Chapter 5. Orchestration

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the fifth chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *sevans@oreilly.com*.

Now that your agent has a set of skills that can be used, it's time to start using them to solve real tasks. While simple tasks can be handled by relying on the intrinsic information contained within the models' weights, and some tasks can be handled with a single skill, more complex tasks will often require multiple skills, with the potential for dependencies between these skills. To handle more complex tasks, agents need to perform more than one action and do so in a reasonable order, appropriately managing dependencies between the tasks. This chapter will cover orchestration, including skill selection, execution, skill topologies, and planning.

Skill Selection

Before we get to planning, we will consider the critical task of skill selection, because it is the foundation for more advanced planning. Different approaches to skill selection offer unique advantages and considerations, meeting different requirements and environments. We assume a set of skills have already been developed, so if you need a refresher, go back to Chapter 3.

Generative Skill Selection

The simplest approach is Generative Skill Selection. In this case, the skill, its definition, and its description are provided to a foundation model, and the model is asked to select the most appropriate skill for the given context. The output from the foundation model is then compared to the skillset, and the closest one is chosen. This approach is easy to implement, and requires no additional training, embedding, or a skillset 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.

```
from langchain_core.tools import tool
import requests

@tool def query_wolfram_alpha(expression: str) -> str:
    """ Query Wolfram Alpha to compute mathematical expressions or retrieve information.
    Args: expression (str): The mathematical expression or query to evaluate.
    Returns: str: The result of the computation or the retrieved information.
    api_url = f"https://api.wolframalpha.com/v1/result?i={requests.utils.quote(expression)}&appid=YOUR_W

@tool def trigger_zapier_webhook(zap_id: str, payload: dict) -> str:
    """ Trigger a Zapier webhook to execute a predefined Zap. Args:
    zap_id (str): The unique identifier for the Zap to be triggered.
    payload (dict): The data to send to the Zapier webhook. Returns:
    str: Confirmation message upon successful triggering of the Zap.
    Raises: ValueError: If the API request fails or returns an error.
    """

    f"https://hooks.zapier.com/hooks/catch/{zap_id}/"
        try:
            response = requests.post(zapier_webhook_url, json=payload)
            if response.status_code == 200:
                return f"Zapier webhook '{zap_id}' successfully triggered."

            else:
                raise ValueError(f"Zapier API Error: {response.status_code} - {response.text}")
        except
            requests.exceptions.RequestException as e:
                raise ValueError(f"Failed to trigger Zapier webhook '{zap_id}': {e}") #

@tool def send_slack_message(channel: str, message: str) -> str:
    """ Send a message to a specified Slack channel. Args: channel (str): The Slack channel ID or name w

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

    # Initialize the LLM with GPT-4o and bind the tools
    llm = ChatOpenAI(model_name="gpt-4o", temperature=0)
    llm_with_tools = llm.bind_tools([get_stock_price])

    messages = [HumanMessage("What is the stock price of Apple?")]

    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)

        print(tool_msg.name)
        print(tool_call['args'])
        print(tool_msg.content)
        messages.append(tool_msg)
        print()
```

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

Semantic Skill Selection

Another approach, Semantic Skill Selection, uses semantic representations to identify and select the relevant skill based on their semantic similarity to the task requirements. Ahead of time, each skill definition and description is embedded using an encoder-only model, such as OpenAI's ada model, Amazon's Titan model, Cohere's Embed model, BERT, or others. These skills are then indexed in a lightweight vector database. At runtime, the current context is then embedded using the same embedding model, a search is performed on the database, and the top skill is selected. The skill is then parameterized with a text completion model, invoked, and the response is used to compose the response for the user. This is the most common pattern, and is recommended for most use cases. It's typically faster than Generative Skill Selection, performant, and reasonably scalable.

