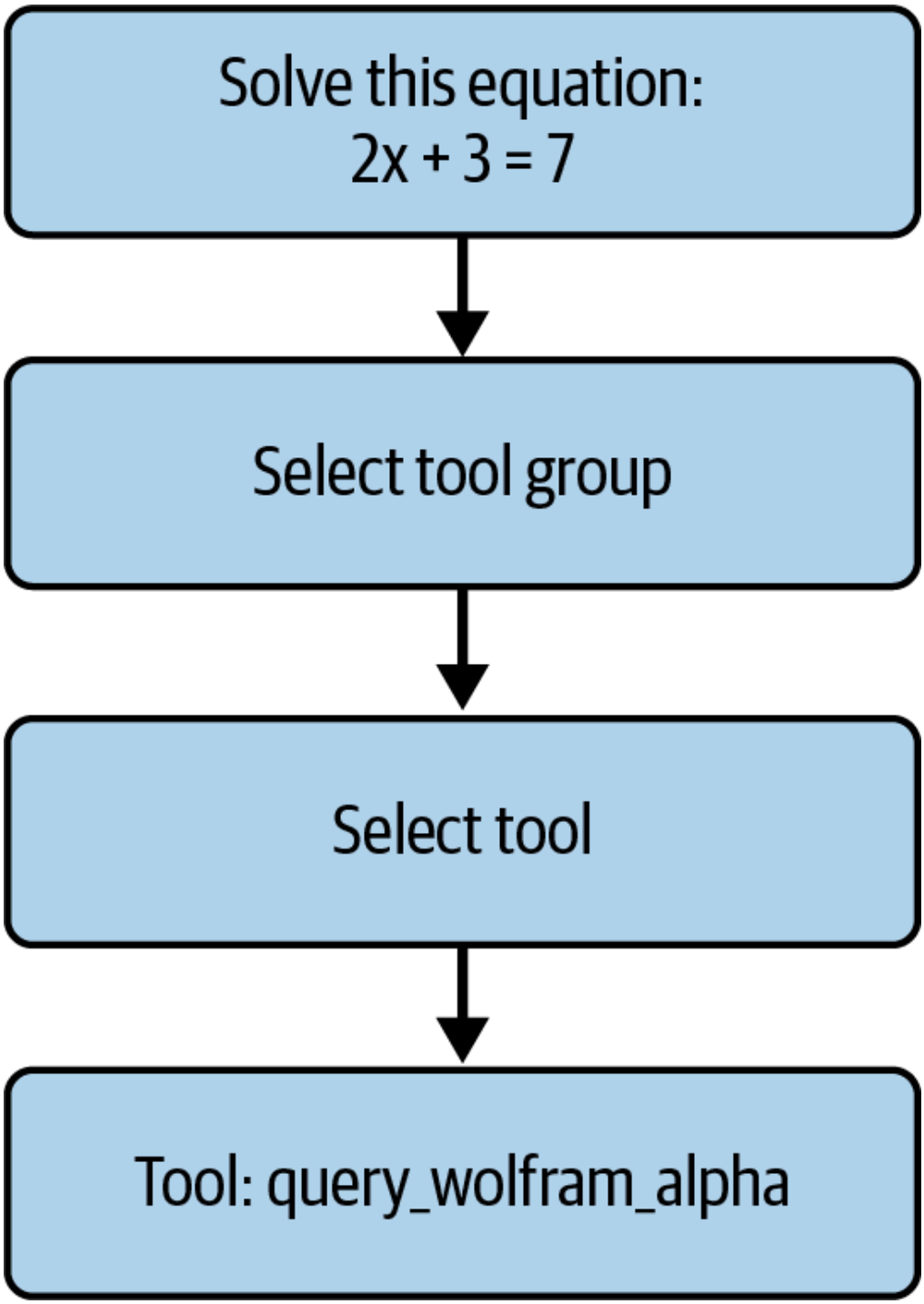

Figure 5-4. Hierarchical tool-selection workflow. The agent first chooses the most relevant tool group for the query, and then narrows the search to select a single tool within that group—in this example, routing a math question through the tool group and ultimately invoking `query_wolfram_alpha`.

