innovation lab

BY FETCH.AI

Fetch.ai
Innovation Lab

We lead the way in advancing artificial intelligence and driving innovation **to create value at scale.**

UNDERSTANDING AI AGENTS

Rajashekar Vennavelli

Preface

This guide draws upon insights from Google's work on AI agents, Anthropic's perspectives on building effective agents, and various other resources available on the internet. I'm particularly grateful to my friends and colleagues for their valuable review and insightful discussions that helped shape this document.

Disclaimer

This guide is not meant to be an authoritative document - it is intended to be a quick primer to help understand AI agents and build a simple AI agent. The definitions and concepts presented here are not universally accepted standards in the field. Nothing in this document should be considered exhaustive or complete, as the field of AI agents is rapidly evolving with new approaches and understandings emerging regularly.

The views, information and opinions presented in this document combine insights from various sources along with the author's own experiences and perspectives, and do not necessarily represent those of Fetch.ai. This document is not an official publication of Fetch.ai.

Purpose and Audience

This guide is a quick introduction to AI agents. It specifically aims to help:

- Students and developers new to the concept of AI agents
- Hackathon participants looking to build their first agents
- Developers seeking to understand core concepts without diving into theoretical depths

Through discussions at hackathons and developer meetups, I noticed common questions and misconceptions about AI agents. This guide aims to provide:

- A collection of fundamental concepts explained in simple terms
- Addressing common misconceptions
- Providing a clear path from understanding to implementation

Table of Contents

3.3 Sample Usage and Examples

- Example Queries and Responses
- Response Formats
- Interaction Examples

Conclusion

References

Appendix

Chapter 1 - Foundation and Core Concepts

1.1 Introduction to AI Agents

In the rapidly evolving landscape of artificial intelligence, AI agents represent a significant advancement over traditional software applications, offering more flexibility, adaptability, and intelligence in tackling complex tasks across various domains.

At their core, agents are software entities designed to perform autonomous actions by:

1. Observing their environment through various inputs (digital or physical)
2. Processing, analyzing, and reasoning about information using advanced algorithms and large language models
3. Making decisions and taking actions, often by leveraging external tools and APIs
4. Learning from outcomes and adapting their behavior over time
5. Utilizing memory to retain information and improve performance
6. Engaging in self-reflection, evaluation, and course correction

The Evolution of AI Systems

To understand AI agents, it's crucial to recognize the progression of AI systems:

1. **Traditional Applications**
    - Fixed logic and predefined rules
    - Limited or no adaptation
    - Direct input-to-output mapping
    - Example : Rule-based expert systems
2. **AI-Enhanced Applications**
    - Foundation model integration (LLMs, neural networks)
    - Task-specific intelligence, guided learning abilities, Human-directed operations
    - Limited context awareness
    - Example : Modern Chatbots, Specialized AI tools (image generators, code assistants)
3. **Agentic Systems**
    - Capable of taking autonomous decisions with or without human in the loop
    - Multi-step planning and execution
    - Dynamic tool discovery and usage, self-directed learning, continuous context awareness
    - Example : Operator released by OpenAI, Self-driving cars, Swarm robotics

Understanding System Types

AI systems can be categorized into two primary types:

1. **AI Workflows:** These are predefined sequences where LLMs and other tools are orchestrated using explicit code paths. They follow structured logic and operate with a defined start and end point.
2. **AI Agents:** These are more dynamic, allowing LLMs to take control of their processes and tool usage, making autonomous decisions on how to accomplish a task.

While the term "AI agent" is often used interchangeably, many practical applications don't need full agentic behavior. Instead, structured workflows are sufficient for most tasks, offering better control and predictability.

Agentic Workflow and Degree of Autonomy

Since full autonomy is neither possible (in majority of the systems) nor needed in most practical applications, the term 'Agentic Workflow' is gaining popularity as it combines the benefits of structured workflows with the flexibility of AI agents. This hybrid approach allows for more dynamic decision-making within a controlled framework, striking a balance between autonomy and predictability.

Agentic workflows represent a middle ground where AI agents operate within defined processes but have the ability to make decisions and adapt to changing circumstances. They leverage the strengths of both AI workflows and agents by:

1. Providing a structured sequence of tasks for consistency and control
2. Allowing AI agents to make autonomous decisions within these sequences
3. Enabling dynamic problem-solving and adaptation to complex scenarios
4. Maintaining oversight and predictability for critical business processes

This approach is particularly useful for tasks that require some level of flexibility but still need to operate within certain boundaries or comply with specific rules. As businesses seek to optimize their operations while managing risks, agentic workflows offer a practical solution that combines the efficiency of automation with the intelligence of AI agents.

The term 'AI agent' is widely used in the industry and by startups, often without a clear, universal definition. In practice, the autonomy of these so-called agents falls on a spectrum rather than being a binary classification. There is no definitive technical measure of autonomy, which leads to varying interpretations and implementations across different systems.

This spectrum of autonomy can range from:

1. Highly structured workflows with minimal decision-making capabilities
2. Semi-autonomous systems that can make decisions within predefined parameters
3. More flexible agents that can adapt their approach based on context
4. Highly autonomous systems that can formulate and pursue their own goals within a given domain

The degree of autonomy granted to an AI system often depends on factors such as:

- The complexity of the task
- The potential risks involved
- The need for human oversight
- The capabilities of the underlying AI technologies

As the field evolves, we may see more standardized ways to measure and describe the level of autonomy in AI systems. For now, it's important to understand that when someone claims to use 'AI agents', the actual level of autonomy can vary significantly, and it's crucial to delve deeper into the specific capabilities and limitations of each system.

This nuanced understanding of autonomy reinforces the value of agentic workflows, as they offer a flexible framework that can accommodate various degrees of AI decision-making while maintaining necessary control structures.

Three Pillars of Agentic Workflows

The effectiveness of agentic workflows is based on three key elements.

1. **Autonomy**: Handling tasks with minimal human input.
2. **Adaptability**: Adjusting to unique business needs and changing conditions.
3. **Optimization**: Continuously improving through machine learning.

Implementation Challenges

While the benefits are significant, it's important to note that implementing and managing these workflows can be complex. This complexity reinforces the need for a nuanced approach to autonomy and careful consideration of the specific use case and organizational context.

AI Workflow

- Combine LLMs with predefined processes
- Follow structured but flexible paths
- Best for: Complex but predictable tasks where:
  - Process steps are well-understood and consistent
  - Predictability is more important than flexibility
  - You need tight control over execution
  - Performance and reliability are crucial
- Example:

```
# AI workflow example
class StockAnalysisWorkflow:
  def analyze(self, stock_symbol):
      # Uses LLM but in FIXED order:
      # Always: price → news → recommendation
      # Can't skip steps or change order

      price_analysis = llm.analyze(f"Analyze {stock_symbol} price
trends")
      news_analysis = llm.analyze(f"Analyze {stock_symbol} recent news")
      return llm.recommend(price_analysis, news_analysis)
```

AI Agents

- Dynamically direct their own processes
- Maintain control over task execution
- Best for: Dynamic planning, adaptive execution, goal-oriented behavior where:
  - Tasks require dynamic decision-making
  - Problems have multiple valid solution paths
  - Flexibility and adaptation are crucial
  - Complex tool interactions are needed
  - Tasks benefit from maintaining context
- Example:

```
# AI agent example
class StockAnalysisAgent:
   def analyze(self, stock_symbol):
       # LLM DECIDES everything:
       # - What to analyze first (price? news? competitors?)
```

```
        # - Whether to dig deeper into any area
        # - When analysis is sufficient
        # Can adapt strategy based on what it finds

        strategy = llm.decide(f"How should we analyze {stock_symbol}?")
        while not self.analysis_complete():
            next_step = llm.decide("What should we analyze next?")
            findings = self.execute_step(next_step)
            if findings.need_different_approach:
                strategy = llm.revise_strategy(findings)
```

The core difference is that AI agents have:

Genuine Autonomy

- Not just following predefined steps with decision points
- Actually reasoning about what actions to take
- Ability to discover and adapt strategies

Strategic Flexibility

- Can handle unexpected situations
- Doesn't just choose from predefined options
- Creates novel approaches to problems

Contextual Understanding

- Understands the implications of its actions
- Can reason about tool capabilities
- Maintains meaningful context about its goals and progress

Dynamic Goal Management

- Can reformulate goals when needed
- Understands when to abandon or modify objectives
- Can handle competing or conflicting goals

Core Components of an Agent

A generic agent architecture, consists of several key components:

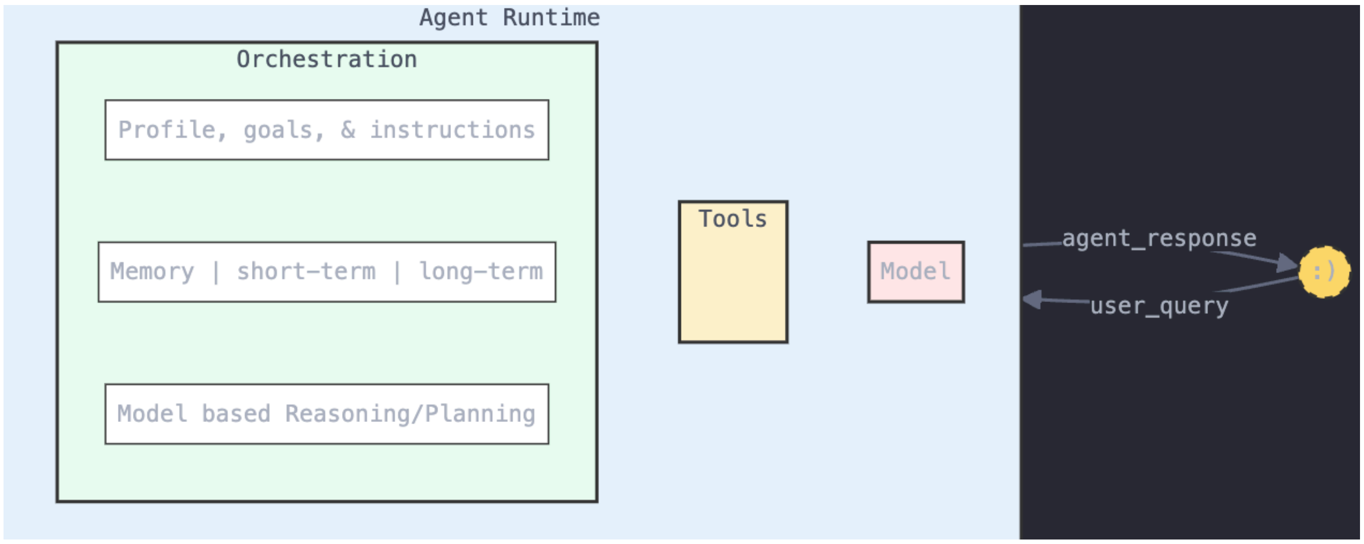

Agent Runtime
Orchestration
Profile, goals, & instructions
Memory | short-term | long-term
Model based Reasoning/Planning
Tools
Model
agent_response
user_query
:)

Figure 1: Core components of an AI Agent

1. Model Layer

- Central decision-making engine
- Processes input and context
- Generates reasoning and plans

2. Orchestration Layer

- Manages the execution flow
- Coordinates tool usage
- Monitors progress towards goals

3. Memory System

- Short-term working memory
- Long-term knowledge storage
- Context retention

4. Tool Integration

- External API connections
- Data processing capabilities
- Action execution interfaces

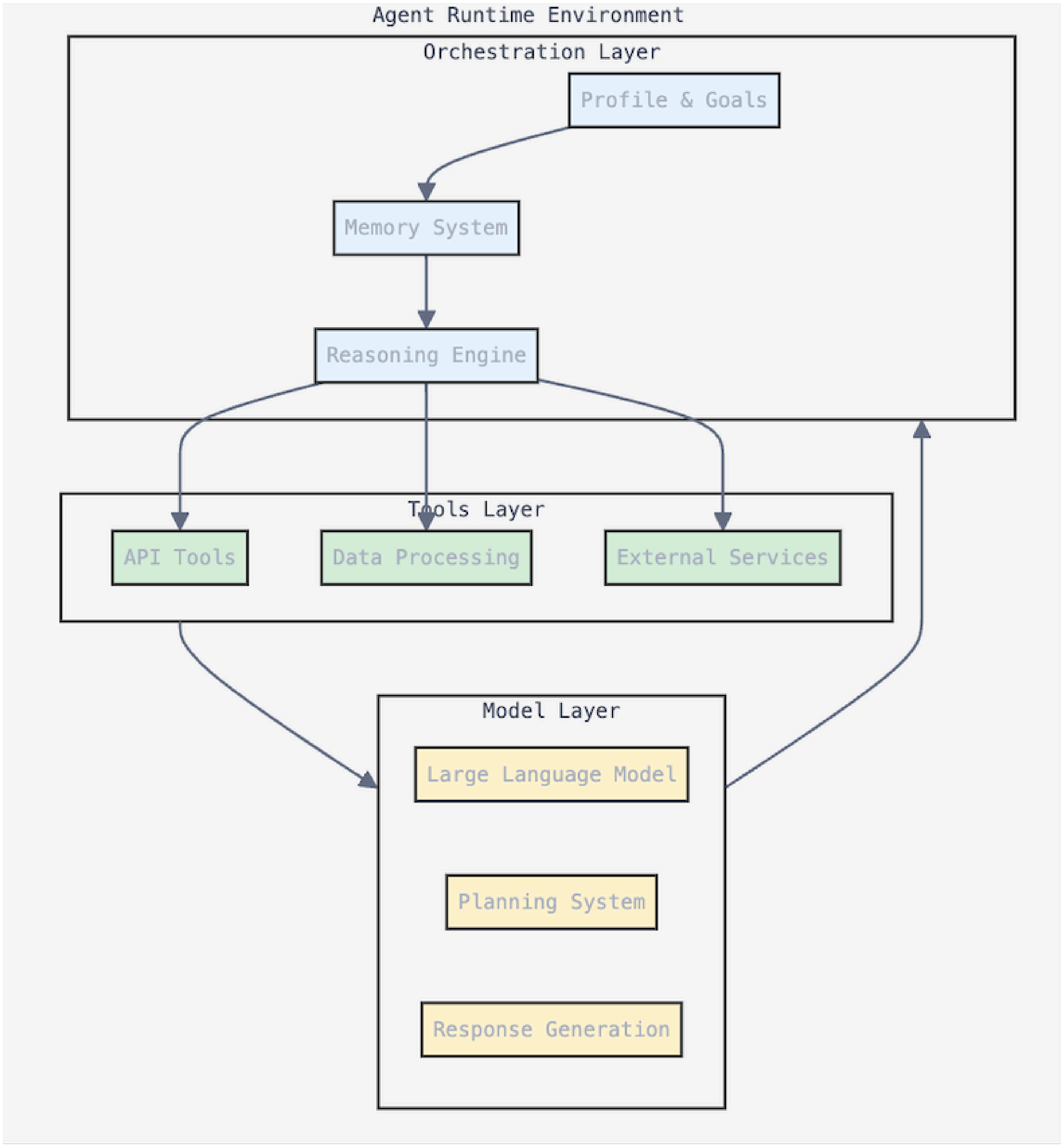


Figure 2: Agent Runtime Environment

This diagram shows a more detailed "Agent Runtime Environment" with three main layers that work together:

Orchestration Layer (Top)

- Contains the high-level control components:
  - Profile & Goals: Defines the agent's objectives and constraints
  - Memory System: Implements both short and long-term memory
  - Reasoning Engine: Coordinates decision-making and orchestrates the flow between components

Tools Layer (Middle)

- Breaks down tool integration into three specific categories:
  - API Tools: Interfaces for external API connections
  - Data Processing: Tools for handling and transforming data
  - External Services: Integration with third-party services
- This layer implements the "Tool Integration" component from the core architecture

Model Layer (Bottom)

- Contains three specialized components:
  - Large Language Model: The foundation model for understanding and generation
  - Planning System: Handles task decomposition and strategy
  - Response Generation: Manages the creation of outputs

The arrows in the diagram show the information flow:

- The Orchestration Layer controls the overall process flow
- The Tools Layer acts as an intermediary between orchestration and models
- There's a feedback loop from the Model Layer back to the Orchestration Layer, showing how the system can iteratively refine its responses

This implementation provides more concrete details about how the four core components (Model, Orchestration, Memory, and Tools) work together in a practical system.