Skill database setup:

```
import os
import requests
import logging
from langchain_core.tools import tool
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain_core.messages import HumanMessage, AIMessage, ToolMessage
from langchain.vectorstores import FAISS
import faiss
import numpy as np

# Initialize OpenAI embeddings and LLM
embeddings = OpenAIEmbeddings(openai_api_key=OPENAI_API_KEY)
llm = ChatOpenAI(api_key=OPENAI_API_KEY)

# Tool descriptions
tool_descriptions = {
    "query_wolfram_alpha": "Use Wolfram Alpha to compute mathematical expressions or retrieve informatio
    "trigger_zapier_webhook": "Trigger a Zapier webhook to execute predefined automated workflows.",
    "send_slack_message": "Send messages to specific Slack channels to communicate with team members."
}

# 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])  # Assuming all embeddings have the same dimension
index = faiss.IndexFlatL2(dimension)

# Normalize embeddings for cosine similarity
faiss.normalize_L2(np.array(tool_embeddings).astype('float32'))

# Convert list to FAISS-compatible format
tool_embeddings_np = np.array(tool_embeddings).astype('float32')
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"
}

def select_tool(query: str, top_k: int = 1) -> list:
    """
    Select the most relevant tool(s) based on the user's query using 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).astype('float32')
    faiss.normalize_L2(query_embedding.reshape(1, -1))
    D, I = index.search(query_embedding.reshape(1, -1), top_k)
    selected_tools = [index_to_tool[idx] for idx in I[0] if idx in index_to_tool]
    return selected_tools

def determine_parameters(query: str, tool_name: str) -> dict:
    """
    Use the LLM to analyze the query and determine the parameters for the tool 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 query: '{query}', what parameters should be used for
    ]

    # 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', "123456")  # Default Zap ID if not provided
        parameters["payload"] = response.get('payload', {"data": query})
    elif tool_name == "send_slack_message":
        parameters["channel"] = response.get('channel', "#general")
        parameters["message"] = response.get('message', query)

    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 None

if tool_name:
    # Use LLM to determine the parameters based on the query and the selected tool
```

```
    args = determine_parameters(user_query, tool_name)

    # Invoke the selected tool
    try:
        # Assuming each tool has an `invoke` method to execute it
        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.")
```