While this is slower and would be expensive to parallelize, it reduces the complexity of the tool selection task into two smaller chunks, and frequently results in higher overall tool selection accuracy. Crafting and maintaining these tool groups takes time and effort, so this is not recommended unless you have a large number of tools:

```
import os
import requests
```

```python
import logging
import numpy as np
from langchain_core.tools import tool
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage, AIMes

# Initialize the LLM
llm = ChatOpenAI(model_name="gpt-4", temperature=0)

# Define tool groups with descriptions
tool_groups = {
    "Computation": {
        "description": '''Tools related to mathematical
                          data analysis.''',
        "tools": []
    },
    "Automation": {
        "description": '''Tools that automate workflows
                          different services.''',
        "tools": []
    },
    "Communication": {
        "description": "Tools that facilitate communica
        "tools": []
    }
}

# Define Tools
@tool
def query_wolfram_alpha(expression: str) -> str:
    api_url = f'''https://api.wolframalpha.com/v1/resul
    {requests.utils.quote(expression)}&appid={WOLFRAM_A
    try:
        response = requests.get(api_url)
        if response.status_code == 200:
            return response.text
        else:
            raise ValueError(f'''Wolfram Alpha API Erro
                             - {response.text}''')
    except requests.exceptions.RequestException as e:
        raise ValueError(f"Failed to query Wolfram Alph

@tool
def trigger_zapier_webhook(zap_id: str, payload: dict)
    zapier_webhook_url = f"https://hooks.zapier.com/hoo
    try:
```

```
        response = requests.post(zapier_webhook_url, js
        if response.status_code == 200:
            return f"Zapier webhook '{zap_id}' successf
        else:
            raise ValueError(f'''Zapier API Error: {res
                                {response.text}''')
    except requests.exceptions.RequestException as e:
        raise ValueError(f"Failed to trigger Zapier web

@tool
def send_slack_message(channel: str, message: str) -> s
    api_url = "https://slack.com/api/chat.postMessage"
    headers = {
        "Authorization": f"Bearer {SLACK_BOT_TOKEN}",
        "Content-Type": "application/json"
    }
    payload = {
        "channel": channel,
        "text": message
    }
    try:
        response = requests.post(api_url, headers=heade
        response_data = response.json()
        if response.status_code == 200 and response_dat
            return f"Message successfully sent to Slack
        else:
            error_msg = response_data.get("error", "Unk
            raise ValueError(f"Slack API Error: {error_
    except requests.exceptions.RequestException as e:
        raise ValueError(f'''Failed to send message to
                         '{channel}': {e}''')

# Assign tools to their respective groups
tool_groups["Computation"]["tools"].append(query_wolfra
tool_groups["Automation"]["tools"].append(trigger_zapie
tool_groups["Communication"]["tools"].append(send_slack

# ------------------------------
# LLM-Based Hierarchical Tool Selection
# ------------------------------
def select_group_llm(query: str) -> str:
    """
    Use the LLM to determine the most appropriate tool
    user's query.

    Args:
```

```
        query (str): The user's input query.

    Returns:
        str: The name of the selected group.
    """
    prompt = f'''Select the most appropriate tool group
        '{query}'.\nOptions are: Computation, Automatic
    response = llm([HumanMessage(content=prompt)])
    return response.content.strip()

def select_tool_llm(query: str, group_name: str) -> str
    """
    Use the LLM to determine the most appropriate tool
    on the user's query.

    Args:
        query (str): The user's input query.
        group_name (str): The name of the selected tool

    Returns:
        str: The name of the selected tool function.
    """
    prompt = f'''Based on the query: '{query}', select
                tool from the group '{group_name}'.'''
    response = llm([HumanMessage(content=prompt)])
    return response.content.strip()

# Example user query
user_query = "Solve this equation: 2x + 3 = 7"

# Step 1: Select the most relevant tool group using LLM
selected_group_name = select_group_llm(user_query)
if not selected_group_name:
    print("No relevant tool group found for your query.
else:
    logging.info(f"Selected Group: {selected_group_name
    print(f"Selected Tool Group: {selected_group_name}"

    # Step 2: Select the most relevant tool within the
    selected_tool_name = select_tool_llm(user_query, se
    selected_tool = globals().get(selected_tool_name, N

    if not selected_tool:
        print("No relevant tool found within the select
    else:
        logging.info(f"Selected Tool: {selected_tool.__
```

```
print(f"Selected Tool: {selected_tool.__name__}

# Prepare arguments based on the tool
args = {}
if selected_tool == query_wolfram_alpha:
    # Assume the entire query is the expressior
    args["expression"] = user_query
elif selected_tool == trigger_zapier_webhook:
    # Use placeholders for demo
    args["zap_id"] = "123456"
    args["payload"] = {"message": user_query}
elif selected_tool == send_slack_message:
    # Use placeholders for demo
    args["channel"] = "#general"
    args["message"] = user_query
else:
    print("Selected tool is not recognized.")

# Invoke the selected tool
try:
    tool_result = selected_tool.invoke(args)
    print(f"Tool '{selected_tool.__name__}' Res
except ValueError as e:
    print(f"Error: {e}")
```