General Criteria for Agency

Before implementing an agent, evaluate if your system really needs true agency by checking these criteria:

Decision Autonomy

- Can the system choose different paths based on context?
- Does it make meaningful decisions about tool usage?
- Can it adapt its strategy during execution?

State Management

- Does it maintain meaningful state?
- Can it use past interactions to inform decisions?
- Does it track progress toward goals?

Tool Integration

- Can it choose tools dynamically?
- Does it understand tool capabilities?
- Can it combine tools in novel ways?

Goal Orientation

- Does it understand and work toward specific objectives?
- Can it recognize when goals are achieved?
- Can it adjust goals based on new information?

ReAct Pattern

The simplest AI agents typically operate using the ReAct (Reason-Action) framework, which follows a cyclical pattern. ReAct is an iterative approach that alternates between thinking and acting, combining the reasoning capabilities of large language models (LLMs) with the ability to interact with external tools and environments. The core workflow includes:

1. **Reasoning (Thought)**: The agent analyzes the current state, objectives, and available information.
2. **Acting (Action)**: Based on its reasoning, the agent executes specific operations or uses tools.
3. **Observation**: The agent obtains results from its actions.

4. **Iteration**: The cycle continues, with the agent thinking and acting based on new observations until reaching a final answer.

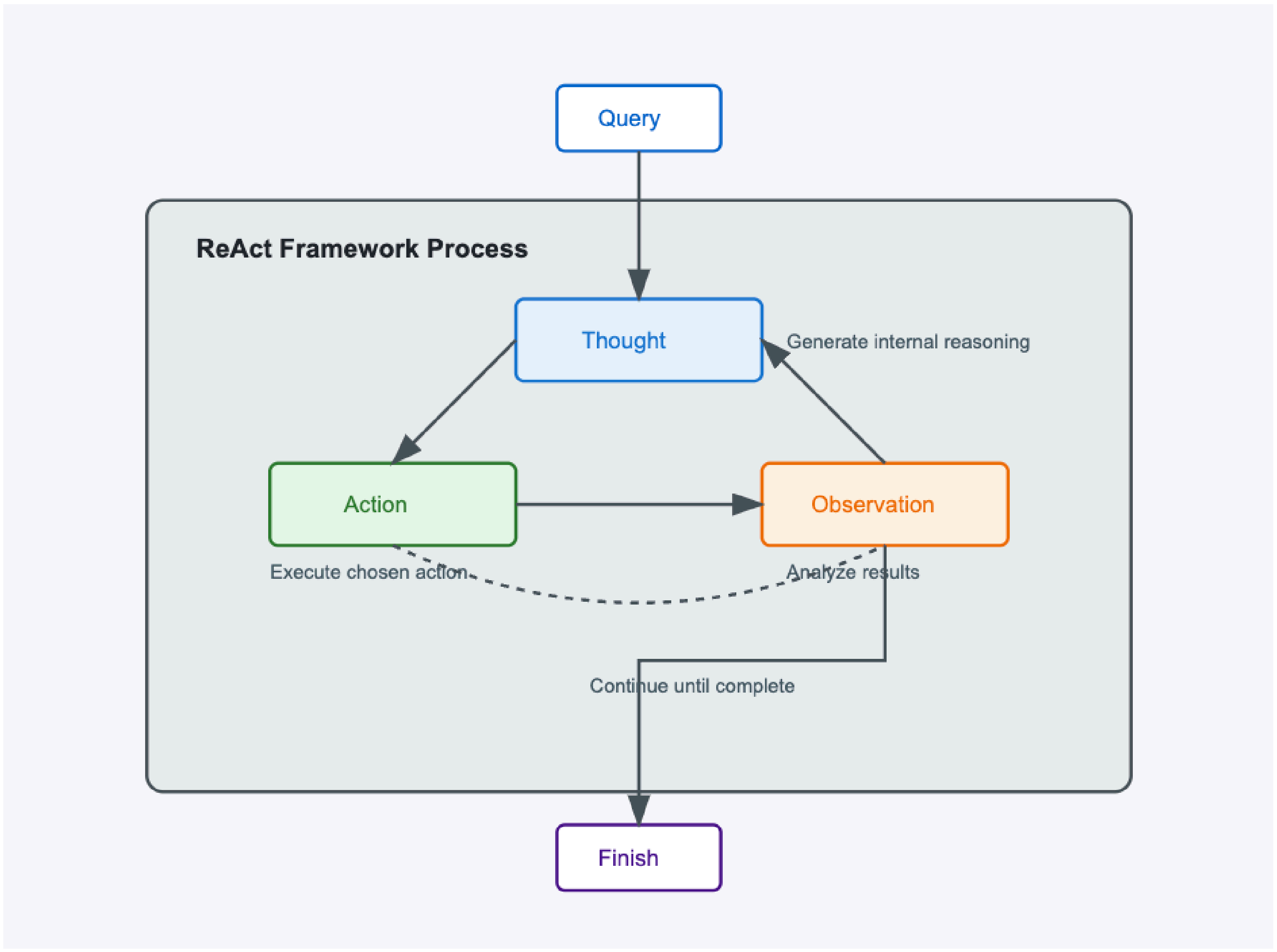


Figure 3: ReAct Pattern

Key Components:

1. **Thought**:
   - Internal reasoning about the current state and objectives
   - Analysis of available information
   - Planning next steps and formulating strategies
2. **Action**:
   - Execution of chosen steps
   - Tool usage and integration (e.g., calculators, search engines, APIs)
   - Interaction with external systems or environments
3. **Observation**:
   - Gathering results from actions
   - Analyzing outcomes
   - Updating understanding and knowledge base

4. **Iteration**:
    - Continuous loop of Thought-Action-Observation
    - Dynamic adjustment of plans based on new information
    - Progress towards final goal or answer

Implementation and Best Practices:

1. **Prompt Engineering**: Craft a clear system prompt that defines the agent's behavior and available tools.
2. **Tool Integration**: Provide the agent with access to relevant external tools and APIs to expand its capabilities.
3. **Memory Management**: Implement a mechanism for the agent to retain and utilize information from previous steps.
4. **Error Handling**: Design the system to gracefully handle unexpected inputs or tool failures.
5. **Performance Optimization**: Balance the number of reasoning steps with action execution to maintain efficiency.

Advantages of ReAct:

- Combines internal knowledge with external information gathering
- Enables complex problem-solving through iterative reasoning and action
- Improves transparency and interpretability of AI decision-making
- Allows for dynamic adaptation to new information and changing scenarios

Applications:

ReAct has shown promise in various domains, including:

- Question answering systems
- Task planning and execution
- Data analysis and interpretation
- Decision-making in complex environments

By implementing the ReAct pattern, developers can create more versatile and capable AI agents that can handle a wide range of tasks requiring both reasoning and interaction with external resources.

Part 1.2 - Common Misconceptions

The term "agent" in AI is overused and often applied inconsistently, diminishing its meaning and creating confusion about the actual capabilities of AI systems. Some of the common misconceptions are explained below:

Misconception 1: "My application uses multiple LLM calls, so it's an agent"

```python
# This is NOT an agent - it's a multi-step LLM application
def process_document(doc):
    summary = llm.generate_summary(doc)
    keywords = llm.extract_keywords(summary)
    sentiment = llm.analyze_sentiment(summary)
    return {
        "summary": summary,
        "keywords": keywords,
        "sentiment": sentiment
    }
```

Misconception 2: "I'm using tools and APIs, so it's an agent"

```python
# This is NOT an agent - it's an automated workflow
def analyze_stock(symbol):
    price_data = stock_api.get_price(symbol)
    news = news_api.get_recent_news(symbol)
    analysis = llm.analyze(f"Price: {price_data}, News: {news}")
    return analysis
```

Misconception 3: "Having memory makes it an agent"

```python
# This is NOT an agent - just stateful LLM interaction
class ChatSystem:
    def __init__(self):
        self.conversation_history = []

    def respond(self, user_input):
        self.conversation_history.append(user_input)
        response = llm.generate(context=self.conversation_history)
        self.conversation_history.append(response)
        return response
```

Misconception 4: "Using planning means its an agent"

```python
# This is NOT an agent - it's structured task decomposition
def handle_task(task):
    # Fixed planning template
    steps = llm.break_down_task(task)
    results = []
    for step in steps:
        result = execute_step(step)
        results.append(result)
    return combine_results(results)
```

Misconception 4: "Complex prompt engineering makes it an agent"

```python
# This is NOT an agent - just sophisticated prompting
def analyze_with_cot(query):
    prompt = f"""
    Step 1: Understand the query
    {query}
    Step 2: Break down the components
    Step 3: Analyze each component
    Step 4: Synthesize findings
    """
    return llm.generate(prompt)
```

Misconception 5: "Having a feedback loop makes it an agent"

```python
# This is NOT an agent - just iterative refinement
def iterative_response(query, max_iterations=3):
    response = initial_response(query)
    for _ in range(max_iterations):
        quality = evaluate_response(response)
        if quality > threshold:
            break
        response = improve_response(response)
    return response
```

Misconception 6: "The LLM performs the actions in an agent"

```python
# Common MISCONCEPTION: People think this actually performs actions
def incorrect_understanding():
    llm_response = llm.generate("Please save this file to disk")
    # The LLM can't actually save files!

# REALITY: Tools perform actions, LLM orchestrates
class PropertyAgent:
    def __init__(self):
        self.tools = {
            'database': DatabaseTool(),
            'email': EmailTool(),
            'calendar': CalendarTool()
        }

    def handle_request(self, query):
        # LLM determines what needs to be done
        action_plan = llm.plan_actions(query)

        # TOOLS actually perform the actions
        for action in action_plan:
            if action.type == "schedule_viewing":
                # Calendar tool performs the actual scheduling
                self.tools['calendar'].create_appointment(action.details)
            elif action.type == "send_confirmation":
                # Email tool performs the actual sending
                self.tools['email'].send_message(action.details)
```

Key Points About LLM's Role

1. **LLM's Actual Functions:**
   - Planning and strategizing actions
   - Reasoning about which tools to use
   - Interpreting results from tools
   - Generating natural language responses

2. **Tools' Actual Functions:**
    - File operations
    - Database queries
    - API calls
    - Network requests
    - System modifications
    - Real-world interactions

```python
# Clear separation of responsibilities
class AgentSystem:
    def process_task(self, task):
        # LLM PLANS the action
        plan = self.llm.create_execution_plan(task)

        # TOOLS EXECUTE the action
        for step in plan:
            if step.requires_web_access:
                result = self.web_tool.fetch_data(step.url)
            elif step.requires_database:
                result = self.db_tool.query(step.sql)
            elif step.requires_file_operation:
                result = self.file_tool.process(step.path)

            # LLM INTERPRETS results and plans next steps
            next_actions = self.llm.analyze_results(result)
```

This misconception is particularly important because it helps explain:

- Why tool integration is crucial for practical agent systems
- Why agents need careful permission and capability management
- The importance of proper tool abstraction and safety measures
- Why LLM responses alone can't perform real-world actions

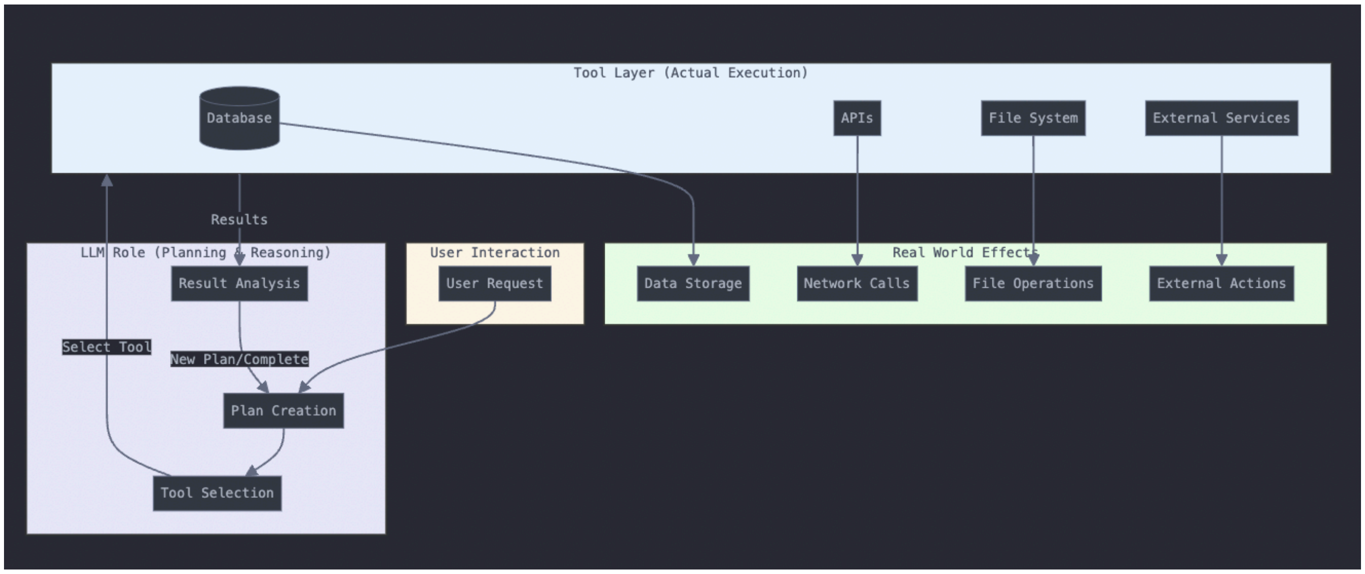


Figure 4: LLM vs Tools: Planning and Execution Flow

This diagram illustrate several key points:

1. **Separation of Responsibilities**
    - LLM handles planning, reasoning, and decision-making
    - Tools perform actual real-world actions
    - Clear boundaries between thinking and doing
2. **Flow of Control**
    - User requests flow through the LLM first
    - LLM determines which tools to use
    - Tools execute actions and return results
    - LLM interprets results and plans next steps
3. **Real World Impact**
    - Only tools can affect the external world
    - LLM provides intelligence but not execution
    - Actions are constrained by available tools

This helps explain why:

- Tool integration is crucial for practical agent systems
- Security and permissions must be implemented at the tool level
- LLM capabilities alone don't enable real-world actions
- System design must carefully consider tool access and limitations

Misconception 7:"My AI Assistant/AI chatbot is an AI Agent"

```
# This is NOT an agent - it's an AI Assistant
class BasicAIAssistant:
    def chat(self, user_input):
        response = llm.generate_response(user_input)
        return response

# This is CLOSER to an agent
class AIAgent:
    def __init__(self):
        self.tools = load_available_tools()
        self.memory = AgentMemory()
        self.planner = ActionPlanner()

    def handle_task(self, task):
        # Autonomous decision making
        goal = self.planner.define_goal(task)
        plan = self.planner.create_plan(goal)

        # Dynamic tool selection and execution
        while not goal.is_achieved():
            next_action = self.planner.next_action(plan)
            tool = self.select_tool(next_action)
            result = tool.execute(next_action.parameters)

            # Adaptive behavior
            if not result.is_successful():
                plan = self.planner.revise_plan(result)

            self.memory.update(result)
```

However, sometimes as we discussed in chapter 1, AI assistants could have certain level of Agentic behavior depending on how they are implemented.

Key Differences:

1. **Autonomy Level**
    - Assistant: Responds to direct commands and questions
    - Agent: Makes autonomous decisions about how to achieve goals
2. **Tool Usage**
    - Assistant: May have access to tools but uses them as instructed
    - Agent: Autonomously decides which tools to use and when

3. **Goal Orientation**
    - Assistant: Focuses on responding to immediate requests
    - Agent: Maintains and works toward longer-term goals
4. **Memory Usage**
    - Assistant: May maintain conversation history
    - Agent: Uses memory strategically for goal achievement
5. **Decision Making**
    - Assistant: Makes limited decisions within conversation scope
    - Agent: Makes complex decisions about actions, strategy, and resource use

Example Task Comparison:

```python
# Research Task Example

# AI Assistant Approach:
async def assistant_research(query):
    """Responds to direct questions with available information"""
    response = await llm.generate(
        f"Please research about {query}"
    )
    return response

# AI Agent Approach:
async def agent_research(query):
    """Autonomously conducts comprehensive research"""
    plan = await self.create_research_plan(query)
    sources = []

    for step in plan:
        if step.type == "web_search":
            results = await self.tools.search(step.query)
            sources.extend(results)
        elif step.type == "verify_information":
            verified_data = await self.tools.fact_check(results)
        elif step.type == "synthesize":
            synthesis = await self.tools.analyze(verified_data)

        # Adaptive planning
        if self.evaluate_progress() < self.quality_threshold:
            plan = await self.revise_research_plan()

    return self.compile_research(sources, synthesis)
```

This misconception is particularly important because:

1. It affects system design expectations
2. It influences how we evaluate AI system capabilities
3. It impacts how we implement security and permissions
4. It shapes user expectations and interaction patterns

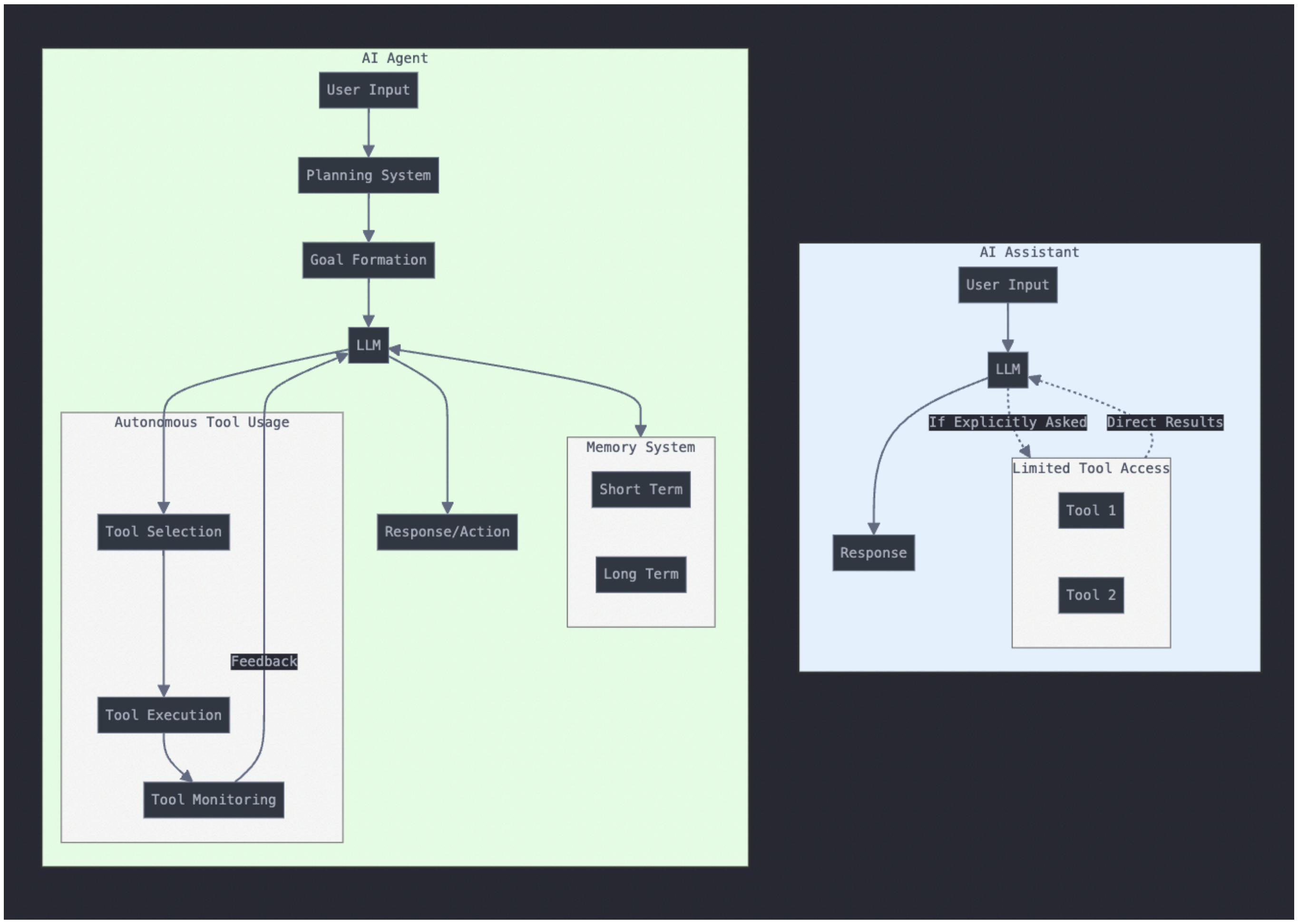


Figure 5: AI Agent Vs. AI Assistant

These diagrams highlight the key differences between AI Assistants and AI Agents:

1. **Architecture Complexity**
   - Assistant: Simple, linear flow with reactive tool usage
   - Agent: Complex system with multiple interacting components
2. **Processing Flow**
   - Assistant: Direct input → response pattern
   - Agent: Multi-step process with planning and feedback loops
3. **Tool Integration**
   - Assistant: Passive, explicitly requested tool usage
   - Agent: Active, autonomous tool selection and execution

4. **Memory Usage**
    - Assistant: Basic conversation tracking
    - Agent: Sophisticated memory system for context and learning
5. **Decision Making**
    - Assistant: Reactive decisions based on immediate input
    - Agent: Proactive decisions based on goals and strategy

Part 1.3 - Building a Multi-Agent System

Multi-agent Systems

An agent uses an LLM to control application flow. However, as systems grow complex, using a single agent can become challenging. This is where multi-agent systems come in.

Why Use Multiple Agents?

- **Simplicity**: Break complex tasks into manageable pieces
- **Expertise**: Create specialized agents for specific tasks
- **Better Control**: Manage how agents work together

Common Multi-agent Patterns

1. **Network Pattern**
    - Agents can communicate freely with every other agent
    - Any agent can decide which other agent to call next
    - Flexible but potentially complex to manage
2. **Supervisor Pattern**
    - Central supervisor coordinates other agents
    - Clear control flow through supervisor
    - Better oversight and management
    - Can be implemented through tool-calling
3. **Hierarchical Pattern**
    - Supervisors of supervisors
    - Allows for more complex control flows
    - Suitable for large-scale systems
4. **Custom Workflow Pattern**
    - Agents communicate with specific subset of agents
    - Parts of flow are deterministic
    - Limited decision-making about which agents to call next

We will build our agent using a supervisor based multi-agent pattern.

Supervisor-Based Agent Architecture

A team-based architecture typically consists of three main components:

1. **Supervisor Agent**
   - Coordinates team activities
   - Makes routing decisions
   - Ensures task completion
2. **Specialist Agents**
   - Handle domain-specific tasks
   - Maintain focused expertise
   - Provide detailed analysis
3. **State Management System**
   - Maintains conversation context
   - Tracks team progress
   - Manages shared resources

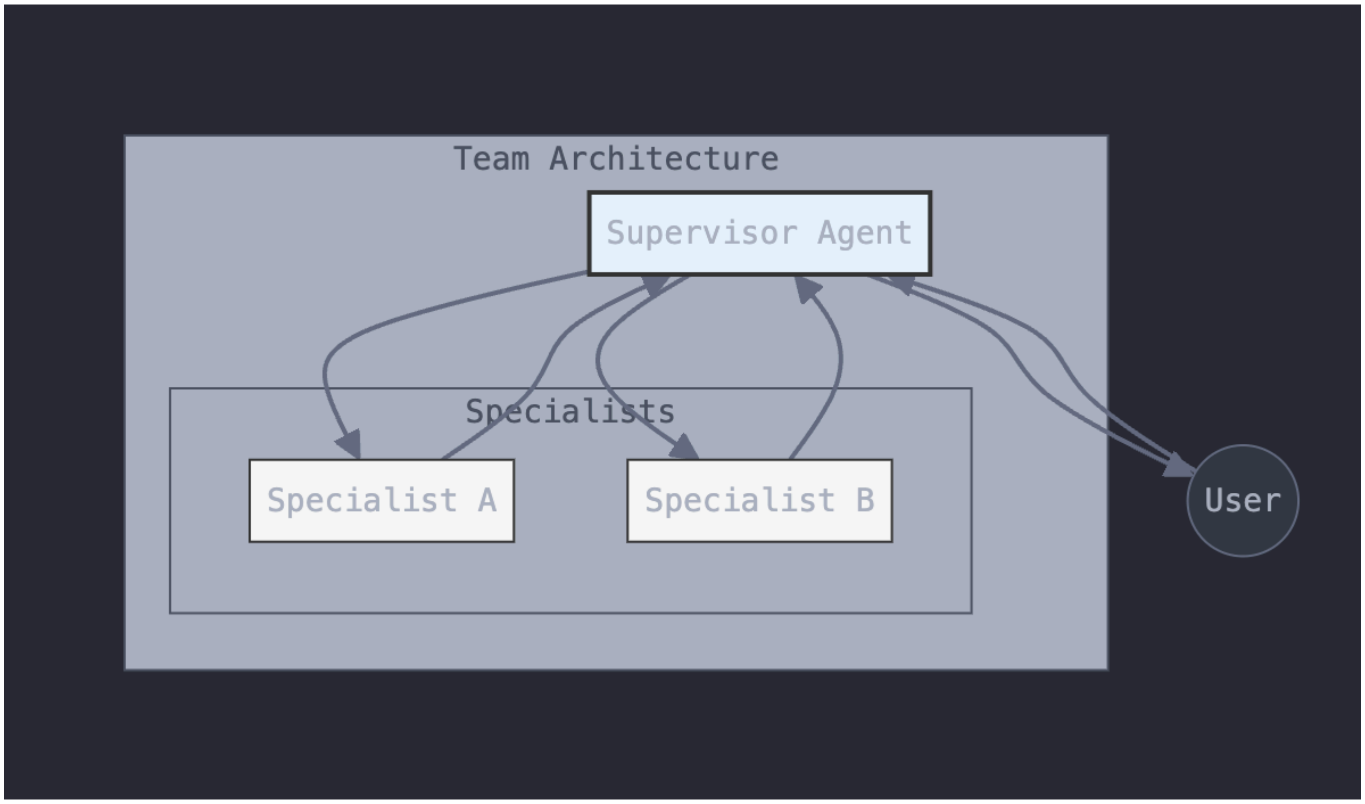


Figure 6: Supervisor-Based Multi-Agent Architecture

Implementation Pattern using Langgraph

The code implements a team-based architecture using langgraph with three essential components that work together to process complex tasks:

1. Supervisor Agent

```
def create_supervisor_agent(llm: ChatOpenAI):
    """Creates a supervisor agent that orchestrates the team's activities.

    The supervisor agent is the core decision-maker that:
    1. Analyzes incoming queries to understand requirements
    2. Determines which specialist agent is best suited for each subtask
    3. Routes tasks to appropriate specialists
    4. Monitors the overall progress of the task
    5. Decides when enough information has been gathered

    Args:
        llm: Language model for decision making

    Returns:
        A supervisor agent configured with team coordination capabilities
    """
    supervisor_prompt = """
    Core responsibilities:
    1. Analyze incoming queries to break down complex tasks
    2. Determine what specific information is needed
    3. Select the most appropriate specialist for each subtask
    4. Monitor progress and ensure task completion
    5. Decide when sufficient information has been gathered
    """

    return create_team_supervisor(
        llm=llm,
        system_prompt=supervisor_prompt,
        members=["SpecialistA", "SpecialistB"]
    )
```

2. Specialist Agents

```
def create_specialist_agent(
    llm: ChatOpenAI,
    domain: str,
    tools: List[Tool]
):
    """Creates a specialist agent with domain-specific expertise.

    Specialist agents are focused experts that:
    1. Handle specific types of tasks within their domain
    2. Use specialized tools for their domain
    3. Provide structured analysis and insights
    4. Request clarification when needed

    Args:
        llm: Language model for domain-specific processing
        domain: Area of expertise (e.g., "financial analysis", "market
research")
        tools: List of domain-specific tools available to this specialist

    Returns:
        A specialist agent configured for its specific domain
    """
    system_prompt = f"""
    You are specialized in {domain}.
    When responding:
    1. Use your domain-specific tools effectively
    2. Provide clearly structured outputs
    3. Explicitly request any missing information needed
    """

    return create_agent(
        llm=llm,
        tools=tools,
        system_prompt=system_prompt
    )
```

3. State Management

```
class TeamState(TypedDict):
    """Manages the shared state and context for the entire team.

    Attributes:
        messages: List of all messages in the conversation history
        team_members: List of available specialist agents
        next: Identifier of the next agent to act
        information_needed: List of missing information to be gathered
        reasoning: Explanation for the current decision or action
    """
    messages: List[BaseMessage]
    team_members: List[str]
    next: str
    information_needed: List[str]
    reasoning: str
```

Team Graph Implementation

The team graph orchestrates how all components work together:

```
def create_team_graph():
    """Creates a coordinated team of agents with defined interaction
patterns.

    The graph defines:
    1. How agents communicate with each other
    2. The flow of information between agents
    3. Decision points for task routing
    4. Conditions for task completion

    Process Flow:
    1. Supervisor receives task and analyzes requirements
    2. Tasks are routed to appropriate specialists
    3. Specialists process tasks and return results
    4. Supervisor evaluates results and decides next steps
    5. Process continues until task is complete

    Returns:
        A compiled graph ready for task processing
    """
    # Initialize team members
    specialist_a = create_specialist_agent(...)
    specialist_b = create_specialist_agent(...)
    supervisor = create_supervisor_agent(...)

    # Create the coordination graph
    graph = StateGraph(TeamState)

    # Define team structure
    graph.add_node("SpecialistA", specialist_a)
    graph.add_node("SpecialistB", specialist_b)
    graph.add_node("supervisor", supervisor)
    # Define information flow
    graph.add_edge("SpecialistA", "supervisor")
    graph.add_edge("SpecialistB", "supervisor")

    # Set up decision routing
    graph.add_conditional_edges(
        "supervisor",
        lambda x: x["next"],
        {
            "SpecialistA": "SpecialistA",
            "SpecialistB": "SpecialistB",
            "FINISH": END
        }
    )

    return graph.compile()
```

Chapter 2 - Building Your Own Financial Analysis AI Agent

2.1 Core Systems and Architecture

System Overview

The Financial Analysis Agent is an intelligent system designed to analyze financial information using multiple specialized agents with direct tool integration through the LangChain framework. The system combines SEC filing analysis with real-time market data to provide comprehensive financial insights.

Architecture Diagram

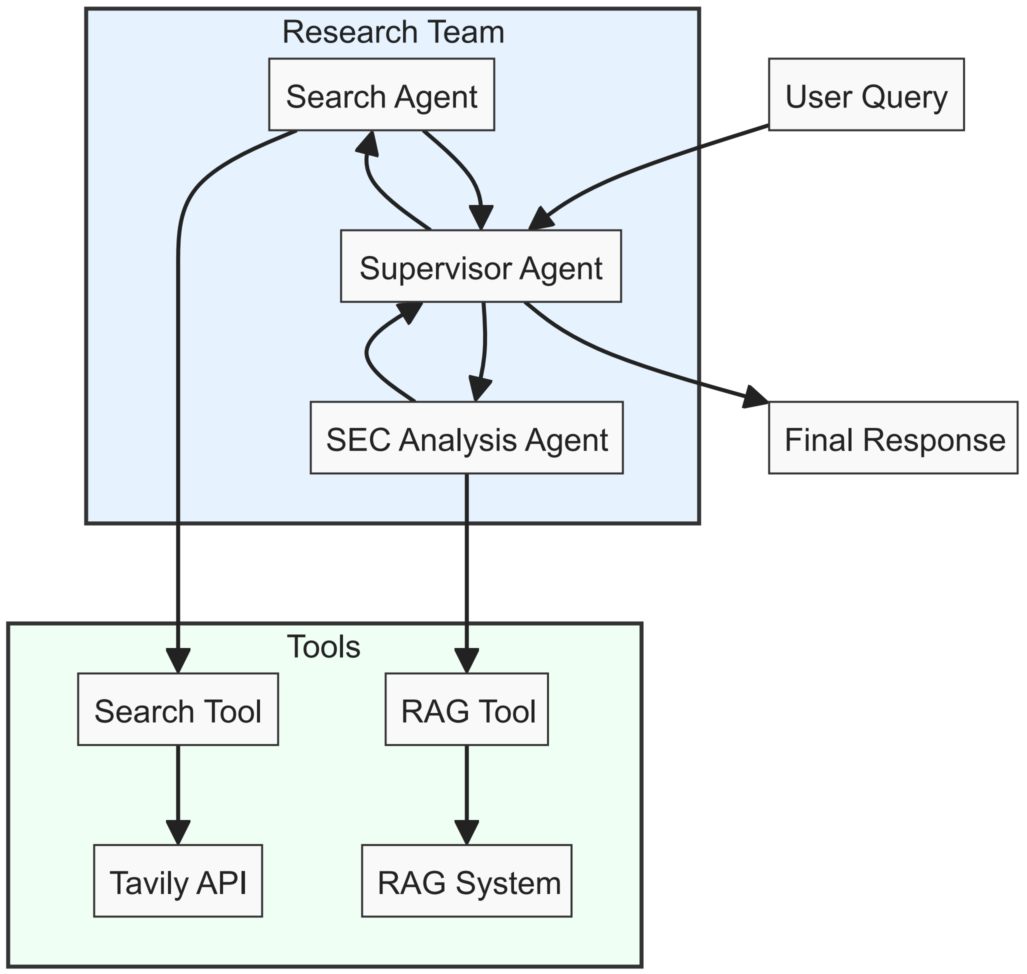


Figure 7: Financial Analysis Agent Architecture

Project Structure

```
src/
├── agents/
│   ├── __init__.py
│   ├── search_agent.py      # Search specialist using Tavily
│   ├── sec_agent.py         # SEC specialist using RAG
│   └── supervisor.py        # Team coordinator
├── tools/
│   ├── __init__.py
│   ├── search.py            # Tavily tool implementation
│   └── analysis.py          # RAG tool implementation
├── rag/
│   ├── __init__.py
│   ├── chain.py            # RAG chain implementation
│   └── loader.py           # Document loading utilities
├── graph/
│   ├── __init__.py
│   └── state.py            # Research team state management
├── utils/
    ├── __init__.py
    └── helpers.py          # Helper functions for agents
```

Core Components

A. Research Team State

```python
class ResearchTeamState(TypedDict):
    messages: List[BaseMessage]       # Conversation history
    team_members: List[str]           # Active team members
    next: str                         # Next agent to act
    information_needed: List[str]     # Required information
    reasoning: str                    # Decision reasoning
```

B. Specialized Agents

1. **SEC Analysis Agent**
   - Purpose: Analyzes SEC filings and financial documents
   - Key Features:

    - Uses RAG system directly through LangChain tool
    - Focused on historical financial data
    - Processes regulatory filings
- Implementation:

```python
from ..tools.analysis import retrieve_information

def create_sec_agent(llm: ChatOpenAI):
    system_prompt = """You are a financial analyst specialized in 
SEC filings...."""
    return create_agent(
        llm=llm,
        tools=[retrieve_information],
        system_prompt=system_prompt
    )
```

2. **Search Agent**
    - Purpose: Gathers real-time market information
    - Key Features:
        - Uses Tavily search directly through LangChain tool
        - Focuses on current market data
        - Retrieves analyst opinions
    - Implementation:

```python
from ..tools.search import tavily_search

def create_search_agent(llm: ChatOpenAI):
    system_prompt = """You are a research assistant..."""
    return create_agent(
        llm=llm,
        tools=[tavily_search],
        system_prompt=system_prompt
    )
```

2.2 Tool Implementation and Integration

1. Direct Tools

A. Tavily Search Tool

```python
# src/tools/search.py
from typing import Annotated
from langchain_core.tools import tool
from tavily import TavilyClient
import os
from dotenv import load_dotenv

load_dotenv()

tavily_client = TavilyClient(api_key=os.getenv("TAVILY_API_KEY"))

@tool
def tavily_search(query: str) -> str:
    """Search for real-time information using Tavily."""
    try:
        result = tavily_client.search(query)
        return str(result)
    except Exception as e:
        return f"Error performing search: {str(e)}"
```

B. RAG Analysis Tool

```python
# src/tools/analysis.py
from typing import Annotated
from langchain_core.tools import tool
from ..rag.chain import create_rag_chain

rag_chain = None

@tool
def retrieve_information(query: Annotated[str, "query to analyze financial
documents"]) -> str:
    """Use RAG to get specific information from financial documents."""
    try:
        global rag_chain
        if rag_chain is None:
            rag_chain = create_rag_chain()
        return rag_chain.invoke(query)
    except Exception as e:
        return f"Error analyzing documents: {str(e)}"
```

2. RAG Implementation

A. Document Processing

```python
# src/rag/loader.py
class DocumentLoader:
    def __init__(self, file_path: str):
        self.file_path = file_path

    @staticmethod
    def tiktoken_len(text):
        tokens = tiktoken.encoding_for_model("gpt-4").encode(text)
        return len(tokens)

    def load_and_split(self):
        # Load document using PyMuPDF
        docs = PyMuPDFLoader(self.file_path).load()

        # Split into chunks
        text_splitter = RecursiveCharacterTextSplitter(
            chunk_size=300,
            chunk_overlap=0,
            length_function=self.tiktoken_len
        )
        return text_splitter.split_documents(docs)
```

B. RAG Chain Setup

```python
# src/rag/chain.py
def create_rag_chain(file_path: str = "data/raw/apple_10k.pdf"):
    """Create RAG chain for document analysis."""
    # Load and split document
    loader = DocumentLoader(file_path)
    split_chunks = loader.load_and_split()

    # Create embeddings and vectorstore
    embedding_model = OpenAIEmbeddings(model="text-embedding-3-small")
    vectorstore = Qdrant.from_documents(
        split_chunks,
        embedding_model,
        location=":memory:",
        collection_name="sec_filings"
    )

    # Create retriever
    retriever = vectorstore.as_retriever()

    # Create prompt
    template = """Use the provided context to answer questions about the
company's financials.

    Context: {context}
    Question: {question}

    Answer the question based on the context provided. Include specific
numbers and data when available."""

    prompt = ChatPromptTemplate.from_template(template)

    # Create chain
    chain = (
        {"context": retriever, "question": RunnablePassthrough()}
        | prompt
        | ChatOpenAI(model="gpt-4-turbo-preview")
        | StrOutputParser()
    )

    return chain
```

3. Tool Usage in Agents

A. Search Agent Integration

```
# src/agents/search_agent.py
def create_search_agent(llm: ChatOpenAI):
    """Creates a search agent specialized in market research."""

    system_prompt = """You are a research assistant who can search for up-
to-date
    financial information using the tavily search engine.

    When responding:
    1. Always cite sources
    2. Focus on recent market data and analyst reports
    3. If SEC data is mentioned, compare it with current market views
    4. Highlight any significant discrepancies with official filings
    Format your response as:
    1. Market Data: [your findings]
    2. Analyst Views: [key opinions]
    3. Relevance to SEC Data: [if applicable]
    """

    return create_agent(
        llm=llm,
        tools=[tavily_search],
        system_prompt=system_prompt
    )
```

B. SEC Analyst Integration

```
# src/agents/sec_agent.py
def create_sec_agent(llm: ChatOpenAI):
    """Creates an agent specialized in SEC filings analysis."""

    system_prompt = """You are a financial analyst specialized in SEC
filings analysis.
    After analyzing SEC filings:
    1. If you need market context, clearly state what specific market data
you need
    2. If numbers need industry comparison, explicitly request competitor
data
    3. Always include specific numbers and trends from the filings
    4. If you spot significant changes or unusual patterns, highlight them

    Format your response as:
    1. Data from SEC Filings: [your findings]
    2. Additional Context Needed: [if any]
    3. Analysis: [your insights]
    """

    return create_agent(
        llm=llm,
        tools=[retrieve_information],
        system_prompt=system_prompt
    )
```

2.3 State Management and Helper Functions

1. State Management

What is State?