If your scenario involves a large number of skills, however, you might need to consider Hierarchical Skill Selection. This is especially true if many of those skills are semantically similar, and you are looking to improve skill selection accuracy at the price of higher latency and complexity. In this pattern, you organize your skills into groups, and provide a description for each group. Your skill selection (either Generative or Semantic) first selects a group, and then performs a secondary search only among the skills in that group. While this is slower and would be expensive to parallelize, it reduces the complexity of the skill selection task into two smaller chunks, and frequently results in higher overall skill selection accuracy. Crafting and maintaining these skill groups takes time and effort, so this is not recommended as a technique to begin with.

```
import os
import requests
import logging
import numpy as np

from langchain_core.tools import tool
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain_core.messages import HumanMessage, AIMessage, ToolMessage
from langchain.vectorstores import FAISS
import faiss

embeddings = OpenAIEmbeddings(openai_api_key=OPENAI_API_KEY)
# Define tool groups with descriptions
tool_groups = {
    "Computation": {
        "description": "Tools related to mathematical computations and data analysis.",
        "tools": []
    },
    "Automation": {
        "description": "Tools that automate workflows and integrate different services.",
        "tools": []
    },
    "Communication": {
        "description": "Tools that facilitate communication and messaging.",
        "tools": []
    }
}

# Define Tools
@tool
def query_wolfram_alpha(expression: str) -> str:
    api_url = f"https://api.wolframalpha.com/v1/result?i={requests.utils.quote(expression)}&appid={WOLFR
    try:
        response = requests.get(api_url)
        if response.status_code == 200:
            return response.text
        else:
```

```
            raise ValueError(f"Wolfram Alpha API Error: {response.status_code} - {response.text}")
    except requests.exceptions.RequestException as e:
        raise ValueError(f"Failed to query Wolfram Alpha: {e}")

@tool
def trigger_zapier_webhook(zap_id: str, payload: dict) -> str:
    """
    Trigger a Zapier webhook to execute a predefined Zap.

    Args:
        zap_id (str): The unique identifier for the Zap to be triggered.
        payload (dict): The data to send to the Zapier webhook.

    Returns:
        str: Confirmation message upon successful triggering of the Zap.

    Raises:
        ValueError: If the API request fails or returns an error.
    """
    zapier_webhook_url = f"https://hooks.zapier.com/hooks/catch/{zap_id}/"
    try:
        response = requests.post(zapier_webhook_url, json=payload)
        if response.status_code == 200:
            return f"Zapier webhook '{zap_id}' successfully triggered."
        else:
            raise ValueError(f"Zapier API Error: {response.status_code} - {response.text}")
    except requests.exceptions.RequestException as e:
        raise ValueError(f"Failed to trigger Zapier webhook '{zap_id}': {e}")

@tool
def send_slack_message(channel: str, message: str) -> str:
    """
    Send a message to a specified Slack channel.

    Args:
        channel (str): The Slack channel ID or name where the message will be sent.
        message (str): The content of the message to send.

    Returns:
        str: Confirmation message upon successful sending of the Slack message.

    Raises:
        ValueError: If the API request fails or returns an error.
    """
    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=headers, json=payload)
        response_data = response.json()
        if response.status_code == 200 and response_data.get("ok"):
            return f"Message successfully sent to Slack channel '{channel}'."
        else:
            error_msg = response_data.get("error", "Unknown error")
            raise ValueError(f"Slack API Error: {error_msg}")
    except requests.exceptions.RequestException as e:
        raise ValueError(f"Failed to send message to Slack channel '{channel}': {e}")

# Assign tools to their respective groups
```

```
tool_groups["Computation"]["tools"].append(query_wolfram_alpha)
tool_groups["Automation"]["tools"].append(trigger_zapier_webhook)
tool_groups["Communication"]["tools"].append(send_slack_message)

# ------------------------------
# Embed Group and Tool Descriptions
# ------------------------------
# Embed group descriptions
group_names = []
group_embeddings = []
for group_name, group_info in tool_groups.items():
    group_names.append(group_name)
    group_embeddings.append(embeddings.embed_text(group_info["description"]))

# Create FAISS index for groups
group_embeddings_np = np.array(group_embeddings).astype('float32')
faiss.normalize_L2(group_embeddings_np)
group_index = faiss.IndexFlatL2(len(group_embeddings_np[0]))
group_index.add(group_embeddings_np)

# Embed tool descriptions within each group
tool_indices = {}  # Maps group name to its FAISS index and tool functions
for group_name, group_info in tool_groups.items():
    tools = group_info["tools"]
    tool_descriptions = []
    tool_functions = []
    for tool_func in tools:
        description = tool_func.__doc__.strip().split('\n')[0]  # First line of docstring
        tool_descriptions.append(description)
        tool_functions.append(tool_func)
    if tool_descriptions:
        tool_embeddings = embeddings.embed_texts(tool_descriptions)
        tool_embeddings_np = np.array(tool_embeddings).astype('float32')
        faiss.normalize_L2(tool_embeddings_np)
        tool_index = faiss.IndexFlatL2(len(tool_embeddings_np[0]))
        tool_index.add(tool_embeddings_np)
        tool_indices[group_name] = {
            "index": tool_index,
            "functions": tool_functions,
            "embeddings": tool_embeddings_np
        }

# ------------------------------
# Hierarchical Skill Selection
# ------------------------------
def select_group(query: str, top_k: int = 1) -> list:
    query_embedding = embeddings.embed_text(query).astype('float32')
    faiss.normalize_L2(query_embedding.reshape(1, -1))
    D, I = group_index.search(query_embedding.reshape(1, -1), top_k)
    selected_groups = [group_names[idx] for idx in I[0]]
    return selected_groups

def select_tool(query: str, group_name: str, top_k: int = 1) -> list:
    tool_info = tool_indices[group_name]
    query_embedding = embeddings.embed_text(query).astype('float32')
    faiss.normalize_L2(query_embedding.reshape(1, -1))
    D, I = tool_info["index"].search(query_embedding.reshape(1, -1), top_k)
    selected_tools = [tool_info["functions"][idx] for idx in I[0] if idx < len(tool_info["functions"])]
    return selected_tools


# Initialize the LLM with GPT-4 and set temperature to 0 for deterministic responses
llm = ChatOpenAI(model_name="gpt-4", temperature=0)