Tool Execution

Parametrization is the process of defining and setting the parameters that will guide the execution of a tool in a language model. This process is crucial, as it determines how the model interprets the task and tailors its response to meet the specific requirements. Parameters are defined by the tool definition, as discussed in more detail in Chapter 4. The current state of the agent, including progress so far, is included as additional context in the prompt window, and the foundation model is instructed to fill the parameters with appropriate data types to match the expected inputs for the function call. Additional context, such as the current time or the user's location, can be injected into the context window to provide additional guidance for functions that require this type of information. It is recommended to use a basic parser to validate that the inputs meet the basic criteria for the data types, and to instruct the foundation model to correct the pattern if it does not pass this check.

Once the parameters are set, the tool execution phase begins. Some of these tools can easily be executed locally, while others will be executed remotely by API. During execution, the model might interact with various APIs, databases, or other tools to gather information, perform calculations, or execute actions that are necessary to complete the task. The integration of external data sources and tools can significantly enhance the utility and accuracy of the agent's outputs. Timeout and retry logic will need to be adjusted to the latency and performance requirements for the use case.

Tool Topologies

Today, the majority of chatbot systems rely on single tool execution without planning. This makes sense: it is easier to implement, and has lower latency. If your team is developing its first agent-based system, or if that is sufficient to meet the needs for your scenario, then you can stop there after the following section, "Single Tool Execution." For many cases, however, we want our agents to be able to perform complex tasks that require multiple tools. By providing an agent with a sufficient range of tools, you can then enable your agent to flexibly arrange those tools and apply them in correct order to solve a wider variety of problems. In traditional software engineering, the designers had to implement the exact control flow and order in which steps should be taken. Now, we can implement the tools and define the tools topology in which the agent can operate, and then allow the exact composition to be designed dynamically in response to the context and task at hand. This section considers this range of tool topologies and discusses their trade-offs.

Single Tool Execution

We'll begin with tasks that require precisely one tool. In this case, planning consists of choosing the one tool most appropriate to address the task. Once the tool is selected, it must be correctly parameterized based on the tool definition. The tool is then executed, and its output is used as an input when composing the final response for the user, which can be seen in Figure 5-5. While this is a minimal definition of a plan, it is the foundation from which we will build more complex patterns.

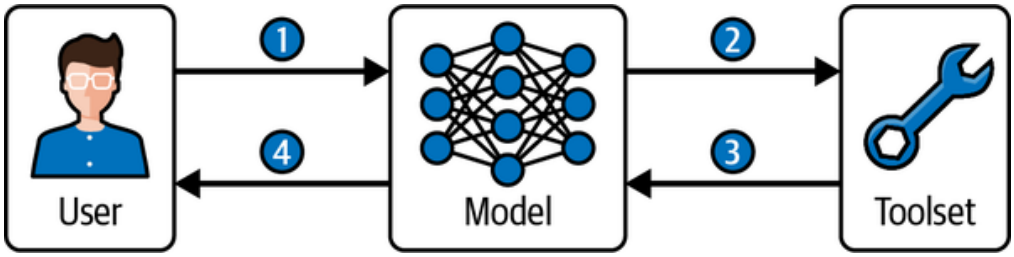

Figure 5-5. Single tool execution workflow. The user query is passed to the model (step 1), which selects the appropriate tool from the toolset (step 2), receives the tool output (step 3), and composes the final response for the user (step 4).