State in our system functions like a brain's working memory. It's the central system that keeps track of all vital information during the analysis process. Think of it as a group project manager that:

- Records and maintains the entire conversation history
- Determines which agent should work next
- Tracks what information is still missing
- Documents the reasoning behind decisions

State Structure Explained

```python
from typing import TypedDict, List, Annotated
from langchain_core.messages import BaseMessage
import operator

class ResearchTeamState(TypedDict):
    """Define the state structure for the research team."""
    messages: Annotated[List[BaseMessage], operator.add]  # Conversation
history
    team_members: List[str]                              # Available agents
    next: str                                           # Next agent to act
    information_needed: List[str]                       # Required
information
    reasoning: str                                      # Decision reasoning
```

Each component serves a specific purpose:

1. **messages**:
    - Functions like a conversation transcript
    - Maintains chronological order of all interactions
    - Provides context for agents to understand previous discussions
    - Example: If a user asks about "these numbers", agents can look back to see what numbers were discussed
2. **team_members**:
    - Acts as a team roster

- Contains list of available specialized agents (e.g., Search, SECAnalyst)
- Helps supervisor know available resources for task assignment
- Example: `["Search", "SECAnalyst"]`

3. **next**:
    - Works like a "passing the baton" system
    - Indicates which agent should act next
    - Can signal completion with "FINISH"
    - Example: If market data is needed, `next = "Search"`
4. **information_needed**:
    - Functions as a shopping list of missing information
    - Guides agents on what to look for
    - Helps drive conversation towards completion
    - Example: `["current stock price", "last quarter revenue"]`
5. **reasoning**:
    - Serves as decision documentation
    - Explains why specific agents were chosen
    - Aids in debugging and system improvement
    - Example: "Choosing SECAnalyst because we need historical financial data"

State Management Flow

1. **Initial State Creation**

```
def create_initial_state(query: str) -> ResearchTeamState:
    """Create initial state from user query."""
    return {
        "messages": [HumanMessage(content=query)],
        "team_members": ["Search", "SECAnalyst"],
        "next": "",
        "information_needed": [],
        "reasoning": ""
    }
```

2. **State Updates**

```
def update_state(state: ResearchTeamState, agent_response: dict) ->
ResearchTeamState:
    """Update state with agent response."""
    new_state = state.copy()
    new_state["messages"].extend(agent_response["messages"])
    return new_state
```

The state management system ensures:

- Consistent tracking of conversation progress
- Proper coordination between agents
- Clear documentation of decision-making
- Efficient information gathering

2. Helper Functions Explained

A. agent_node Helper

This helper functions as a universal translator and task manager for agents. Think of it as a standardized communication interface that ensures all agents can work together effectively.

```
def agent_node(state, agent, name):
    """Helper function to create agent nodes."""
    try:
        # Add information needed to the state if available
        if "information_needed" in state:
            message_content = f"""Information needed:
            {', '.join(state['information_needed'])}

            Query: {state['messages'][-1].content}"""
            state['messages'][-1] = HumanMessage(content=message_content)

        # Process through agent
        result = agent.invoke(state)

        # Format response
        return {
            "messages": [
                HumanMessage(
                    content=result["output"],
                    name=name
                )
            ]
        }
    except Exception as e:
        logger.error(f"Error in agent node {name}: {e}")
        raise
```

The agent_node helper:

1. **Prepares Information**:
    - Takes the current state as input
    - Adds any required information to the query
    - Ensures the agent has full context for decision-making
2. **Manages Communication**:
    - Standardizes inter-agent communication
    - Maintains consistent message formatting
    - Keeps conversation history organized and accessible

B. create_agent Helper

This helper serves as a specialized agent factory that sets up agents with their specific capabilities and communication patterns.

```
def create_agent(
    llm: ChatOpenAI,
    tools: list,
    system_prompt: str,
) -> AgentExecutor:
    """Create a function-calling agent and add it to the graph."""
    try:
        # Enhance system prompt with team context
        system_prompt += (
            "\nWork autonomously according to your specialty, using the
tools available to you."
            " Do not ask for clarification."
            " Your other team members will collaborate with you with their
own specialties."
            " You are chosen for a reason! You are one of the following team
members: {team_members}."
        )

        # Create prompt template
        prompt = ChatPromptTemplate.from_messages([
            ("system", system_prompt),
            MessagesPlaceholder(variable_name="messages"),
            MessagesPlaceholder(variable_name="agent_scratchpad"),
        ])

        # Create and return agent
        agent = create_openai_functions_agent(llm, tools, prompt)
        executor = AgentExecutor(agent=agent, tools=tools)
        return executor

    except Exception as e:
        logger.error(f"Error creating agent: {e}")
        raise
```

The create_agent helper:

1. **Sets Up Agent Capabilities**:
    - Assigns specific tools to the agent
    - Defines the agent's area of expertise
    - Establishes behavioral guidelines
2. **Configures Communication Style**:
    - Sets up tool interaction patterns
    - Defines communication protocols
    - Establishes standardized message formats

C. create_team_supervisor Helper

This crucial helper creates the supervisor agent that coordinates the entire team's activities.

```
def create_team_supervisor(
    llm: ChatOpenAI,
    system_prompt: str,
    members: List[str]
) -> Callable:
    """Create an LLM-based supervisor with enhanced reasoning."""

    # Define possible routing options
    options = ["FINISH"] + members

    # Define routing function schema
    function_def = {
        "name": "route",
        "description": "Select the next role based on query analysis.",
        "parameters": {
            "title": "routeSchema",
            "type": "object",
            "properties": {
                "next": {
                    "title": "Next",
                    "anyOf": [{"enum": options}],
                },
                "reasoning": {
                    "title": "Reasoning",
                    "type": "string",
                    "description": "Explanation for why this agent should
act next"
                },
                "information_needed": {
                    "title": "Information Needed",
                    "type": "array",
                    "items": {"type": "string"},
                    "description": "List of specific information needed from
this agent"
                }
            },
            "required": ["next", "reasoning", "information_needed"],
        },
    }

    # Create prompt template
    prompt = ChatPromptTemplate.from_messages([
        ("system", system_prompt),
        MessagesPlaceholder(variable_name="messages"),
        (
            "system",
            """Given the conversation above, who should act next? Consider:
            1. What information do we have?
            2. What's still missing?
            3. Which agent can best provide the missing information?

            Select from: {options}"""
        ),
    ]).partial(options=str(options), team_members=", ".join(members))

    return (
        prompt
        | llm.bind_functions(functions=[function_def],
function_call="route")
        | JsonOutputFunctionsParser()
    )
```

Why These Helpers Matter

These helper functions form the backbone of our agent system by enabling:

1. Standardized agent creation and configuration
2. Consistent communication patterns across the system

3. Proper state management and tracking
4. Efficient team coordination
5. Easy system expansion and modification

The combination of these helpers ensures:

- Clean separation of concerns
- Maintainable and extensible code
- Reliable agent interactions
- Structured team coordination
- Consistent system behavior

3: Research Graph System

Overview

The research graph functions as a sophisticated traffic control system for our agent-based research process. It manages the flow of information and coordinates the actions of different agents to ensure efficient and organized research operations.

1. Core Functions

A. Flow Management

The research graph:

- Directs queries to appropriate agents based on expertise
- Handles responses and information processing
- Coordinates communication between different agents
- Ensures smooth transitions between different research phases

Example Flow:

```
User Query → Supervisor → Search Agent → Supervisor → SEC Agent → Supervisor
→ Response
```

B. Implementation

```
def create_research_graph(rag_chain) -> StateGraph:
    """Create the research team graph."""
    # Initialize LLM
    llm = ChatOpenAI(model="gpt-4-turbo-preview")

    # Create specialized agents
    search_agent = create_search_agent(llm)
    sec_agent = create_sec_agent(llm)

    # Create agent nodes with partial application
    search_node = functools.partial(agent_node, agent=search_agent,
name="Search")
    sec_node = functools.partial(agent_node, agent=sec_agent,
name="SECAnalyst")

    # Create team supervisor
    supervisor = create_supervisor_agent(llm)

    # Initialize graph with state type
    graph = StateGraph(ResearchTeamState)

    # Add agent nodes to graph
    graph.add_node("Search", search_node)
    graph.add_node("SECAnalyst", sec_node)
    graph.add_node("supervisor", supervisor)

    # Define primary transitions
    graph.add_edge("Search", "supervisor")
    graph.add_edge("SECAnalyst", "supervisor")

    # Add conditional routing from supervisor
    graph.add_conditional_edges(
        "supervisor",
        lambda x: x["next"],
        {
            "Search": "Search",
            "SECAnalyst": "SECAnalyst",
            "FINISH": END
        },
    )

    # Configure entry point
    graph.set_entry_point("supervisor")

    return graph.compile()
```

2. Query Processing and Communication

A. Query Processing Flow

The `process_financial_query` function serves as the entry point for all financial analysis requests:

```python
def process_financial_query(chain, query: str):
    """Process a financial query through the research graph."""
    try:
        # Initialize and invoke the research chain
        result = chain.invoke({
            "messages": [HumanMessage(content=query)],
            "team_members": ["Search", "SECAnalyst"],
            "information_needed": [],
            "reasoning": ""
        })
        return result
    except Exception as e:
        return f"Error processing query: {str(e)}"
```

This function:

1. Takes a query and initializes the research process
2. Creates initial state with default team configuration
3. Triggers the supervisor's analysis and decision-making
4. Manages the flow through various agents
5. Returns consolidated findings and analysis

B. Query Flow Lifecycle

1. **Query Reception**:
    - System receives financial query
    - Initializes research state
    - Prepares team configuration
2. **Analysis Phase**:
    - Supervisor analyzes query intent
    - Identifies required information types
    - Determines optimal agent sequence
    - Plans information gathering strategy
3. **Execution Phase**:

- Routes tasks to appropriate agents
- Coordinates information gathering
- Manages agent transitions
- Consolidates findings

Example Usage

```
# Example query processing
query = "What are Apple's recent revenue trends and market performance?"
result = process_financial_query(research_chain, query)

# Example response format
{
    "messages": [
        {
            "role": "agent",
            "name": "Search",
            "content": "Market Data: [Recent market findings...]"
        },
        {
            "role": "agent",
            "name": "SECAnalyst",
            "content": "Financial Analysis: [SEC filing insights...]"
        }
    ]
}
```

C. Agent Communication Protocol

The system uses a standardized communication protocol to ensure consistent and effective interaction between agents:

1. **Message Structure**:

   ```
   {
       "messages": List[BaseMessage],    # Complete conversation history
       "next": str,                      # Next agent designation
       "information_needed": List[str],  # Specific data requirements
       "reasoning": str                  # Decision justification
   }
   ```

2. **Agent Response Formats**:
   - **SEC Analysis Agent**:

        - Financial Data Findings
        - Context Requirements
        - Analytical Insights
        - Historical Trends
    - **Search Agent**:
        - Current Market Data
        - Analyst Perspectives
        - SEC Data Correlation
        - Market Trends
3. **Communication Flow Control**:
    - Structured message passing
    - Clear handoff protocols
    - Explicit state transitions
    - Documented decision paths

3. Key Components

A. Node Structure

1. **Agent Nodes**:
    - Represent specialized research agents (Search, SECAnalyst)
    - Contain specific tools and capabilities
    - Handle particular aspects of research
2. **Supervisor Node**:
    - Manages workflow and agent selection
    - Makes routing decisions
    - Ensures research completeness

B. Edge System

1. **Direct Edges**:
    - Connect agents to supervisor
    - Enable immediate feedback loops
    - Maintain clear communication paths
2. **Conditional Edges**:
    - Enable dynamic routing based on state
    - Allow for flexible workflow adaptation
    - Support complex decision trees

The research graph serves as the foundation for coordinated, efficient research operations while maintaining flexibility and reliability.

2.4 System Configuration and Startup

Environment Configuration

A. Required API Keys

The system requires several API keys to function properly:

```
# .env file structure
OPENAI_API_KEY=your_openai_api_key
TAVILY_API_KEY=your_tavily_api_key
AGENTVERSE_API_KEY=your_agentverse_api_key
```

B. Environment Setup

```python
from dotenv import load_dotenv
import os

# Load environment variables
load_dotenv()

# Verify environment configuration
def verify_env_config():
    """Verify all required API keys are present."""
    required_keys = [
        "OPENAI_API_KEY",
        "TAVILY_API_KEY",
        "AGENTVERSE_API_KEY"
    ]

    missing_keys = [
        key for key in required_keys
        if not os.getenv(key)
    ]

    if missing_keys:
        raise EnvironmentError(
            f"Missing required API keys: {', '.join(missing_keys)}"
        )
```

System Initialization

A. RAG and Research Chain Setup

```python
from src.rag.chain import create_rag_chain
from src.graph.state import create_research_graph

def init_financial_system():
    """Initialize the RAG and research chain."""
    # Create RAG chain with specific document
    rag_chain = create_rag_chain("data/raw/apple_10k.pdf")

    # Initialize research graph with RAG chain
    chain = create_research_graph(rag_chain)

    return chain
```

B. Main Entry Point

```python
if __name__ == "__main__":
    # Load environment variables
    load_dotenv()

    # Verify environment configuration
    verify_env_config()

    # Import and run agent
    from src.agentverse.register import run_agent
    run_agent()
```

Chapter 3 - End to End Application with Agentverse Integration

3.1 Architecture Overview

What is Agentverse?

Agentverse is a virtual hub designed for creating, managing, and deploying agents. It provides an accessible platform for both developers and non-technical users to interact with Fetch.ai's autonomous agent technology, offering features like agent discovery, remote communication, and storage capabilities.

🌐 Fetch Network Integration

The Fetch Network, through its open nature, provides the foundational infrastructure with:

1. **Almanac Contract**
    - Core smart contract for agent registration
    - Acts as a comprehensive repository
    - Serves as single point of truth
2. **Discovery Mechanism**
    - Enables network-wide agent discovery
    - Maintains registry of agent capabilities

Why register your agent with Agentverse?