   selected_groups = select_group(user_query, top_k=1)
```

```
    if not selected_groups:
        print("No relevant skill group found for your query.")
        return

    selected_group = selected_groups[0]
    logging.info(f"Selected Group: {selected_group}")
    print(f"Selected Skill Group: {selected_group}")

    # Step 2: Select the most relevant tool within the group
    selected_tools = select_tool(user_query, selected_group, top_k=1)

    if not selected_tools:
        print("No relevant tool found within the selected group.")
        return

    selected_tool = selected_tools[0]
    logging.info(f"Selected Tool: {selected_tool.__name__}")
    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 expression
        args["expression"] = user_query
    elif selected_tool == trigger_zapier_webhook:
        # For demonstration, use placeholders
        args["zap_id"] = "123456"  # Replace with actual Zap ID
        args["payload"] = {"message": user_query}
    elif selected_tool == send_slack_message:
        # For demonstration, use placeholders
        args["channel"] = "#general"  # Replace with actual Slack channel
        args["message"] = user_query
    else:
        print("Selected tool is not recognized.")
        return

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

Machine Learned Skill Selection

Machine Learned Skill Selection employs machine learning techniques to automatically learn and select skills based on past experiences and task feedback. Generic generative and embedding models are often larger, slower, and more expensive than is necessary for skill selection, so by training specific models on task-skill pairs, you can potentially reduce the cost and latency of this part of your agent-based solution. Both historical data and data samples generated by a foundation model can be used to train your skill selection model. Similarly, you could fine-tune a smaller model to improve the classification performance on your skill selection task. The key drawback is it introduces a new model that your team will need to maintain. Carefully consider the costs before choosing to proceed down this path, as it may require extensive training data and computational resources to achieve optimal performance.

Skill Execution

Parametrization is the process of defining and setting the parameters that will guide the execution of a skill 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 skill definition as discussed in more detail in Chapter 3. 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 skill execution phase begins. This involves the actual execution of the skill. Some of these skills 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.

Skill Topologies

Today, the majority of chatbot systems rely on Single Skill 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 Skill Execution. For many cases, however, we want our agents to be able to perform complex tasks that require multiple skills. By providing an agent with a sufficient range of skills, you can then enable your agent to flexibly arrange those skills 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 skills, and define the skills topology in which the agent can operate, then allow the exact composition to be designed dynamically in response to the context and task at hand. This section considers this range of skill topologies and discusses their tradeoffs.

Single Skill Execution

We'll begin with tasks that require precisely one skill. In this case, planning consists of choosing the one skill most appropriate to address the task. Once the skill is selected, it must be correctly parameterized based on the skill definition. The skill is then executed, and its output is used as an input when composing the final response for the user. This can be seen in Figure 4-1 below (TODO: draft). While this is a minimal definition

of a plan, it is the foundation from which we will build more complex patterns.

Parallel Skill Execution

The first increase in complexity comes with skill parallelism. In some cases, it might be worth taking multiple actions on the input. For example, consider you need to look up a record for a patient. If your skillset includes multiple skills 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 skills need to be executed. A common approach is to retrieve a maximum number of skills that might be executed, say 5, using Semantic Skill Selection. Next, make a second call to a foundation model with each of these five skills, and ask it to select the five or fewer skills that are necessary to the problem, filtering down to the skills necessary for the task. Similarly, the foundation model can be called repeatedly with the additional context of which skills have already been selected until it chooses to add no fewer skills. Once selected, these skills are independently parameterized and executed. After all skills have been completed, their results are passed to the foundation model to draft a final response for the user. Figure 4-2 (TODO) illustrates this pattern.

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.

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

Trees

In more complex scenarios, tasks may require branching sequences of actions, where the agent must choose between multiple possible paths at each decision point. Planning trees involve exploring different branches of action possibilities, evaluating the consequences of each choice, and selecting the most promising path towards the goal. Trees are useful for tasks with multiple options or alternative courses of action. This structure enables the natural expansion that is involved in certain tasks, especially when a prior skill returns multiple outputs that need to be considered.

By increasing the skill topology from a chain to a tree, the skill structure contains the state of the execution. Compared to the chain structure, the

agent has more options to choose from. In addition to selecting and executing a skill from its current position, the agent can determine that the task has been completed, decide that it is unable to complete the task, or it can traverse to another leaf node on the tree, and proceed from there. This structure reduces the likelihood that subtasks are forgotten by the agent. In this structure, key parameters to tune for your use case include the maximum number of skills per execution and the maximum depth of the tree.

Graphs

Graphs represent interconnected networks of actions, where dependencies between tasks can be more complex and nonlinear. Graphs are an extension of trees, and while they enable the same expansion to multiple items as trees, they also enable topologies that consolidate multiple nodes together. This structure allows for an expressive representation that tracks the flow of information across multiple skill executions.