To make this example more concrete, Figure 5-6 shows this same single tool execution workflow where the agent retrieves and returns the current weather for New York City.

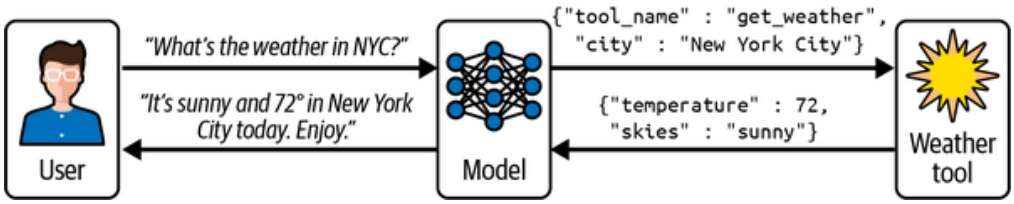

Figure 5-6. Example of single tool execution for weather retrieval. The user asks for the weather in New York City, the model selects and parameterizes the weather tool, retrieves the temperature and conditions as a JSON payload, and composes a natural language response using this information for the user.

While this single tool execution pattern is simple, it forms the foundation upon which more complex multistep planning and tool orchestration strategies are built in advanced agent systems. In the next section, we'll look at how we can execute more tools without sacrificing latency.

Parallel Tool Execution

The first increase in complexity comes with tool parallelism. In some cases, it might be worth taking multiple actions on the input. For example, imagine that you need to look up a record for a patient. If your toolset includes multiple tools that access multiple sources of data, then it will be necessary to execute multiple actions to retrieve data from each of the sources. This increases the complexity of the problem because it is unclear how many tools need to be executed. A common approach is to retrieve a maximum number of tools that might be executed—say, five—using semantic tool selection. Next, make a second call to a foundation model with each of these five tools, and ask it to select the five or fewer tools that are necessary to the problem, filtering down to the tools necessary for the task. Similarly, the foundation model can be called repeatedly with the additional context of which tools have already been selected until it chooses to add no more tools. Once selected, these tools are independently parameterized and executed. After all tools have

been completed, their results are passed to the foundation model to draft a final response for the user. Figure 5-7 illustrates this pattern.

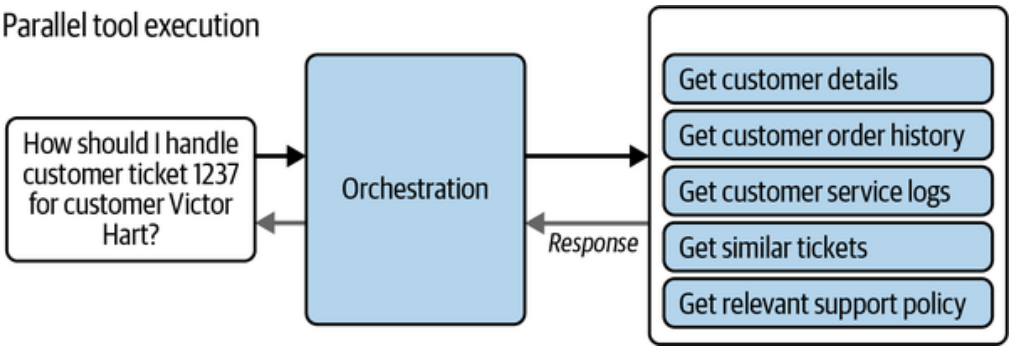

Figure 5-7. Parallel tool execution pattern. In this example, the user asks how to handle a customer ticket. The orchestration process selects multiple tools to run in parallel—such as retrieving customer details, order history, service logs, similar tickets, and relevant support policies—before integrating their outputs to generate the final response.

This pattern of parallel tool execution enables agents to efficiently gather comprehensive information from multiple sources in a single step. By integrating these results before composing a response, the agent can provide richer, more informed outputs while minimizing overall latency.

Chains

The next increase in complexity brings us to chains. Chains refer to sequences of actions that are executed one after another, with each action depending on the successful completion of the previous one. Planning chains involves determining the order in which actions should be performed to achieve a specific goal while ensuring that each action leads to the next without interruption. Chains are common in tasks that involve step-by-step processes or linear workflows.