Registration with Agentverse offers several key benefits:

1. **Agent Discovery**
    - Makes your agent discoverable by other agents in the network
    - Publishes your agent's capabilities to potential collaborators
2. **Interaction Capabilities**
    - Enables peer-to-peer agent communication
    - Allows leveraging functionalities of other registered agents
3. **Secure Collaboration**
    - Establishes secure connections through the Almanac contract
    - Facilitates seamless agent-to-agent interactions
    - Enables collaborative problem-solving

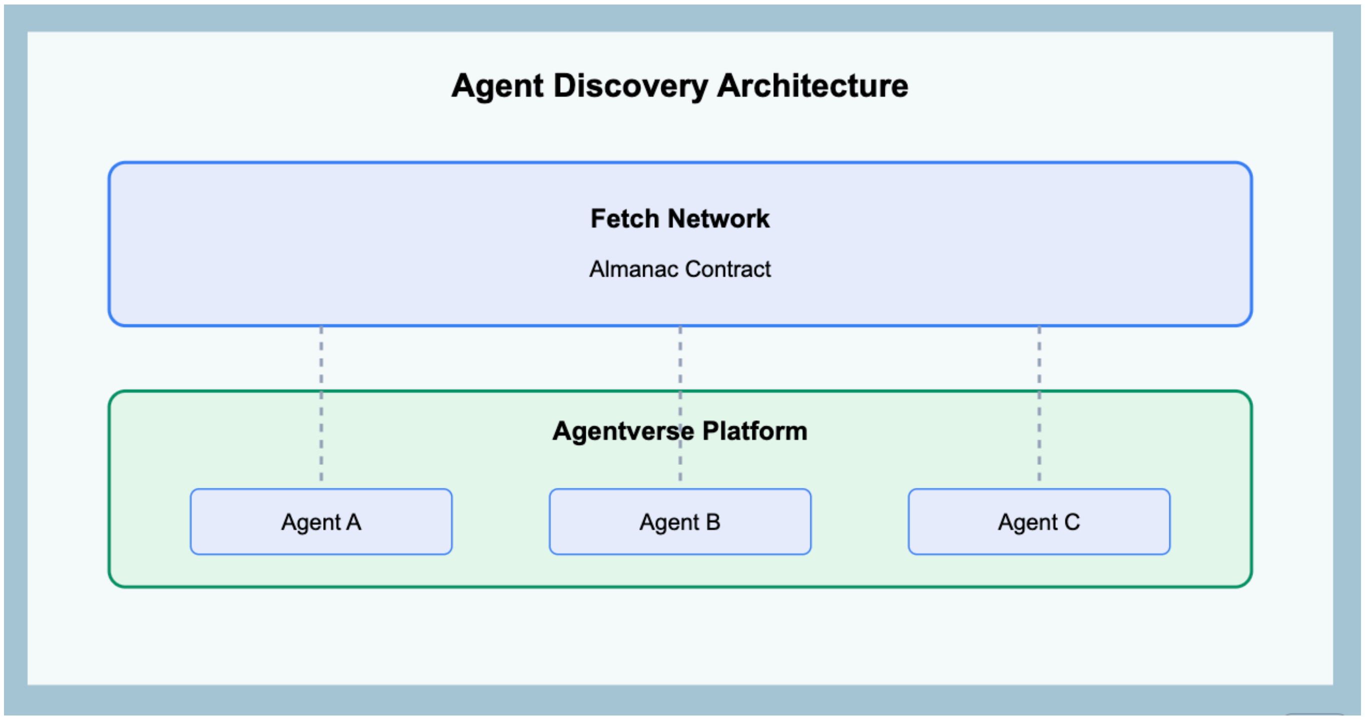


Figure 8: Agent registration with Fetch Network

System Overview

- This diagram depicts an end to end application architecture, where the Client Application interacts with a Prime Agent to discover and utilize various specialized Agents registered within the Agentverse.
- The Prime Agent orchestrates the communication between the Client and the Agents to fulfill the client's requests.
- This architecture allows for a modular and extensible system, where new agents can be easily integrated, and the Prime Agent can orchestrate the interactions between agents to fulfill complex client requests.

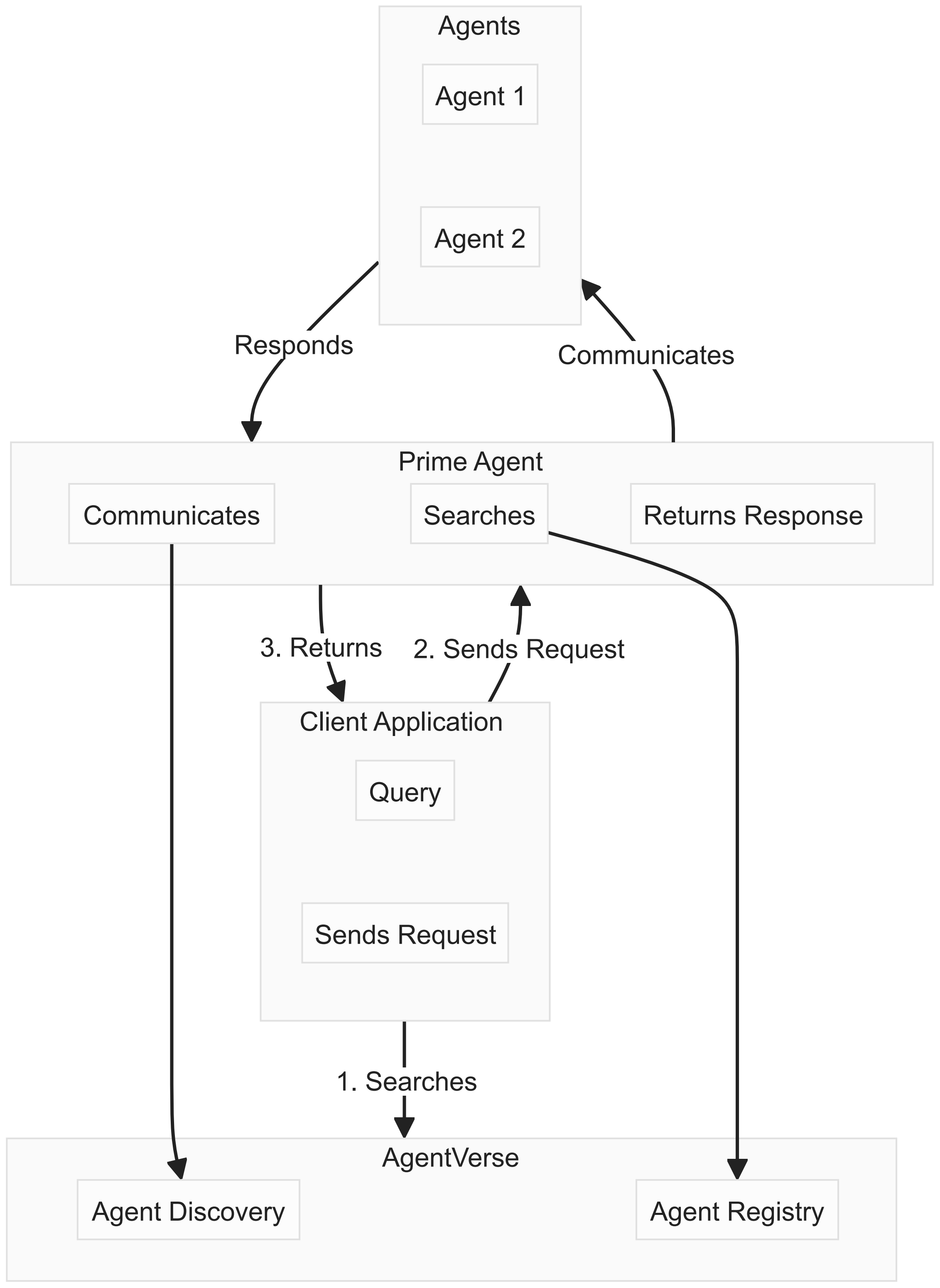


Figure 9: Application Architecture

3.2: Core components and Implementation

In this chapter, we'll transform the Financial Analysis Agent we built in Chapter 2 into a complete application by:

1. Registering it with Agentverse for discovery
2. Creating a Primary Agent to handle routing and communication
3. Building a user interface for interaction

Registering financial analysis agent with Agentverse

The Financial Analysis Agent we built in Chapter 2 needs to be registered with Agentverse through the registration process. This enables other agents to discover and communicate with it.For a comprehensive implementation of the end to end application, refer to the full source code repository at Financial-Analysis-AI-Agent.

Agent Registration Implementation (src/agentverse/register.py)

This code needs to be added in the financial analysis agent folder.

Agent Registration and Setup

```
def init_agent():
    """Initialize and register the agent with agentverse"""
    global financial_identity, research_chain
    try:
        # Initialize the research chain
        research_chain = init_financial_system()

        # Initialize identity and register with Agentverse
        financial_identity = Identity.from_seed(
            os.getenv("FINANCIAL_AGENT_KEY"),
            0
        )

        # Register with detailed capabilities description
        register_with_agentverse(
            identity=financial_identity,
            url="http://localhost:5008/webhook",
            agentverse_token=os.getenv("AGENTVERSE_API_KEY"),
            agent_title="Financial Analysis Agent",
            readme = """
                <description>A comprehensive financial analysis agent that 
combines
Inc.</description>
                <use_cases>
                    <use_case>Get detailed revenue analysis from SEC
filings</use_case>
                    <use_case>Analyze risk factors from latest 10-
K</use_case>
                    <use_case>Track financial metrics and trends</use_case>
                </use_cases>
                <payload_requirements>
                    <payload>
                        <requirement>
                            <parameter>query</parameter>
                            <description>What would you like to know about
Apple's financials?</description>
                        </requirement>
                    </payload>
                </payload_requirements>
            """
        )
```

Query Processing and Response

```
@app.route('/webhook', methods=['POST'])
def webhook():
    try:
        # Parse incoming message
        data = request.get_data().decode('utf-8')
        message = parse_message_from_agent(data)
        query = message.payload.get("request", "")
        agent_address = message.sender

        # Validate query
        if not query:
            return jsonify({"status": "error", "message": "No query
provided"}), 400

        # Process query using research chain
        result = research_chain.invoke({
            "messages": [HumanMessage(content=query)],
            "team_members": ["Search", "SECAnalyst"]
        })

        # Format response for client
        formatted_result = {
            "analysis": [
                {
                    "role": msg.type if hasattr(msg, 'type') else "message",
                    "content": msg.content,
                    "name": msg.name if hasattr(msg, 'name') else None
                }
                for msg in result.get('messages', [])
            ]
        }

        # Send response back through Agentverse
        send_message_to_agent(
            financial_identity,
            agent_address,
            {'analysis_result': formatted_result}
        )
        return jsonify({"status": "analysis_sent"})

    except Exception as e:
        logger.error(f"Error in webhook: {e}")
        return jsonify({"status": "error", "message": str(e)}), 500
```

Project Structure Overview

```
Financial-Analysis-AI-Agent/
├── backend/               # Primary Agent implementation
│   └── app.py            # Primary Agent Flask application
├── frontend/             # React Vite frontend
│   ├── src/
│   │   ├── components/
│   │   │   └── ui/     # shadcn/ui components
│   │   │       └── card.jsx
│   │   ├── OptimusPrime.jsx
│   │   ├── App.jsx
│   │   └── main.jsx
│   ├── public/
│   ├── package.json
│   ├── vite.config.js
│   └── .env
└── financial-analysis-agent/  # From Chapter 2
```

Prerequisites

Before setting up the application, ensure you have:

```
# Required software
Python 3.8+
Node.js 16+
Git

# Python packages for backend
flask
flask-cors
python-dotenv
fetchai-sdk

# Node packages will be installed during frontend setup
```

Environment Setup

Create .env files for both backend and frontend:

```
# backend/.env
PRIMARY_AGENT_KEY=your_primary_agent_key
AGENTVERSE_API_KEY=your_agentverse_api_key
FINANCIAL_AGENT_PORT=5008
PRIMARY_AGENT_PORT=5001

# frontend/.env
VITE_API_URL=http://localhost:5001
```

Starting Order

The components must be started in this specific order:

1. Start Financial Analysis Agent (from Chapter 2):

```
# From project root
python main.py
```

2. Start Primary Agent:

```
cd backend
```

```
python app.py
```

3. Start Frontend:

```
cd frontend
npm run dev
```

Primary Agent Implementation

The Primary Agent acts as an intermediary between the frontend and the Financial Analysis Agent. It's implemented in `backend/app.py`.

Required Imports

```
from flask import Flask, request, jsonify
from flask_cors import CORS
from fetchai.crypto import Identity
from fetchai.registration import register_with_agentverse
from fetchai.communication import parse_message_from_agent,
send_message_to_agent
from fetchai import fetch
import logging
import os
from dotenv import load_dotenv
```

Basic Setup

```
# Configure logging
logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)

# Initialize Flask app with CORS
app = Flask(__name__)
CORS(app, resources={r"/api/*": {'origins': 'http://localhost:5174'}})

# Initialize Primary Agent class
class PrimaryAgent:
    def __init__(self):
        self.identity = None
        self.latest_response = None

    def initialize(self):
        try:
            self.identity = 
Identity.from_seed(os.getenv("PRIMARY_AGENT_KEY"), 0)

            register_with_agentverse(
                identity=self.identity,
                url="http://localhost:5001/webhook",
                agentverse_token=os.getenv("AGENTVERSE_API_KEY"),
                agent_title="Financial Query Router",
                readme="<description>Routes queries to Financial Analysis 
Agent</description>"
            )
            logger.info("Primary agent initialized successfully!")

        except Exception as e:
            logger.error(f"Initialization error: {e}")
            raise

    def find_financial_agent(self):
        """Find our registered financial analysis agent"""
        try:
            available_ais = fetch.ai("Financial Analysis Agent")
            agents = available_ais.get('ais', [])

            if agents:
                logger.info(f"Found financial agent at address: {agents[0]
['address']}")
                return agents[0]
            return None

        except Exception as e:
            logger.error(f"Error finding financial agent: {e}")
            return None

# Create global instance
primary_agent = PrimaryAgent()
```

API Endpoints

Search Agents Endpoint

```
@app.route('/api/search-agents', methods=['GET'])
def search_agents():
    """Search for available agents based on the financial query"""
    try:
        query = request.args.get('query', '')
        if not query:
            return jsonify({"error": "Query parameter 'query' is 
required."}), 400

        logger.info(f"Searching for agents with query: {query}")
        available_ais = fetch.ai(query)
        agents = available_ais.get('ais', [])

        extracted_data = [
            {
                'name': agent.get('name'),
                'address': agent.get('address')
            }
            for agent in agents
        ]

        logger.info(f"Found {len(extracted_data)} agents matching the 
query")
        return jsonify(extracted_data), 200

    except Exception as e:
        logger.error(f"Error finding agents: {e}")
        return jsonify({"error": str(e)}), 500
```

Send Request Endpoint

```
@app.route('/api/send-request', methods=['POST'])
def send_request():
    try:
        data = request.json
        payload = data.get('payload', {})
        user_input = payload.get('request')  # Get request from nested 
payload
        agent_address = data.get('agentAddress')

        if not user_input:
            return jsonify({"error": "No input provided"}), 400

        # Find financial analysis agent if address not provided
        if not agent_address:
            agent = primary_agent.find_financial_agent()
            if not agent:
                return jsonify({"error": "Financial analysis agent not 
available"}), 404
            agent_address = agent['address']
        # Send request to financial agent
        send_message_to_agent(
            primary_agent.identity,
            agent_address,
            {
                "request": user_input
            }
        )

        return jsonify({
            "status": "request_sent",
            "agent_address": agent_address,
            "payload": payload
        })

    except Exception as e:
        logger.error(f"Error processing request: {e}")
        return jsonify({"error": str(e)}), 500
```

Get Response Endpoint

```
@app.route('/api/get-response', methods=['GET'])
def get_response():
    try:
        if primary_agent.latest_response:
            response = primary_agent.latest_response
            primary_agent.latest_response = None
            return jsonify(response)
        return jsonify({"status": "waiting"})
    except Exception as e:
        logger.error(f"Error getting response: {e}")
        return jsonify({"error": str(e)}), 500
```

