Agentic AI: A Comprehensive Overview

This document provides an in-depth exploration of Agentic AI, covering its core patterns, prerequisites, popular uses, limitations, and architectural components.

March 23, 2025

Contents

Agentic AI Overview **2**

1 Agent AI Prerequisites **2**

2 Patterns in Agentic AI **2**

3 Benefits of Agentic AI **3**

4 Framework in Agentic AI **3**

5 Architecture of Agentic AI **4**

6 Planner and LLM **5**

7 Orchestrator in Agentic AI **6**

8 Executor task in Agentic AI **7**

9 Implementation of a basic AI Agent **7**

10 Pattern in Agentic AI **12**
10.1 Reflection 12
10.2 Routing 13
10.3 Tool Use 14
10.4 Planning Pattern 14

11 Customer services AI agents **15**

Agentic AI Overview

Agentic AI refers to artificial intelligence designed to operate autonomously. It is capable of making decisions and taking actions toward specific goals with minimal human intervention. Agentic AI uses generative AI models to make decisions autonomously. Characteristics of agentic AI include:

- Natural language input and output.
- Autonomy in planning and execution.
- Adaptability to changing inputs and environments.
- Proactive decision-making and actions.
- Awareness of dynamic contexts.
- Interactive and continuous learning.
- Self-validation to overcome hallucination and bias.

1. Agent AI Prerequisites

- Understanding of generative AI concepts.
- Familiarity with LLMs and embeddings.
- Proficiency in Python programming (preferably in a notebook environment).
- Familiarity with the LLama-Index framework.

2. Patterns in Agentic AI

- **Planning:** This pattern involves determining a structured sequence of actions to achieve a desired objective. It includes setting clear goals, decomposing complex tasks into smaller, manageable subtasks, and formulating strategies that allow the system to adapt dynamically to changes in the environment.
- **Tool Use:** Here, the AI system integrates and utilizes a variety of tools and resources to execute tasks efficiently. It involves selecting the most appropriate tools based on the requirements of the task, configuring the necessary inputs for these tools, and then aggregating and processing their outputs to contribute to the overall solution.
- **Routing:** This pattern focuses on directing data or requests to the optimal processing paths within the system. The AI analyzes the input to determine the best route for processing, thereby ensuring that requests are forwarded to the most suitable subsystems or external services for specialized handling, which improves overall efficiency.

- **Reflection:** Reflection entails continuous self-assessment and evaluation of the system's actions and decisions. The AI reviews outcomes, assesses performance, and implements feedback mechanisms. This process of self-refinement helps to correct errors, improve accuracy, and enhance the overall decision-making process.

- **Multi-Agent:** In this pattern, multiple autonomous AI agents operate concurrently, collaborating to solve complex tasks that are beyond the capability of a single agent. These agents communicate, share resources, and coordinate their actions, leading to distributed problem-solving and more robust, flexible performance in handling intricate challenges.

3. Benefits of Agentic AI

- **Increased Efficiency through Autonomy:** Agentic AI systems operate with minimal human intervention, enabling them to execute tasks more rapidly and accurately. This autonomy reduces the need for constant oversight, allowing processes to run continuously and efficiently.

- **Improved Productivity:** By automating complex decision-making and task execution, agentic AI can significantly boost productivity. The system's ability to manage and complete multiple tasks concurrently ensures that outputs are delivered faster, optimizing resource utilization.

- **Enhanced Adaptability and Continuous Learning:** Agentic AI systems are designed to adapt to changing environments and inputs. Through ongoing learning and self-improvement mechanisms, these systems continuously refine their decision-making processes and adjust to new challenges without manual reprogramming.

- **Superior User Experience:** With natural language interfaces and proactive engagement, agentic AI creates more intuitive and responsive interactions. This leads to a smoother and more personalized user experience, as the system anticipates needs and responds promptly to queries or issues.

- **Complex Problem Solving and Workflow Automation:** Agentic AI integrates advanced planning, tool use, and multi-agent coordination to handle intricate problems. It streamlines workflows by automating routine and complex processes, which reduces errors and frees up human resources for more strategic tasks.

4. Framework in Agentic AI

- **LangGraph:** Based on LangChain, LangGraph is designed to facilitate the construction of complex AI workflows. It integrates multiple language model-based operations into a coherent system, enabling developers to design, manage, and optimize various tasks with modular components that communicate effectively.