Fortunately, LangChain offers a declarative syntax, the LangChain Expression Language (LCEL), to build chains by composing existing Runnables rather than manually wiring up `Chain` objects. Under the hood, LCEL treats every chain as a Runnable implementing the same interface, so you can `invoke()`, `batch()`, or `stream()` any LCEL chain just like any other Runnable:

```
from langchain_core.runnables import RunnableLambda
from langchain.chat_models import ChatOpenAI
from langchain_core.prompts import PromptTemplate
# Wrap a function or model call as a Runnable
llm = RunnableLambda.from_callable(ChatOpenAI(model_nam
```

```
                                        temperature=0).genera
prompt = RunnableLambda.from_callable(lambda text:
    PromptTemplate.from_template(text).format_prompt({'
                                        ).to_messages())
# Traditional chain equivalent:
# chain = LLMChain(prompt=prompt, llm=llm)
# LCEL chain using pipes:
chain = prompt | llm
# Invoke the chain
result = chain.invoke("What is the capital of France?")
```

By switching to LCEL, you reduce boilerplate, gain advanced execution features, and keep your chains concise and maintainable. Figure 5-8 illustrates the general agentic chain pattern that underlies many LCEL workflows.

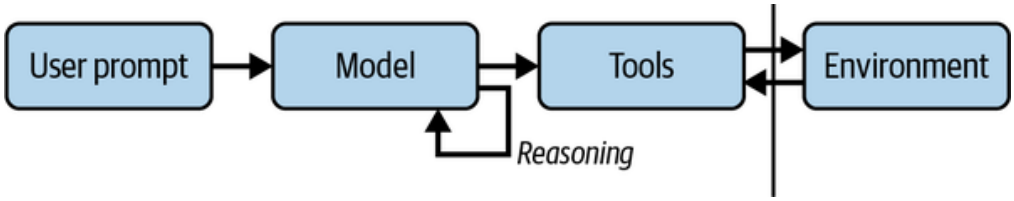

Figure 5-8. Agentic chain execution pattern. The user prompt is passed to the model, which performs reasoning and invokes tools to interact with the environment. The resulting observations are looped back into the model for further reasoning until the task is complete.

The planning of chains requires careful consideration of the dependencies between actions, aiming to orchestrate a coherent flow of activity toward the desired outcome. It is highly recommended that a maximum length be set to the tool chains, as errors can compound down the length of the chain. As long as the task is not expected to fan out to multiple branching subtasks, chains provide an excellent trade-off between adding planning for multiple tools with dependencies and keeping the complexity relatively low.

Graphs

For support scenarios with multiple decision points, a graph topology models complex, nonhierarchical flows far more expressively than chains or trees. Unlike linear chains or strictly branching trees, graph structures let you define both conditional edges *and* consolidation edges, so that parallel paths can merge back into shared nodes.

Each node in a graph represents a discrete tool invocation (or logical step), while edges—including `add_conditional_edges` —declare the exact conditions under which the agent may transition between steps. By

consolidating outputs from multiple branches into a single downstream node (e.g., `summarize_response`), you can stitch together findings from separate handlers into a unified customer reply.

However, full graph execution typically incurs significantly more foundation model calls than chains—adding latency and cost—so it's crucial to cap depth and branching factor. In addition, cycles, unreachable nodes, or conflicting state merges introduce new classes of errors that must be managed through rigorous validation and testing. The following is an example for how to implement a graph in LangGraph:

```
from langgraph.graph import StateGraph, START, END
from langchain.chat_models import ChatOpenAI

# Initialize LLM
llm = ChatOpenAI(model_name="gpt-4", temperature=0)
# 1. Node definitions
def categorize_issue(state: dict) -> dict:
    prompt = (
        f"Classify this support request as 'billing' or
        f"Message: {state['user_message']}"
    )
    generations = llm.generate([{"role":"user","content
    kind = generations[0][0].text.strip().lower()
    return {**state, "issue_type": kind}
def handle_invoice(state: dict) -> dict:
    # Fetch invoice details...
    return {**state, "step_result": f"Invoice details f
def handle_refund(state: dict) -> dict:
    # Initiate refund workflow...
    return {**state, "step_result": "Refund process ini
def handle_login(state: dict) -> dict:
    # Troubleshoot login...
    return {**state, "step_result": "Password reset lir
def handle_performance(state: dict) -> dict:
    # Check performance metrics...
    return {**state, "step_result": "Performance metric
def summarize_response(state: dict) -> dict:
    # Consolidate previous step_result into a user-faci
    details = state.get("step_result", "")
    summary = llm.generate([{"role":"user","content":
        f"Write a concise customer reply based on: {det
    }]).generations[0][0].text.strip()
```

```
    return {**state, "response": summary}
```