Webhook Endpoint

```
@app.route('/webhook', methods=['POST'])
def webhook():
    try:
        data = request.get_data().decode("utf-8")
        message = parse_message_from_agent(data)
        primary_agent.latest_response = message.payload
        return jsonify({"status": "success"})
    except Exception as e:
        logger.error(f"Error in webhook: {e}")
        return jsonify({"error": str(e)}), 500
```

Running the Agent

```
if __name__ == "__main__":
    load_dotenv()
    primary_agent.initialize()
    app.run(host="0.0.0.0", port=5001)
```

Communication Flow

1. User submits query through frontend
2. Query reaches Primary Agent
3. Primary Agent forwards to Financial Analysis Agent
4. Financial Analysis Agent processes query (using implementation from Chapter 2)
5. Response returns through Primary Agent
6. Frontend displays results

Frontend Implementation

Setting up the Vite Project

Create a new Vite project:

```
# Create new project
npm create vite@latest frontend -- --template react
cd frontend

# Install required dependencies
npm install

# Install additional dependencies
npm install lucide-react  # For icons
npm install @radix-ui/react-slot @radix-ui/react-icons  # For shadcn/ui

# Set up Tailwind CSS
npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init -p
```

Configure Tailwind CSS (tailwind.config.js)

```
/** @type {import('tailwindcss').Config} */
export default {
  content: [
    "./index.html",
    "./src/**/*.{js,ts,jsx,tsx}",
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

Main Application Component (src/App.jsx)

```
import OptimusPrime from './components/OptimusPrime'

function App() {
  return (
    <div>
      <OptimusPrime />
    </div>
  )
}

export default App
```

Main Chat Component (src/components/OptimusPrime.jsx)

The OptimusPrime component implements the chat interface:

Message Handling and UI State

```
const OptimusPrime = () => {
    const [messages, setMessages] = useState([]);
    const [inputText, setInputText] = useState('');
    const [isProcessing, setIsProcessing] = useState(false);

    // Handles submitting new messages
    const handleSendMessage = async () => {
        if (!inputText.trim() || isProcessing) return;
        // Add user message to UI
        const userMessage = {
            type: 'user',
            content: inputText,
            timestamp: new Date().toLocaleTimeString()
        };
        setMessages(prev => [...prev, userMessage]);
        setInputText('');
        setIsProcessing(true);

        try {
            // Send request to primary agent

            await fetch('/api/send-request', {
                method: 'POST',
                headers: { 'Content-Type': 'application/json' },
                body: JSON.stringify({ input: inputText }),
            });


            // Start polling for response
            startPollingForResponse();
        } catch (error) {
            handleError(error);
        }
    };
```

Response Polling System

```
    // Polls for agent response
    const startPollingForResponse = () => {
        const pollInterval = setInterval(async () => {
            try {
                const responseData = await fetch('/api/get-response');
                const data = await responseData.json();

                if (data.status !== 'waiting' && data.analysis_result) {
                    clearInterval(pollInterval);
                    setIsProcessing(false);

                    // Process agent responses
                    data.analysis_result.analysis.forEach(response => {
                        setMessages(prev => [...prev, {
                            type: 'agent',
                            agentName: response.name || 'Agent',
                            content: response.content,
                            timestamp: new Date().toLocaleTimeString()
                        }]);
                    });
                }
            } catch (error) {
                clearInterval(pollInterval);
                setIsProcessing(false);
                handleError(error);
            }
        }, 1000);
    };
```

3.3 Testing and System Interaction

Prerequisites Check

Before testing, ensure all components are running:

```
# 1. Financial Analysis Agent
# In project root
python main.py

# Terminal output should show:
INFO:__main__:Financial Analysis Agent registered successfully!
 * Serving Flask app 'app'
 * Running on http://0.0.0.0:5008

# 2. Primary Agent
# In another terminal, in backend directory
python app.py

# Terminal output should show:
INFO:__main__:Primary agent initialized successfully!
 * Serving Flask app 'app'
 * Running on http://0.0.0.0:5001

# 3. Frontend
# In another terminal, in frontend directory
npm run dev

# Terminal output should show:
VITE v5.x.x ready in xxx ms
➜ Local:   http://localhost:5174/
```

Testing API Endpoints

You can test the Primary Agent's endpoints using curl:

```
# 1. Test agent search
curl -X GET "http://localhost:5001/api/search-agents?
query=financial%20analysis"

# Expected response:
{
    "name": "Financial Analysis Agent",
    "address": "agent1..."
}

# 2. Test sending request
curl -X POST "http://localhost:5001/api/send-request" \
-H "Content-Type: application/json" \
-d '{
    "payload": {
        "request": "What are Apple'\''s recent revenue trends?"
    }
}'

# Expected response:
{
    "status": "request_sent",
    "agent_address": "agent1...",
    "payload": {
        "request": "What are Apple's recent revenue trends?"
    }
}

# 3. Test getting response
curl -X GET "http://localhost:5001/api/get-response"

# Expected response while processing:
{
    "status": "waiting"
}

# Expected response when ready:
{
    "analysis_result": {
        "analysis": [
            {
                "type": "agent",
                "content": "Based on the analysis...",
                "name": "Financial Analyst"
            }
        ]
    }
}
```

End-to-End Testing

1. Startup Sequence Test:

```
# Check if agents are registered correctly
curl -X GET "http://localhost:5001/api/search-agents?query=financial"
# Should return the Financial Analysis Agent in the results
```

2. UI Testing Checklist:

- [ ] Open frontend at http://localhost:5174
- [ ] Verify chat interface loads
- [ ] Check message input field is enabled
- [ ] Verify send button is visible

3. Communication Flow Test:

```
# 1. Send a test query through the UI
Enter: "What were Apple's revenues in the last quarter?"

# 2. Check Primary Agent logs
# Should see something like:
INFO:__main__:Found financial agent at address: agent1...
INFO:__main__:Sending request to financial agent...

# 3. Check Financial Analysis Agent logs
# Should see processing messages and analysis completion
```

Common Issues and Solutions

1. CORS Issues:

```
Access to fetch at 'http://localhost:5001/api/...' from origin
'http://localhost:5174' has been blocked by CORS policy
```

Solution: Check CORS configuration in app.py:

```
CORS(app, resources={r"/api/*": {'origins': 'http://localhost:5174'}})
```

2. Agent Not Found:

```
{"error": "Financial analysis agent not available"}
```

Solution: Ensure Financial Analysis Agent is running and properly registered:

```
# Check agent registration
curl -X GET "http://localhost:5001/api/search-agents?query=financial"
```

3. Connection Refused:

```
Failed to fetch: NetworkError when attempting to fetch resource
```

Solution: Verify all services are running on correct ports:

- Financial Analysis Agent: 5008
- Primary Agent: 5001
- Frontend: 5174

System Verification Checklist

1. Environment Setup:

- [ ] All environment variables set correctly
- [ ] Required Python packages installed
- [ ] Node.js dependencies installed

2. Component Status:

- [ ] Financial Analysis Agent running (port 5008)
- [ ] Primary Agent running (port 5001)
- [ ] Frontend running (port 5174)

3. Integration Points:

- [ ] Frontend can connect to Primary Agent

- [ ] Primary Agent can find Financial Analysis Agent
- [ ] Messages flow through the complete system

4. Functionality:

- [ ] Chat interface responsive
- [ ] Messages sent successfully
- [ ] Responses received and displayed
- [ ] Error handling working
- [ ] Loading states visible

Sample Usage and Response Formats

Query Processing Flow

The system processes financial analysis queries through a structured flow, with agents collaborating to provide comprehensive analysis.

Example Query Types:

1. R&D Investment Analysis
2. Supply Chain Risk Assessment
3. Capital Allocation Analysis

Response Formats

1. Information Gathering Phase

When agents need additional information, they format their response as:

```
Information needed:
        [List of required information]

Query: [Original query text]
```

Example:

```
Information needed:
        Historical R&D investment data for Apple.
        Comparison of Apple's R&D investment trends with key competitors.

Query: How has Apple's R&D investment strategy evolved...
```

2. SEC Analysis Response Format

SEC Agent provides structured analysis in three sections:

```
1. Data from SEC Filings:
   - [Specific data points]
   - [Financial metrics]
   - [Trends]

2. Additional Context Needed:
   - [Required supplementary data]
   - [Missing information]

3. Analysis:
   - [Detailed analysis]
   - [Insights]
   - [Implications]
```

3. Search Agent Response Format

Search Agent provides market research in three sections:

```
1. Market Data:
   - [Current data]
   - [Statistics]
   - [Metrics]

2. Analyst Views:
   - [Expert opinions]
   - [Market sentiment]
   - [Industry analysis]

3. Relevance to SEC Data:
   - [Connections to SEC findings]
   - [Comparative analysis]
   - [Market context]
```

Example Interaction Flow

Query:

"How has Apple's R&D investment strategy evolved compared to competitors?"

Response Sequence:

1. Initial Information Request:

```
Information needed:
        Historical R&D investment data for Apple.
        Comparison of Apple's R&D investment trends with key
competitors.
```

2. SEC Agent Analysis:

```
Data from SEC Filings:
- Apple's R&D Investment Trends:
  - 2024: $31,370 million
  - 2023: $29,915 million
  [...]
```

3. Search Agent Analysis:

```
Market Data:
- Samsung's R&D Investment:
  - 2024: $24 billion
  - 2023: 28.34 trillion KRW
[...]
```

Sample queries and output

Let's test it with some complex queries that will show how the agents work together. Try these:

1. A query that needs both historical and market data:

```python
"How has Apple's R&D investment strategy evolved compared to
competitors, and what does this suggest about their future innovation
priorities?"
```

2. A query that needs deep SEC analysis with market context:

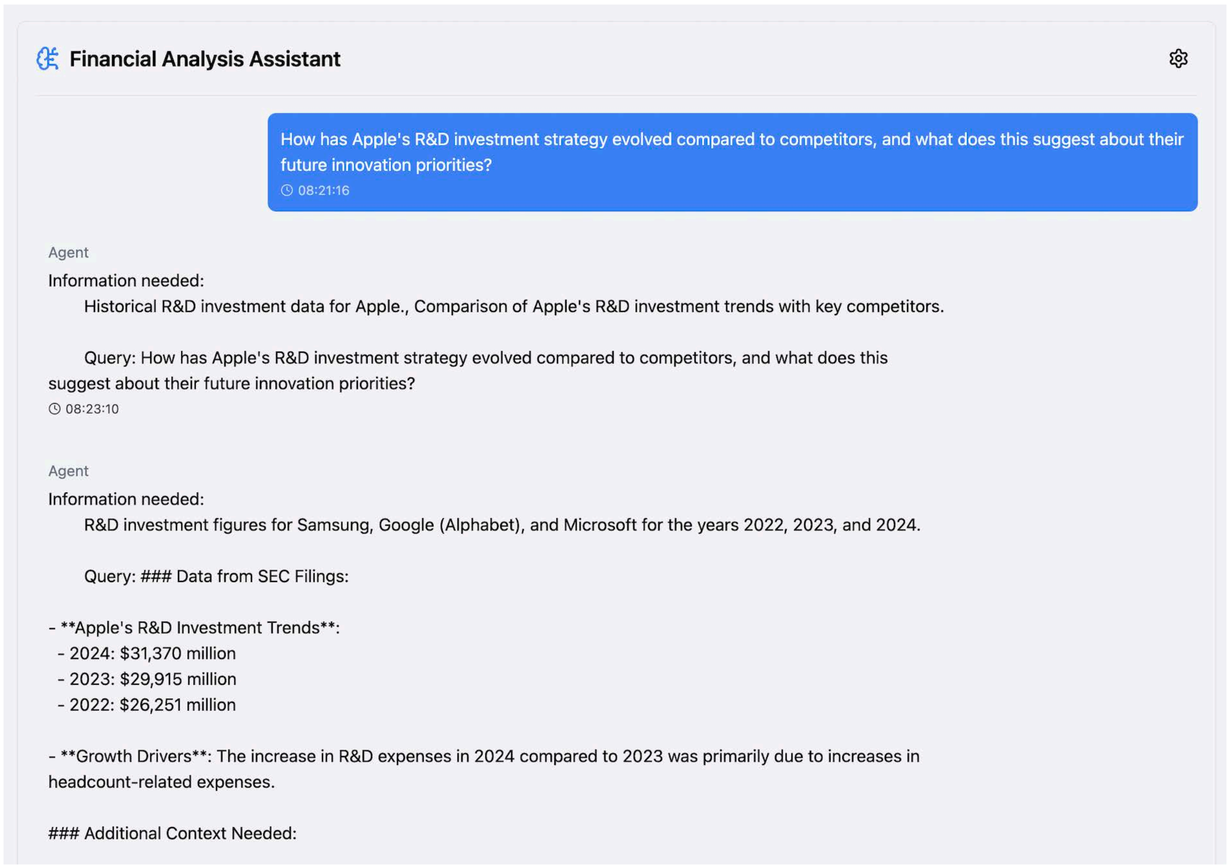
```python
"Analyze Apple's supply chain risks from their 10-K and how these
risks are currently manifesting in their business operations."
```

3. A forward-looking query requiring multiple perspectives:

```python
"Based on Apple's capital allocation patterns and current market
trends, what are their likely investment priorities for the next year?"
```

Let's try running one of these and watch:

1. How the supervisor breaks down the information needs
2. The order of agent involvement
3. How information is built up between agents
4. When the supervisor decides to FINISH


Financial Analysis Assistant

How has Apple's R&D investment strategy evolved compared to competitors, and what does this suggest about their future innovation priorities?
08:21:16

Agent
Information needed:
Historical R&D investment data for Apple., Comparison of Apple's R&D investment trends with key competitors.

Query: How has Apple's R&D investment strategy evolved compared to competitors, and what does this suggest about their future innovation priorities?
08:23:10

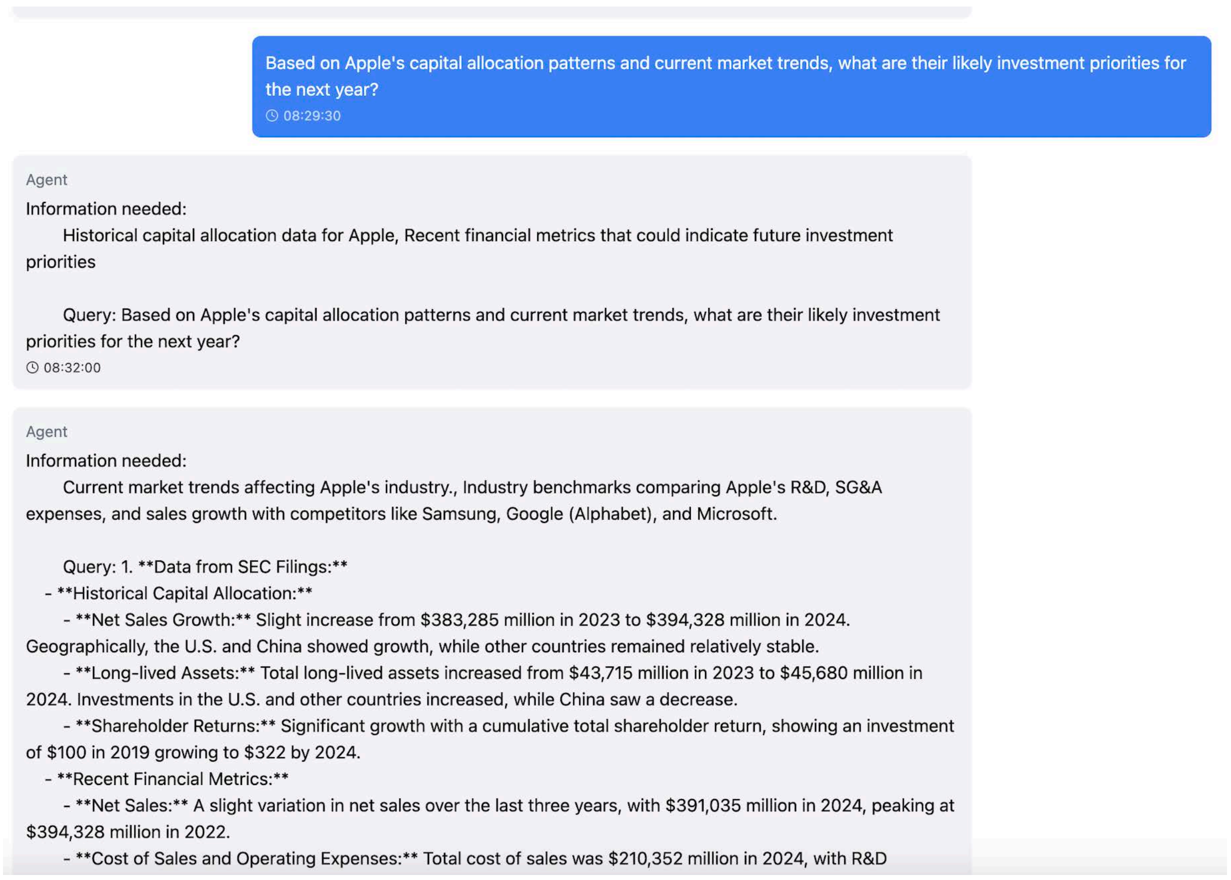
Agent
Information needed:
R&D investment figures for Samsung, Google (Alphabet), and Microsoft for the years 2022, 2023, and 2024.

Query: ### Data from SEC Filings:

- **Apple's R&D Investment Trends**:
 - 2024: $31,370 million
 - 2023: $29,915 million
 - 2022: $26,251 million

- **Growth Drivers**: The increase in R&D expenses in 2024 compared to 2023 was primarily due to increases in headcount-related expenses.

Additional Context Needed:

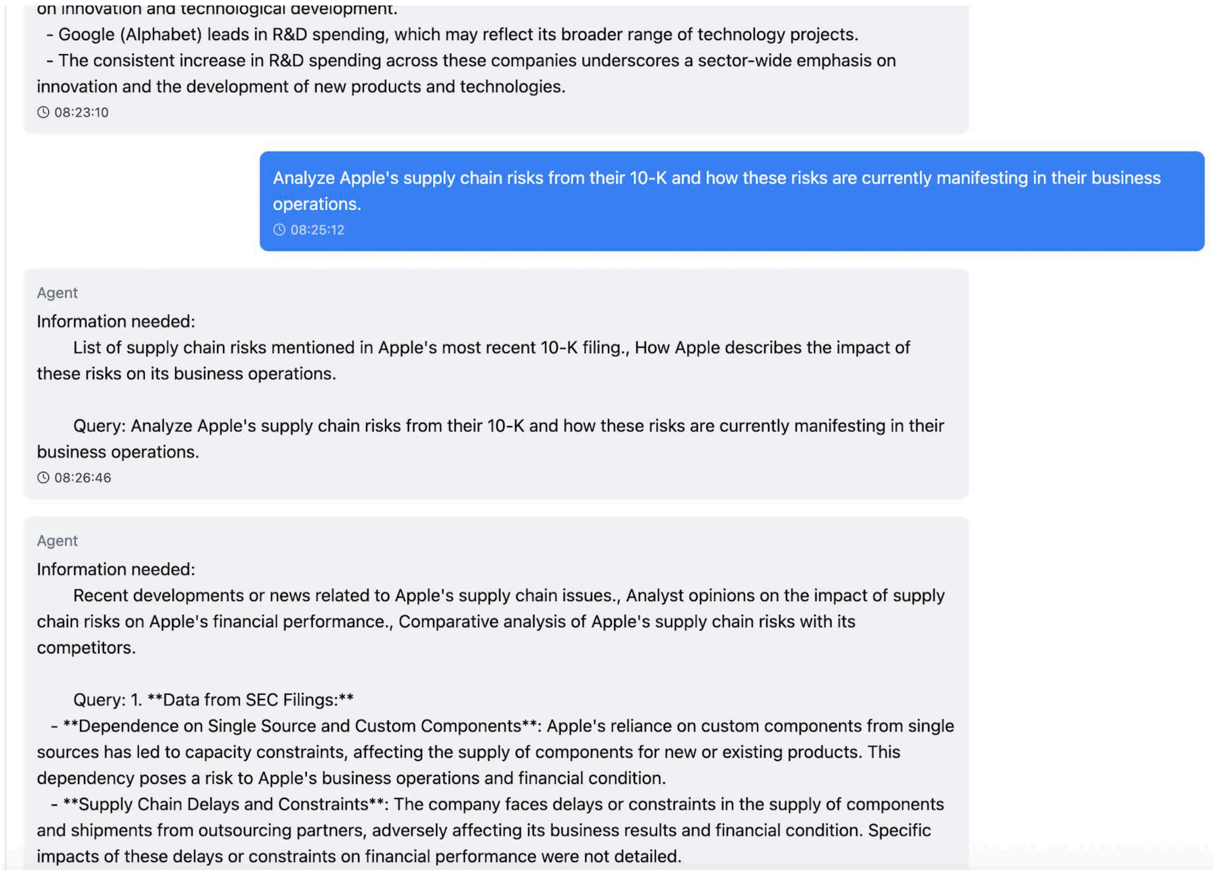

Based on Apple's capital allocation patterns and current market trends, what are their likely investment priorities for the next year?
08:29:30

Agent
Information needed:
Historical capital allocation data for Apple, Recent financial metrics that could indicate future investment priorities

Query: Based on Apple's capital allocation patterns and current market trends, what are their likely investment priorities for the next year?
08:32:00

Agent
Information needed:
Current market trends affecting Apple's industry., Industry benchmarks comparing Apple's R&D, SG&A expenses, and sales growth with competitors like Samsung, Google (Alphabet), and Microsoft.

Query: 1. **Data from SEC Filings:**
 - **Historical Capital Allocation:**
 - **Net Sales Growth:** Slight increase from $383,285 million in 2023 to $394,328 million in 2024. Geographically, the U.S. and China showed growth, while other countries remained relatively stable.
 - **Long-lived Assets:** Total long-lived assets increased from $43,715 million in 2023 to $45,680 million in 2024. Investments in the U.S. and other countries increased, while China saw a decrease.
 - **Shareholder Returns:** Significant growth with a cumulative total shareholder return, showing an investment of $100 in 2019 growing to $322 by 2024.
 - **Recent Financial Metrics:**
 - **Net Sales:** A slight variation in net sales over the last three years, with $391,035 million in 2024, peaking at $394,328 million in 2022.
 - **Cost of Sales and Operating Expenses:** Total cost of sales was $210,352 million in 2024, with R&D



on innovation and technological development.
- Google (Alphabet) leads in R&D spending, which may reflect its broader range of technology projects.
- The consistent increase in R&D spending across these companies underscores a sector-wide emphasis on innovation and the development of new products and technologies.

08:23:10

Analyze Apple's supply chain risks from their 10-K and how these risks are currently manifesting in their business operations.

08:25:12

Agent

Information needed:
List of supply chain risks mentioned in Apple's most recent 10-K filing., How Apple describes the impact of these risks on its business operations.

Query: Analyze Apple's supply chain risks from their 10-K and how these risks are currently manifesting in their business operations.

08:26:46

Agent

Information needed:
Recent developments or news related to Apple's supply chain issues., Analyst opinions on the impact of supply chain risks on Apple's financial performance., Comparative analysis of Apple's supply chain risks with its competitors.

Query: 1. **Data from SEC Filings:**
- **Dependence on Single Source and Custom Components**: Apple's reliance on custom components from single sources has led to capacity constraints, affecting the supply of components for new or existing products. This dependency poses a risk to Apple's business operations and financial condition.
- **Supply Chain Delays and Constraints**: The company faces delays or constraints in the supply of components and shipments from outsourcing partners, adversely affecting its business results and financial condition. Specific impacts of these delays or constraints on financial performance were not detailed.

Conclusion

This guide provides an easy introduction to AI agents, clarifying their fundamental characteristics and common implementation patterns. We started by examining the core concepts that distinguish AI agents from other types of AI applications. While there is no single universal standard definition of an 'AI agent', we identified a spectrum of autonomy and capabilities that characterize these systems. The discussion covered essential components of AI agents, various system types, common misconceptions, and the dynamics of multi-agent interactions.

We developed a very simple AI agent using langgraph. More importantly, this guide emphasizes the platform-agnostic nature of agent development and registration. Whether you're building agents using LangGraph, CrewAI, AutoGen, LangChain, Phidata, or any other framework, the Fetch.ai's Agentverse platform offers a unified solution for agent discovery and collaboration. By registering your agents with Agentverse through the Fetch Network's Almanac contract, you enable them to be discovered by and interact with other agents, regardless of their underlying framework.

This open and flexible approach to agent registration promotes:

- Cross-framework collaboration between agents
- Framework-independent agent discovery
- Seamless integration of diverse agent implementations
- Growth of a diverse agent ecosystem

References

1. **Google AI Agents White Paper**
   Google Research. AI Agents White Paper.
   Retrieved from https://archive.org/details/google-ai-agents-whitepaper
2. **Building Effective Agents**
   Anthropic. A Guide to Building Effective Agents.
   Retrieved from https://www.anthropic.com/research/building-effective-agents
3. **LangChain Resources**
   LangChain-AI
   https://langchain-ai.github.io/
   https://python.langchain.com/docs/concepts/
4. **Fetch.ai Documentation**
   Fetch.ai. Developer Documentation.
   https://fetch.ai/docs
5. **Additional Resources and Market Research**
   - https://scoop.market.us/agentic-ai-for-financial-services-market-news/
   - https://www.turbotic.com/blog/ai-agentic-workflows/
   - https://www.moveworks.com/us/en/resources/blog/what-is-agentic-workflows-in-ai
   - https://digitaldefynd.com/IQ/agentic-ai-statistics/
   - https://www.multimodal.dev/post/ai-agentic-workflows
   - https://www.statista.com/statistics/1552183/global-agentic-ai-market-value/
   - https://www.hypotenuse.ai/blog/what-is-an-ai-agentic-workflow
   - https://www2.deloitte.com/us/en/insights/industry/technology/technology-media-and-telecom-predictions/2025/autonomous-generative-ai-agents-still-under-development.html
   - https://www.miquido.com/ai-glossary/ai-agentic-workflows/

Appendix

Some useful things to know

5 Stages towards AGI

According to OpenAI's recently unveiled five-step roadmap for tracking progress towards Artificial General Intelligence (AGI), the stages are:

1. Conversational AI: This is the current level, where AI systems like ChatGPT excel at engaging in natural, human-like conversations across various topics.
2. Reasoning AI: At this level, AI systems can solve complex problems at a doctorate level of education without access to external resources. OpenAI believes it is approaching this stage.
3. Autonomous AI: These "Agents" can operate independently for extended periods, making decisions and adapting to changing circumstances without constant human oversight.
4. Innovating AI: At this stage, AI systems can develop groundbreaking ideas and solutions across various fields, driving innovation and progress independently.
5. Organizational AI: The ultimate level where AI systems can function as entire organizations, possessing strategic thinking, operational efficiency, and adaptability to manage complex systems and achieve organizational goals.

This classification system was shared with OpenAI employees on July 9, 2024, and represents the company's structured approach to measuring advancements in AI development.

Definition of an AI Agent from a widely followed academic resource

According to Stuart Russell and Peter Norvig in their textbook "Artificial Intelligence: A Modern Approach," an AI agent is defined as:

"Anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators"

This definition emphasizes two key aspects:

1. Perception: The agent gathers information about its environment using sensors.
2. Action: The agent influences its environment by taking actions through actuators.

An agent implements a function that maps percept sequences to actions. This broad definition encompasses a wide range of AI systems.

Classification of AI Agents

Russell and Norvig classify AI agents into five main categories based on their level of sophistication:

1. Simple Reflex Agents:
    - Act based on current percepts, ignoring past history
    - Use condition-action rules: "if condition, then action"
    - Work best in fully observable environments
    - May get stuck in infinite loops in partially observable environments
2. Model-Based Reflex Agents:
    - Maintain an internal model of the world
    - Can handle partially observable environments
    - Use the model to keep track of unseen parts of the environment
    - Make decisions based on both current percepts and internal state
3. Goal-Based Agents:
    - Have explicit goals and consider future consequences of actions
    - Use searching and planning to choose actions that achieve their goals
    - More flexible than reflex agents as they can adapt to changing goals
4. Utility-Based Agents:
    - Use a utility function to measure the desirability of different states
    - Can compare and choose between different goals based on expected utility
    - Aim to maximize their own "happiness" or satisfaction
5. Learning Agents:
    - Can improve performance over time through experience
    - Have a learning component that modifies their behavior based on observations
    - Can adapt to new environments and improve their decision-making abilities

This classification represents increasing levels of sophistication and capability in AI agents, from simple reactive systems to complex, adaptive learners. Each type builds upon the capabilities of the previous one, adding new functionalities and ways of interacting with the environment.