- **Autogen:** Developed by Microsoft, Autogen is a framework specialized for multi-agent conversations. It enables coordinated dialogue management among several autonomous agents, ensuring they can collaborate, share information, and work towards common objectives efficiently.

- **Crew AI:** Crew AI provides tools for building both single-agent and multi-agent systems. It streamlines the development process by offering libraries and utilities that support the creation, deployment, and management of autonomous agents, making it easier to scale complex AI applications.

- **LLama-Index Agents:** This framework offers a robust structure for building agentic AI systems by focusing on effective data indexing and organization. It optimizes performance by allowing AI agents to retrieve and process relevant information quickly, thereby enhancing decision-making and task execution.

- **Cloud Services:** Cloud platforms such as Amazon Bedrock, Vertex AI, and Azure OpenAI provide the necessary infrastructure and scalable resources to support agentic AI systems. They offer computational power, storage solutions, and specialized AI services that facilitate the deployment, management, and scaling of autonomous AI workflows.

5. Architecture of Agentic AI

An agentic AI system comprises several components working together to accomplish complex tasks autonomously. These components are designed to communicate seamlessly, share context, and dynamically adapt to changes in the environment or the task at hand. A brief overview of each component is provided below:

- **Goal:** This is the primary objective or task assigned to the agent. It defines what the system aims to accomplish and serves as a guiding principle for all subsequent actions. Goals can be high-level, such as "improve user engagement," or more concrete, such as "generate a detailed report on sales trends."

- **Input:** In addition to the goal, the system can receive supplementary data or instructions that refine or contextualize its actions. Inputs may include user queries, sensor data, historical logs, or domain-specific datasets. The agent uses this information to make informed decisions, adapt strategies, and maintain relevance to the task.

- **Orchestrator:** Often considered the central coordinating entity, the Orchestrator manages the overall workflow of the agentic AI system. It acts as the point of contact for all incoming goals and inputs, then delegates tasks to other components. The Orchestrator also integrates and monitors the outputs of these components, ensuring the final results align with the system's objectives.

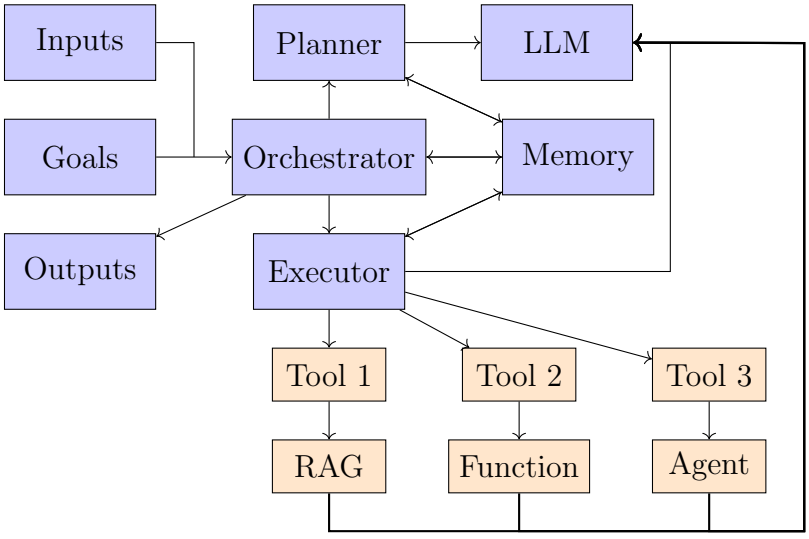
- **Planner:** Responsible for devising a step-by-step strategy to achieve the goal, the Planner uses both logical reasoning and AI-driven insights (such as those provided by language models) to break down complex tasks into smaller, manageable sub-tasks. It identifies the

tools or resources needed, determines the sequence of actions, and creates a structured plan that the Executor can follow.It is responsible for:

- Maintain an inventory of tools and their capabilities.
- Decompose the overall goal into tasks and subtasks.
- Identify and sequence tasks to create a workflow.
- Select appropriate tools.
- Using and updating memory as needed.

- **Executor:** The Executor carries out the plan generated by the Planner. It orchestrates the usage of various tools and modules, passing inputs to them and collecting their outputs. By executing tasks sequentially or in parallel (as dictated by the plan), the Executor ensures each step is completed accurately. Once all sub-tasks are done, the Orchestrator compiles the results into the final output.