This next section wires up the logical flow in each node into an actual execution graph. By creating a new `StateGraph`, we establish the starting point with START → `categorize_issue`, which ensures every request first passes through the classification step. Then, using `add_conditional_edges`, you encode the core business rules: after categorization, only billing issues route into the invoice/refund handlers, and only technical issues route into the login/performance handlers. Each router function inspects the evolving state and returns the name of the next node, and the mapping ensures that only valid successors are enabled at runtime. This approach keeps the decision logic explicit, enforces the correct sequence of tool invocations, and prevents invalid transitions—all before we ever execute a single tool call:

```
# 2. Build the graph
graph = StateGraph()
# Start → categorize_issue
graph.add_edge(START, categorize_issue)
# categorize_issue → billing or technical
def top_router(state):
    return "billing" if state["issue_type"] == "billing
graph.add_conditional_edges(
    categorize_issue,
    top_router,
    mapping={"billing": handle_invoice, "technical": ha
)
# Billing sub-branches: invoice vs. refund
def billing_router(state):
    msg = state["user_message"].lower()
    return "invoice" if "invoice" in msg else "refund"
graph.add_conditional_edges(
    handle_invoice,
    billing_router,
    mapping={"invoice": handle_invoice, "refund": handl
)
# Technical sub-branches: login vs. performance
def tech_router(state):
    msg = state["user_message"].lower()
    return "login" if "login" in msg else "performance'
graph.add_conditional_edges(
    handle_login,
```

```
        tech_router,
        mapping={"login": handle_login, "performance": hanc
    )
```

This final wiring adds consolidation edges so that, no matter which subpath was taken—whether the user needed an invoice lookup, a refund, login troubleshooting, or performance checks—their result feeds into the single `summarize_response` node. By connecting each of the handler nodes (`handle_refund`, `handle_performance`, `handle_invoice`, and `handle_login`) into `summarize_response`, you ensure all divergent outcomes are unified into one coherent customer reply. Finally, linking `summarize_response` to `END` cleanly terminates the workflow, guaranteeing every execution path converges on a polished response before the graph finishes:

```
# Consolidation: both refund and performance (and invoi
graph.add_edge(handle_refund, summarize_response)
graph.add_edge(handle_performance, summarize_response)
# Also cover paths where invoice or login directly go t
graph.add_edge(handle_invoice, summarize_response)
graph.add_edge(handle_login, summarize_response)
# Final: summary → END
graph.add_edge(summarize_response, END)
# 3. Execute the graph
initial_state = {
    "user_message": "Hi, I need help with my invoice ar
    "user_id": "U1234"
}
result = graph.run(initial_state, max_depth=5)
print(result["response"])
```

Graphs offer the ultimate flexibility for modeling complex, nonlinear workflows—enabling you to branch, merge, and consolidate multiple tool executions into a unified process. However, this expressiveness comes with added overhead: more LLM calls, deeper routing logic, and the potential for cycles or unreachable paths. To harness graphs effectively, always anchor your design in your specific use case's requirements, and resist the temptation to overcomplicate.

Start with a chain if your task is strictly linear (e.g., prompt → model → parser). Chains are easy to reason about and debug. Adopt a graph only when you must both branch and later consolidate multiple streams of information (e.g., parallel analysis steps that feed a single summary).

In practice, sketch your topology on paper first: label each node with the tool or logical step, draw arrows for the allowed transitions, and highlight where branches reunite. Then implement incrementally—cap your depth and branching factor, write unit tests for each router, and leverage LangGraph's built-in tracing to validate that every path leads to a terminal node.

