Agent Router Module: Documentation

Financial Application Development Team

August 22, 2025

Contents

1 Overview

The `agent_router.py` module is a critical component of the Financial Agents App, responsible for routing user queries to the appropriate specialized agent based on the query's content. It leverages a language model (GPT-4o-mini) for intelligent routing decisions and integrates with a Retrieval-Augmented Generation (RAG) system to provide contextual information. The module uses a state graph workflow to manage the routing and dispatching process, ensuring seamless interaction between the user interface and the agent layer.

This document details the module's architecture, components, data flow, and design decisions. It is intended for developers and architects to understand the routing mechanism and support maintenance or extension of the Financial Agents App.

2 Module Architecture

The `agent_router.py` module follows a modular, workflow-driven architecture, utilizing the `langgraph` library to define a state graph for query processing. The architecture consists of three main components: a router, a dispatcher, and a state management system. The router analyzes the user query and context to select an agent, while the dispatcher invokes the chosen agent to process the query. The module integrates with external services (e.g., OpenAI's GPT-4o-mini) and the internal RAG system for enhanced decision-making.

2.1 Architecture Diagram

The architecture is depicted in the following workflow diagram, illustrating the flow of a user query through the routing and dispatching process:

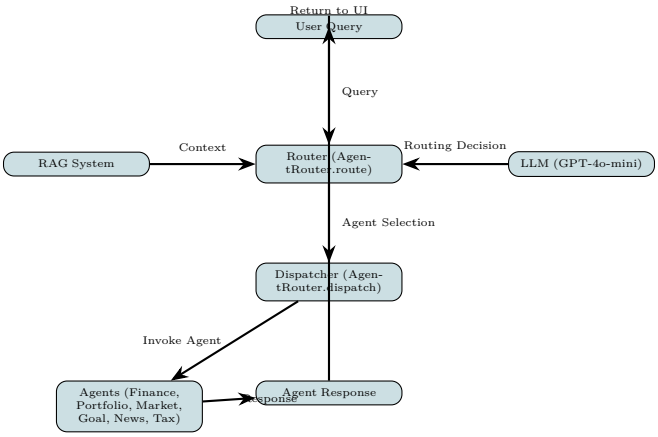

Figure 1: Workflow Diagram for Agent Router Module

3 Components

The `agent_router.py` module consists of the following key components:

3.1 AgentRouter Class

The `AgentRouter` class is the core of the module, encapsulating the routing and dispatching logic. It initializes:

- **LLM**: A `ChatOpenAI` instance (GPT-4o-mini) with structured output (`Route` model) for routing decisions.

- **RAG System**: A `RAGSystem` instance that retrieves context from predefined JSON files (`fin_goal_planning_agent.json`, etc.).
- **Agent Mapping**: A dictionary of available agents (`FinanceQAAgent`, `PortfolioAnalysisAgent`, etc.) for dispatching.

3.2 Route Model

The `Route` class, defined using `pydantic`, ensures structured output for routing decisions. It contains a single field:

- `step`: A string specifying the next agent (`finance`, `portfolio`, `market`, `goal`, `news`, `tax`).

3.3 Workflow

The `create_workflow` function defines a state graph using `langgraph.StateGraph`. It includes:

- **Router Node**: Calls `AgentRouter.route` to determine the appropriate agent.
- **Dispatch Node**: Calls `AgentRouter.dispatch` to invoke the selected agent.
- **Edges**: Connects `START` to `router`, `router` to `dispatch`, and `dispatch` to `END`.

3.4 RAG System

The `RAGSystem` retrieves contextual information from JSON files to inform routing decisions, enhancing the LLM's understanding of the query.

4 Data Flow

The data flow through the `agent_router.py` module is as follows:

1. The user submits a query via the Streamlit UI, which is passed to the workflow as an `AgentState` dictionary.
2. The `router` node (`AgentRouter.route`) retrieves context from the `RAGSystem` and uses the LLM to analyze the query and context.
3. The LLM outputs a `Route` object, setting the `step` field to the selected agent (e.g., `tax`, `market`).
4. The `AgentState` is updated with the `decision` (agent name) and `rag_context`.
5. The `dispatch` node (`AgentRouter.dispatch`) invokes the selected agent using the `get_agents` mapping.
6. The agent processes the query and updates the `AgentState["response"]` with the result.
7. The updated `AgentState` is returned to the Streamlit UI for rendering.

5 Design Decisions

- **Structured Output with Pydantic**: The `Route` model ensures consistent routing decisions by restricting the `step` field to valid agent names.
- **GPT-4o-mini**: Chosen for cost-effective, high-quality natural language processing to interpret complex user queries.
- **RAG Integration**: Enhances routing accuracy by providing contextual data from predefined JSON files, reducing misrouting.
- **State Graph Workflow**: The `langgraph` library provides a flexible, extensible framework for managing query routing and dispatching.
- **Error Handling**: Try-except blocks in `route` and `dispatch` methods ensure robustness, defaulting to the `FinanceQAAgent` on routing errors.
- **Modular Agent Mapping**: The `get_agents` method allows easy addition of new agents without modifying core routing logic.

6 Dependencies

- **Python Libraries**: `streamlit`, `langchain_openai`, `langchain_core`, `pydantic`, `langgraph`.
- **Internal Modules**: `workflow.state`, `rag.rag_system`, `agents` (FinanceQAAgent, PortfolioAnalysisAgent, MarketAnalysisAgent, GoalPlanningAgent, NewsSynthesizerAgent, TaxEducationAgent).
- **External Services**: OpenAI API (for GPT-4o-mini).

7 Usage

The `agent_router.py` module is used within the Financial Agents App as follows:

1. Initialize the workflow in `app.py` using `create_workflow()`.
2. Pass a user query as an `AgentState` dictionary to the workflow's `invoke` method.
3. The workflow routes the query to the appropriate agent and returns the updated `AgentState` with the response.

Example:

```
workflow = create_workflow()
state = workflow.invoke({
    "query": "calculate tax for 50000 wages",
    "agent_name": "",
    "decision": "",
    "rag_context": [],
    "response": {},
    "messages": []
})
```

8 Scalability and Extensibility

- **Scalability**: The state graph architecture supports high query volumes by processing requests sequentially, with potential for parallelization in future iterations.
- **Extensibility**: New agents can be added by updating the `get_agents` method and `Route` model's `step` enum, without modifying the core workflow.
- **Performance**: The RAG system and LLM caching (if enabled) can improve response times for frequent queries.

9 Conclusion

The `agent_router.py` module provides a robust and flexible routing mechanism for the Financial Agents App. By leveraging a language model, RAG system, and state graph workflow, it ensures accurate query routing to specialized agents. The modular design and error handling make it maintainable and extensible, supporting future enhancements such as additional agents or advanced routing logic.