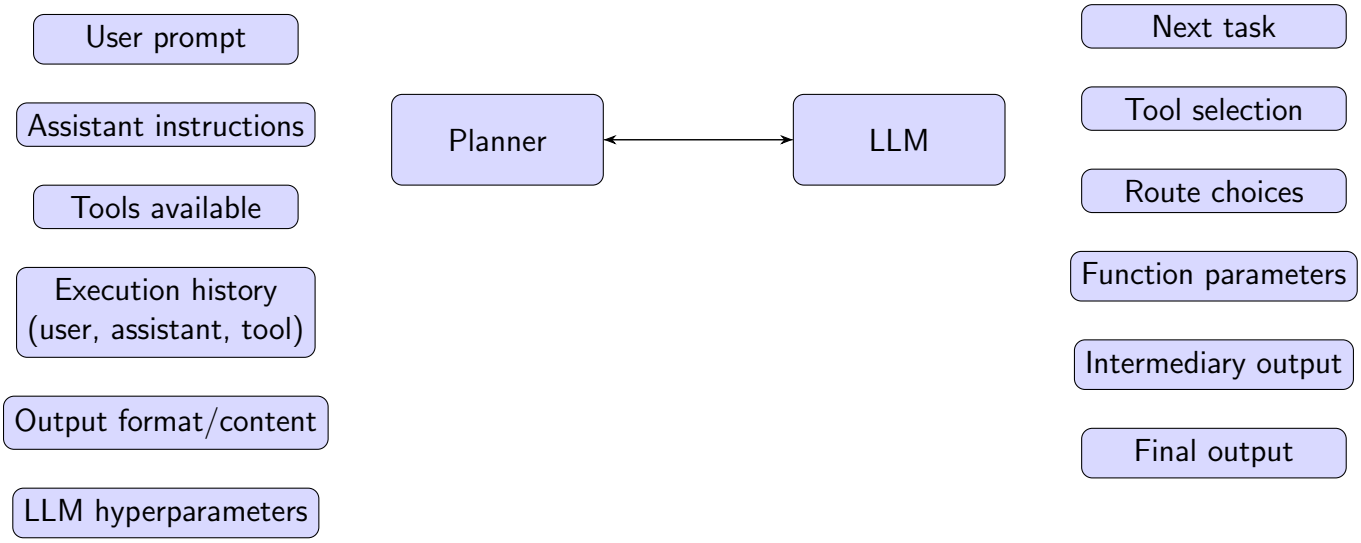
- **Memory:** Memory provides both short-term and long-term context, enabling the system to maintain continuity across multiple interactions or stages of the workflow. Short-term memory might store recent tool outputs or partial solutions, while long-term memory retains historical data, learned experiences, or user preferences. This contextual information can be critical for decision-making, as it helps the agent avoid redundant actions and adapt more effectively to changes or feedback.



6. Planner and LLM

- **Execution Plan Creation:** The planner leverages large language models (LLMs) to generate a comprehensive execution plan. It interprets the overall goal and breaks it down into actionable steps, ensuring that each task is clearly defined and logically sequenced.

- **Iterative Invocation:** Throughout the workflow, the planner may invoke LLMs multiple times. This iterative process allows it to refine the plan, verify intermediate outputs, and make adjustments based on evolving inputs and context.
- **Input Understanding and Task Decomposition:** LLMs play a crucial role in understanding the diverse inputs provided to the system. They help decompose complex tasks into smaller, manageable sub-tasks, which are then assigned to appropriate tools or modules for execution.
- **Tool Selection and Step Generation:** By analyzing the current context and available resources, the LLM assists in selecting the most suitable tools to accomplish each sub-task. It also generates the next steps required to move the process forward, ensuring continuity and coherence in the workflow.
- **Prompt Engineering and Cost Considerations:** Effective prompt engineering is central to the planner's performance. Crafting precise and contextually rich prompts can significantly enhance the quality of the LLM's output. However, frequent or elaborate prompt constructions may lead to high computational costs, which necessitates careful balancing between performance and resource expenditure.