Above all, keep it as simple as possible. Every additional node or edge multiplies the potential execution paths and error modes. If a simpler chain or tree meets your needs, save the graph patterns for genuinely complex scenarios. By starting simple and iterating only as your requirements demand, you'll build robust, maintainable orchestration that scales with confidence.

Context Engineering

Context engineering is a core component of orchestration. It ensures that each step in an agent's plan has the right information and instructions to perform effectively. While prompt engineering focuses on writing effective instructions, context engineering involves dynamically assembling all inputs —user messages, retrieved knowledge, workflow state, and system prompts— into a structured, token-efficient context window that maximizes task performance. For example, planner-executor agents depend on clean plan outputs being passed as context to executor steps, while ReAct agents require relevant tool results embedded clearly in the prompt to inform their next reasoning cycle. Context engineering thus bridges planning and execution, enabling agent workflows to remain coherent, grounded, and aligned with user goals.

At its core, context engineering involves deciding what information to include, how to structure it for maximum clarity and relevance, and how to fit it efficiently within token limits. This includes the current user input, relevant snippets retrieved from memory or external knowledge bases, summaries of prior conversations, system instructions defining the agent's role, and any workflow state necessary for the task at hand. In simple systems, context may consist only of a system prompt and the latest user query. But as agents tackle

more complex tasks—like orchestrating multistep workflows or personalizing recommendations based on past interactions—dynamic context construction becomes critical for maintaining coherence, accuracy, and utility.

For example, an agent handling ecommerce support might construct its context by combining the system prompt defining its allowed actions, the user's current message, a retrieved summary of the order record, and any applicable policy excerpts. In more advanced systems, the context might also include summaries of prior related conversations or the results of tool invocations from earlier in the workflow. Each additional element can improve task performance, but only if included thoughtfully; irrelevant or poorly structured context risks distracting the model or exceeding token budgets without benefit.

Effective context engineering requires several core practices. First, prioritize relevance by retrieving only the most useful information from memory or knowledge bases, rather than indiscriminately appending large blocks of text. Second, maintain clarity through structured formatting or schemas such as Model Context Protocol (MCP), which pass state and retrieved knowledge to the model in a predictable, interpretable way. Third, use summarization techniques to compress longer histories into concise representations, preserving critical details without wasting tokens. Finally, ensure that context is dynamically assembled at each inference step to reflect the agent's current objectives, workflow stage, and user input.

Context engineering sits at the intersection of memory, knowledge, and orchestration. While orchestration decides what steps to take in a workflow, context engineering ensures that each step has the right information to execute effectively. As foundation models continue to improve, the frontier of agentic system design is shifting from model architecture to the quality of context we provide. In essence, a well-engineered context unlocks the full potential of even modest models, while poor context can undermine the performance of the most advanced systems.

By mastering context engineering, developers can create agents that are not only technically powerful but also reliable, grounded, and responsive to the needs of their users and environments. In the coming years, as memory systems, retrieval architectures, and orchestration frameworks evolve, context engineering will remain the glue that binds these components into seamless, effective experiences.

Conclusion

The success of agents relies heavily on the approach to orchestration, making it important for organizations interested in building agentic systems to invest time and energy into designing the appropriate planning strategy for the use case. Here are some best practices for designing a planning system:

- Carefully consider the requirements for latency and accuracy for your system, as there is a clear trade-off between these two factors.
- Determine the typical number of actions required for your scenario's use case. The greater this number, the more complex an approach to planning you are likely to need.
- Assess how much the plan needs to change based on the results from prior actions. If significant adaptation is necessary, consider a technique that allows for incremental plan adjustments.
- Design a representative set of test cases to evaluate different planning approaches and identify the best fit for your use case.
- Choose the simplest planning approach that will meet your use case requirements.

With an orchestration approach that will work well for your scenario, we'll now move on to the next part of the workflow: memory. It is worth starting small with well-designed scenarios and simpler approaches to orchestration, and to then gradually move up the scale of complexity as necessary based on the use case. In the next chapter, we will explore how memory can further enhance your agents' capabilities—enabling them to recall knowledge, maintain context across interactions, and perform tasks with greater intelligence and personalization.