In addition to the tree structure, the graph structure adds a new action: consolidate. This new action enables the agent to connect the results from multiple previously completed skills. This is invaluable when especially complex reasoning is desirable, and the agent is expected to stitch together findings from multiple previous skills. While graphs are a more flexible and expressive structure, they are more complex to manage and traverse, and open up a new class of errors that the agent can make.

Choosing a Topology

Selecting the appropriate topology is crucial for effectively organizing and executing actions. Linear topologies, such as chains, are suitable for tasks with a sequential flow of actions where each step leads directly to the next. This topology simplifies the planning process, as actions are executed in a straightforward order without branching or decision points.

Hierarchical topologies, such as trees, are useful for organizing actions into nested levels of abstraction. This topology allows for both high-level strategic planning and detailed, fine-grained control over individual actions. Hierarchical topologies are well-suited for tasks with multiple layers of complexity or tasks that can be decomposed into subtasks with distinct goals. For example, in project management, tasks can be organized hierarchically based on their dependencies and relationships.

Graph topologies offer the most flexibility and expressiveness in representing complex relationships between actions. In this type of graph, actions are interconnected, allowing for nonlinear dependencies and dynamic decision-making. This topology is ideal for tasks with interconnected components, where actions can affect each other in unpredictable ways. Graph topologies are commonly used in tasks such as resource allocation, where the allocation of resources depends on multiple factors and constraints.

Choosing the appropriate topology depends on factors such as the complexity of the task, the degree of interdependence between actions, and the level of flexibility required in the planning process. By selecting the right topology, autonomous agents can effectively organize their plans, navigate complex environments, and achieve their goals with efficiency

and adaptability. As a principle, start simple, and only add complexity to address specific needs in your use case.

Planning

Now that we have discussed a range of skill topologies, it's time to consider how to use them. We'll begin with the simplest approach, then move through several more complex approaches. Note that any skill topology can be used with any planning approach.

Iterative Planning

We begin with a discussion of the simplest type of plan, which is iterative planning. In this approach, the agent chooses an action and executes it. You can think of this as the "unplanned" or "greedy" approach to planning. This has multiple advantages, including simplicity, lower latency, and easier maintainability. This approach can handle many use cases and is the recommended starting point. For tasks that require a small number of skills, this is probably sufficient.

Zero-Shot Planning

For more complex tasks, it will be necessary to draft a plan before beginning. The more complex the task, the more subtasks it will require. By simply executing one action at a time in a greedy approach, agents will sometimes get caught in loops and fail to make progress toward completion. Taking the time to create a plan, and then to choose actions toward that plan, can increase the overall performance on these tasks that require multiple steps.

The simplest place to start with planning is zero-shot planning, which refers to the ability of an agent to generate plans for tasks it has never encountered before, based solely on its understanding of the task and its environment. This approach requires the agent to possess a robust representation of the task space, including possible actions, their effects, and the relationships between different components. By leveraging this knowledge, the agent can generate plans on the fly, even for novel tasks, without requiring explicit training data or pre-defined solutions. Zero-shot planning is particularly useful in dynamic environments where tasks may vary over time or where the agent needs to adapt quickly to new challenges.

In-Context Learning with Hand-Crafted Examples

This brings us to hand-crafted examples, where developers design plans for specific tasks manually, based on domain expertise or predefined rules. In this approach, human designers analyze the task requirements, identify relevant actions, and determine the sequence in which these actions should be executed to achieve the desired outcome. Hand-crafted plans are often used for tasks with well-defined structures or known solutions, where human intuition or expertise can guide the planning process effectively. While hand-crafted plans offer precision and reliability, they may lack flexibility and scalability, as they rely heavily on human intervention and may not generalize well to new scenarios. By scoping core, high-value scenarios, developers can provide clear examples. By embed-

ding and indexing these into a few-shot database, the relevant examples can be pulled to guide future plans based on semantically-similar examples.

Plan Adaptation

In scenarios where subsequent skills depend on the output of previous skills, the ability to adapt a plan will be necessary. Plan adaptation enables agents to respond to new information, unexpected events, or deviations from the original plan, ensuring continued progress towards their goals. The leading approach to this type of plan reaction, Reason-Act, or ReAct[1] for short, provides a simple but effective framework for approaching the task of plan adaptation. As the name suggests, the agent alternates between reasoning steps and acting steps. In the reasoning step, the agent is asked to consider what it needs to do to answer the question. The foundation model is then invoked to choose the action, one of which is to complete the execution flow. This enables the agent to repeatedly take actions, such as looking up data, and checking to see if the results from the search are sufficient to meet the task. If not, the agent can choose to continue to search or take additional actions. A further extension of this work is PlanReAct, which adds an additional self-think flow, which includes Chain of Thought reasoning. This combines planning, reasoning, and acting into a cohesive process.

Summary

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 tradeoff 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 priori 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 onto the next part of the workflow: memory. It is worth noting that 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.

[1] Shunyu Yao et al.: ReAct: Synergizing Reasoning and Acting in Language Models. Published as a conference paper at ICLR 2023. https://arxiv.org/pdf/2210.03629