7. Orchestrator in Agentic AI

- **Central Communication Hub:** The orchestrator serves as the single point of contact, managing all inputs, goals, and outputs. It coordinates communication between various components of the system, ensuring a streamlined flow of information.
- **Service Awareness:** It maintains a comprehensive understanding of all available system services and their capabilities. This awareness allows the orchestrator to effectively route tasks and data to the appropriate services.

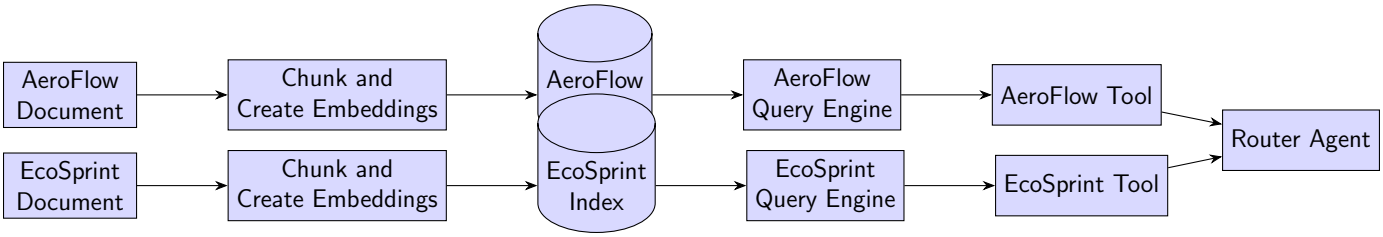
- **Workflow Creation:** By invoking the planner, the orchestrator initiates the creation of detailed workflows. These workflows break down complex goals into manageable steps and sub-tasks, setting the stage for execution.
- **Execution Trigger and Tracking:** Once a workflow is defined, the orchestrator triggers the executor to carry out the tasks. It continuously monitors the progress of execution, ensuring that each step aligns with the overall goal.
- **Performance Monitoring:** The orchestrator actively tracks the executor's performance, identifying any deviations or issues in real-time. This monitoring enables timely adjustments and refinements to improve efficiency.
- **State Management with Memory:** Leveraging both short-term and long-term memory, the orchestrator preserves the state of ongoing processes and historical data. This memory integration supports context-aware decision-making and enhances the system's responsiveness.
- **Iterative Planning and Execution:** Recognizing that achieving complex goals often requires refinement, the orchestrator supports multiple iterations of planning and execution. It adapts to feedback and evolving conditions, ensuring continuous improvement and goal alignment.

8. Executor task in Agentic AI

- Executes tasks or actions designed by the planner.
- Leverages the available tools within the agentic AI system.
- Invokes tools, passes input, and collects outputs.
- Responsibilities between the orchestrator and task executor can be interchangeable or combined.

9. Implementation of a basic AI Agent

This explains a Python code that sets up a local LLM using the Ollama model `qwq:32b` via LlamaIndex, along with a HuggingFace embedding model and PDF document indexing. The code also creates a router query engine that routes queries to the correct document index.



Code Listing

```
import os
import sys
import time
import nest_asyncio
import requests
from typing import List, Optional

import torch

# LlamaIndex imports
from llama_index.core import Settings
from llama_index.core import SimpleDirectoryReader
from llama_index.core.node_parser import SentenceSplitter
from llama_index.core import VectorStoreIndex
from llama_index.core.tools import QueryEngineTool
from llama_index.core.query_engine.router_query_engine import
   RouterQueryEngine
from llama_index.core.selectors import LLMSingleSelector

# Patch asyncio for Jupyter (if needed)
nest_asyncio.apply()

# ------------------------------
# Custom Ollama LLM for "qwq:32b"
# ------------------------------
class OllamaLLM:
    """
    A custom LLM wrapper that calls your locally installed Ollama
       LLM for model "qwq:32b".
    """
    def __init__(self, model: str = "qwq:32b", api_url: str =
       "http://localhost:11434/"):
        self.model = model
        self.api_url = api_url

    def _call(self, prompt: str, stop: Optional[List[str]] = None)
       -> str:
        payload = {
            "model": self.model,
            "prompt": prompt,
            # Optionally include additional parameters like
               temperature, max_tokens, etc.
        }
        response = requests.post(self.api_url, json=payload)
        response.raise_for_status()
```

```
        data = response.json()
        # Adjust the key below based on your Ollama API's response structure.
        return data.get("completion", "")

    def __call__(self, prompt: str, stop: Optional[List[str]] = None) -> str:
        return self._call(prompt, stop)

# Set up the custom Ollama LLM for "qwq:32b"
Settings.llm = OllamaLLM(model="qwq:32b", api_url="http://localhost:11434/")

# --------------------------------
# Embedding Model Setup
# --------------------------------
# For embeddings, we use HuggingFace via the SentenceTransformer.
from llama_index.embeddings.huggingface import HuggingFaceEmbedding

% % Alternatively, one can use:
% from sentence_transformers import SentenceTransformer

class SentenceTransformerEmbedding:
    """
    A simple embedding wrapper using SentenceTransformer.
    """
    def __init__(self, model_name: str = "all-MiniLM-L6-v2"):
        self.model = HuggingFaceEmbedding(model_name=model_name)

    def embed(self, texts):
        if isinstance(texts, str):
            texts = [texts]
        return self.model.embed(texts)

    def get_text_embedding_batch(self, texts: List[str], **kwargs) -> List[List[float]]:
        """
        Returns embeddings for a list of texts. Accepts additional keyword arguments.
        """
        return self.embed(texts)

Settings.embed_model = SentenceTransformerEmbedding(model_name=
"sentence-transformers/all-MiniLM-L6-v2")

# --------------------------------
# Document Index Setup
```

```
# ------------------------------
splitter = SentenceSplitter(chunk_size=1024)

# ------------------------------------------------------------
# Helper Function: Create Query Engine from a PDF file
# ------------------------------------------------------------
def create_query_engine(pdf_file_path: str):
    # Read the document(s)
    documents =
       SimpleDirectoryReader(input_files=[pdf_file_path]).load_data()
    # Parse documents into nodes using the sentence splitter
    nodes = splitter.get_nodes_from_documents(documents)
    # Create the vector store index
    index = VectorStoreIndex(nodes)
    # Return the query engine for the index
    return index.as_query_engine()

# ------------------------------------------------------------
# Setup Document Indexes for Vector Search
# ------------------------------------------------------------
# Aeroflow Specification Document
aeroflow_query_engine =
   create_query_engine("AeroFlow_Specification_Document.pdf")

# EcoSprint Specification Document
ecosprint_query_engine =
   create_query_engine("EcoSprint_Specification_Document.pdf")

from llama_index.core.tools import QueryEngineTool
from llama_index.core.query_engine.router_query_engine import
   RouterQueryEngine
from llama_index.core.selectors import LLMSingleSelector

# Create a query engine tool for Aeroflow specifications
aeroflow_tool = QueryEngineTool.from_defaults(
    query_engine=aeroflow_query_engine,
    name="Aeroflow specifications",
    description="Contains information about Aeroflow: Design,
       features, technology, maintenance, warranty"
)

# Create a query engine tool for EcoSprint specifications
ecosprint_tool = QueryEngineTool.from_defaults(
    query_engine=ecosprint_query_engine,
    name="EcoSprint specifications",
    description="Contains information about EcoSprint: Design,
       features, technology, maintenance, warranty"
```

```
)

# Create a Router Agent to select the appropriate tool based on the query
router_agent = RouterQueryEngine(
    selector=LLMSingleSelector.from_defaults(),
    query_engine_tools=[aeroflow_tool, ecosprint_tool],
    verbose=True
)

# ------------------------------
# Query Examples
# ------------------------------
response = router_agent.query("What colors are available for AeroFlow?")
print("\nResponse: ", str(response))

response = router_agent.query("What colors are available for EcoSprint?")
print("\nResponse: ", str(response))
```

Explanation of the Code Blocks

- **Module Imports and Async Patch:**
  - Imports necessary modules such as `nest_asyncio`, `requests`, and `torch` along with various components from `llama_index`.
  - `nest_asyncio.apply()` is called to allow nested event loops (useful in Jupyter).
- **Custom Ollama LLM Setup:**
  - Defines a class `OllamaLLM` that wraps the local Ollama LLM model (`qwq:32b`).
  - Implements a private method `_call` to send the prompt via a POST request to the API endpoint.
  - The class is then instantiated and assigned to `Settings.llm`.
- **Embedding Model Setup:**
  - Uses the HuggingFace embedding model through the `HuggingFaceEmbedding` class (or optionally, SentenceTransformer).
  - A custom embedding class, `SentenceTransformerEmbedding`, is defined with methods `embed` and `get_text_embedding_batch` (which accepts additional keyword arguments).
  - This embedding instance is assigned to `Settings.embed_model`.

- **Document Index Setup and Helper Function:**
  - A `SentenceSplitter` is initialized to divide documents into nodes.
  - The helper function `create_query_engine` reads a PDF using `SimpleDirectoryReader`, splits it into nodes, creates a `VectorStoreIndex`, and returns its query engine.
- **Creating Document Indexes:**
  - Two document indexes are created for the Aeroflow and EcoSprint specification PDFs using the helper function.
- **Query Engine Tools and Router Agent:**
  - Wraps each document query engine in a `QueryEngineTool` with appropriate names and descriptions.
  - A `RouterQueryEngine` is then instantiated with these tools and a selector (`LLMSingleSelector`) to route incoming queries.
- **Query Examples:**
  - Two sample queries are executed: one for Aeroflow and one for EcoSprint.
  - The responses are printed to the console.

Code & results are here

10. Pattern in Agentic AI

10.1. Reflection

Reflection involves the use of a large language model (LLM) as a reviewer or evaluator. An LLM can assess its own work or that of another LLM by using a defined list of criteria. The evaluation may include external references to verify facts and can produce textual feedback or even numerical scores. This feedback is then provided back to the generator LLM to refine its responses through an iterative process until the results meet the desired quality.

– Use an LLM as a reviewer/evaluator.

– An LLM can review its own or another LLM's work.

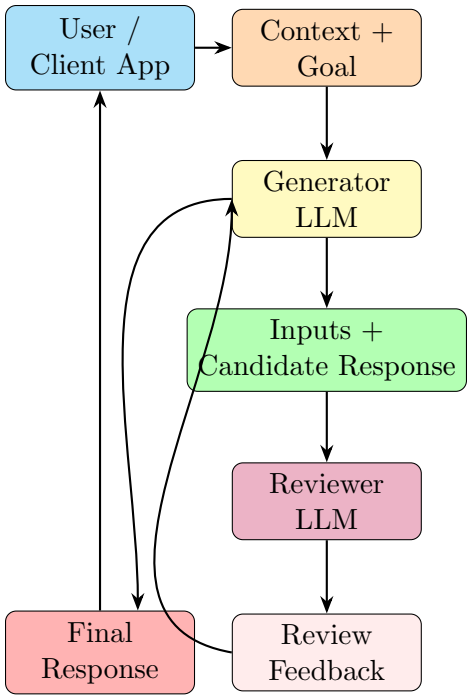
– Reflection uses a list of criteria to evaluate.

– May use external references to validate facts.

– Produces text feedback or a score.

– Feedback is provided to the generator LLM for improvement.

– Iterative process until satisfactory results are achieved.



10.2. Routing

Routing refers to the mechanism by which an agent selects among alternate actions available in its inventory. These actions might include data sources, procedures, retrieval techniques, or system integrations. Each action has an associated capability profile that helps the LLM decide which action to execute based on the specific goal at hand. Once the decision is made, the agent proceeds to execute the chosen action.

– An agent can have a set of alternate actions available in its inventory.

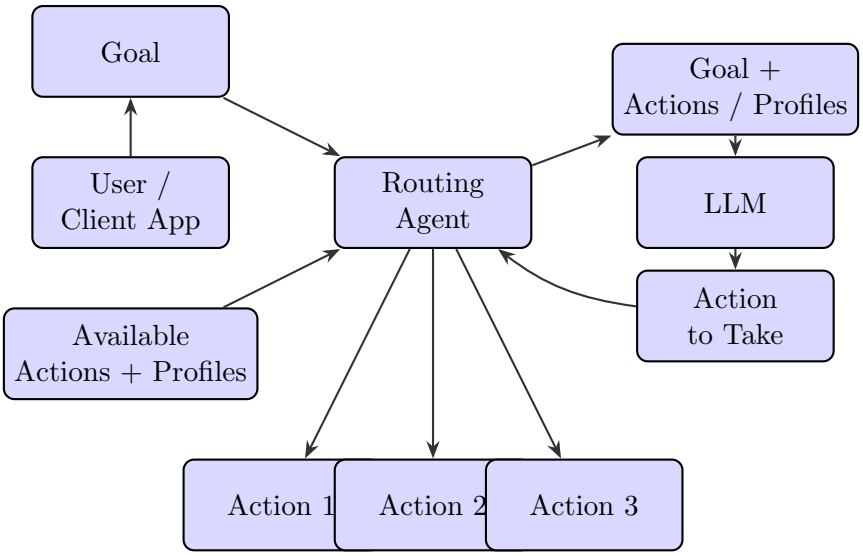
– Actions may include data sources, procedures, retrieval techniques, or system integrations.

– Each action has an associated profile of its capabilities.

– Given a goal, the agent uses the LLM to decide which action to execute.

– Action profiles help the LLM to make a choice based on the goal.

– The agent then executes the action chosen by the LLM.



10.3. Tool Use

The tool use pattern leverages an LLM to determine when and how to employ tools, including the selection of appropriate inputs. While similar to routing, the tool use pattern emphasizes that tools come with defined inputs and associated capability profiles. Given a goal, the LLM not only selects the appropriate tool from the agent's inventory (which may include functions, data sources, or system integrations) but also decides the parameter values for its operation. The agent then executes the tool using the chosen inputs.

- Leverages an LLM to decide when and how to use a tool.
- Tools are actions with specific inputs.
- An agent can have a set of tools available (e.g., data sources, functions, retrieval techniques, system integrations).
- Each tool has an associated profile of its capabilities and inputs.
- Given a goal, the LLM selects the appropriate tool.
- The LLM also decides the parameter values for the tool.
- The agent executes the tool with the chosen inputs.

10.4. Planning Pattern

The planning pattern involves breaking down a complex goal into a series of subgoals and tasks that form a coherent workflow. An LLM is used to decompose a high-level objective into manageable steps, with each step potentially requiring the use of different routes or tools. This systematic breakdown facilitates more precise execution and can integrate other patterns such as reflection, routing, and tool use.

– Leverages an LLM to break down a goal into subgoals and tasks.

– Complex goals require decomposition into individual steps.

– Each step may require corresponding routes or tools.

– Facilitates the creation of an executable plan.

– Can integrate with reflection, routing, tool use, and multi-agent patterns.

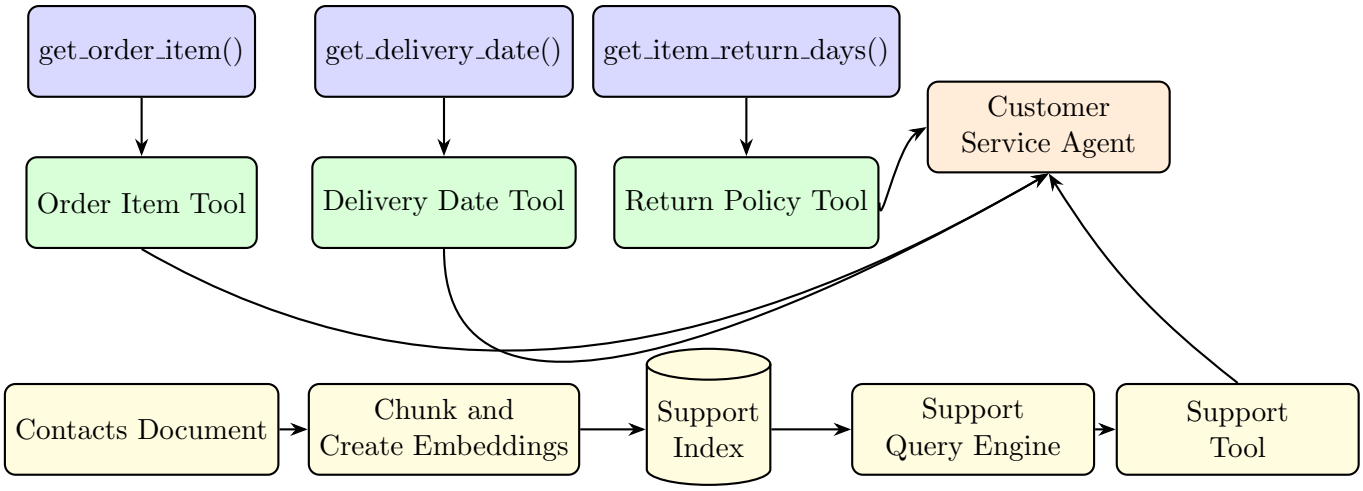
11. Customer services AI agents

It demonstrates the architecture of customer service AI agents. In the diagram, functions such as `get_order_item()`, `get_delivery_date()`, and `get_item_return_days()` are linked to their corresponding tools, which process the input and feed the results to the customer service agent. Additionally, a document processing pipeline is depicted that takes customer support documents through various stages-from chunking and embedding to indexing and querying-and finally routes the support data back to the agent.

This code sets up a local language model (LLM) using Ollama and integrates various tools for order processing and customer support. It includes:

- Functions for retrieving order items, delivery dates, and return policies.
- A vector database workflow for customer service documents using embeddings.
- Wrapping these functions as tools and creating an agent that orchestrates the tools using a function-calling mechanism.

Together, the diagram and code illustrate how multiple components (function calls, tools, and document pipelines) can be coordinated by an AI agent to provide effective customer service.



Code Listing

```
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
from llama_index.llms.ollama import Ollama
from llama_index.core import Settings
import nest_asyncio

nest_asyncio.apply()

# Set up your local LLM using Ollama.
Settings.llm = Ollama(
    model="qwq:32b",
    function_call_support=True,  # Enable function call support if
       available.
)

# For embeddings, we're using the HuggingFace model.
Settings.embed_model =
   HuggingFaceEmbedding(model_name="sentence-transformers
/all-MiniLM-L6-v2")

from typing import List

# Tool 1: Function that returns the list of items in an order
def get_order_items(order_id: int) -> List[str]:
    """Given an order Id, this function returns the list of items
       purchased for that order."""
    order_items = {
        1001: ["Laptop", "Mouse"],
        1002: ["Keyboard", "HDMI Cable"]
    }
    return order_items.get(order_id, [])
```

```
28
29 # Tool 2: Function that returns the delivery date for an order
30 def get_delivery_date(order_id: int) -> str:
31     """Given an order Id, this function returns the delivery date
       for that order."""
32     delivery_dates = {
33         1001: "10-Jun",
34         1002: "12-Jun",
35         1003: "08-Jun"
36     }
37     return delivery_dates.get(order_id, "")
38
39 # Tool 3: Function that returns maximum return days for an item
40 def get_item_return_days(item: str) -> int:
41     """Given an item, this function returns the return policy in
       number of days."""
42     item_returns = {
43         "Laptop": 30,
44         "Mouse": 15,
45         "Keyboard": 15,
46         "HDMI Cable": 5
47     }
48     return item_returns.get(item, 45)
49
50 # Tool 4: Vector DB that contains customer support contacts
51 from llama_index.core import SimpleDirectoryReader, VectorStoreIndex
52 from llama_index.core.node_parser import SentenceSplitter
53
54 # Load the customer service document.
55 support_docs = SimpleDirectoryReader(input_files=["Customer
       Service.pdf"]).load_data()
56
57 # Split the document into nodes/chunks.
58 splitter = SentenceSplitter(chunk_size=1024)
59 support_nodes = splitter.get_nodes_from_documents(support_docs)
60
61 # Build the vector index from the support nodes.
62 support_index = VectorStoreIndex(support_nodes)
63 support_query_engine = support_index.as_query_engine()
64
65 from llama_index.core.tools import FunctionTool, QueryEngineTool
66
67 # Wrap the functions as tools.
68 order_item_tool = FunctionTool.from_defaults(fn=get_order_items)
69 delivery_date_tool =
       FunctionTool.from_defaults(fn=get_delivery_date)
```

```
return_policy_tool =
    FunctionTool.from_defaults(fn=get_item_return_days)

# Create a query engine tool for the customer support vector index.
support_tool = QueryEngineTool.from_defaults(
    query_engine=support_query_engine,
    description="Customer support policies and contact information",
)

from llama_index.core.agent import FunctionCallingAgentWorker,
    AgentRunner

# Setup the Agent worker with all the tools using the local LLM
    (Settings.llm)
agent_worker = FunctionCallingAgentWorker.from_tools(
    [order_item_tool,
     delivery_date_tool,
     return_policy_tool,
     support_tool
    ],
    llm=Settings.llm,
    verbose=True
)

# Create an Agent Orchestrator with LlamaIndex
agent = AgentRunner(agent_worker)

# Get return policy for an order
response = agent.query("What is the return policy for order number
    1001")
print("\n Final output : \n", response)
```

Code & results are here