agents.py Documentation

This document provides a comprehensive guide to the `agents.py` file, detailing its purpose, functions, and usage. This file is responsible for creating and managing intelligent agents within the application. These agents leverage Large Language Models (LLMs) from providers like Groq and MistralAI to perform specific tasks, such as resume analysis and cross-job comparison. Get ready to dive in!

Overview

The `agents.py` file is the heart of our agent-based system. It defines how different agents are created, configured, and utilized. These agents use LLMs to provide intelligent services, such as analyzing resumes and comparing job descriptions. The file also includes caching mechanisms to improve performance and reduce API costs. This file is essential for understanding how the application leverages AI to enhance its functionality.

Imports

The file begins by importing necessary libraries and modules. Here's a breakdown:

- `langchain_groq.ChatGroq` : Integrates with the Groq LLM service.
- `langchain_mistralai.ChatMistralAI` : Integrates with the MistralAI LLM service.
- `langchain_google_genai.ChatGoogleGenerativeAI` : Integrates with the Google Gemini LLM service.
- `langchain_core.prompts.ChatPromptTemplate` : Used for creating chat prompts for the LLMs.
- `langchain_community.embeddings.HuggingFaceInferenceAPIEmbeddings` : For generating embeddings using Hugging Face Inference API.
- `langchain_core.language_models.chat_models.BaseChatModel` : Base class for chat models.
- `langchain_core.runnables.RunnableSerializable` : Interface for creating runnable chains.
- `pathlib.Path` : For handling file paths.
- `time` : For time-related tasks (e.g., delays).
- `sys` : For system-specific parameters and functions.
- `os` : For interacting with the operating system (e.g., environment variables).
- `yaml` : For reading YAML files.
- `dotenv.load_dotenv` : For loading environment variables from a `.env` file.
- `data_models.ResumeFeedback` : Data model for resume feedback.
- `data_models.CrossJobMatchResult` : Data model for cross-job match results.
- `utils.CacheManager` : A utility class for caching data.
- `utils.load_prompts` : A utility function for loading prompts.

These imports provide the building blocks for creating and managing the agents, handling API interactions, and processing data.

Constants

The file defines several constants that configure the agents and their behavior:

- `cache_manager` : An instance of `CacheManager` used for caching LLM models and prompts to improve performance.
- `PROMPTS_PATH` : A `Path` object pointing to the `prompts.yaml` file, which contains the prompts used by the agents.
- `cache_imports` : An empty dictionary intended for caching imported modules (currently not used).
- `GROQ_MODELS` : A list of model names available from the Groq service. Current models include "llama-3.3-70b-versatile", "llama-3.1-8b-instant", "qwen-qwq-32b", and "qwen-2.5-32b".
- `MISTRALAI_MODELS` : A list of model names available from the MistralAI service. Current models include "mistral-large-latest" and "ministral-8b-latest".

These constants centralize configuration values, making it easier to manage and update the agents' behavior.

Functions

initialize_llm(model_name: str) -> BaseChatModel

This function initializes and returns a Large Language Model (LLM) based on the specified `model_name`. It supports models from Groq and MistralAI, utilizing a caching mechanism to avoid redundant initializations.

1. Cache Check: It first checks if the model is already cached using the `CacheManager`. If found, it returns the cached model.

2. Model Selection: If the model is not cached, it determines the appropriate LLM provider based on the `model_name`.

3. Groq Model Initialization: If the model is a Groq model, it initializes a `ChatGroq` instance with the specified model name, temperature, and API key (retrieved from environment variables).

4. MistralAI Model Initialization: If the model is a MistralAI model, it initializes a `ChatMistralAI` instance with the

specified API key and model name. Implements retry logic to handle potential API errors.

5. Caching: After initializing the model, it caches the model using the `CacheManager` for future use.

Example:

Let's say you want to initialize the `llama-3.3-70b-versatile` model from Groq:

```
model = initialize_llm("llama-3.3-70b-versatile")
# Now you can use the 'model' for your tasks.
```

This function ensures that LLM models are initialized efficiently and consistently throughout the application.

create_rar_agent() -> RunnableSerializable

This function creates the Resume Analyzer-Reranker (RAR) agent. The agent is designed to analyze resumes and provide feedback. It uses a pre-defined prompt and an LLM to generate structured output in the form of `ResumeFeedback`.

1. Prompt Loading: It retrieves the RAR agent prompt from the cache. If the prompts are not cached, it loads them from the `prompts.yaml` file and caches them.

2. LLM Initialization: It initializes the LLM using the `initialize_llm` function, specifying the "llama-3.3-70b-versatile" model.

3. Structured Output: It configures the LLM to produce structured output using the `ResumeFeedback` data model.

4. Prompt Template: It creates a chat prompt template from the loaded prompt.

5. Agent Chain: It chains the prompt template with the LLM to create the agent chain.

Example:

```
rar_agent = create_rar_agent()
# Now you can use the 'rar_agent' to analyze resumes.
```

This function encapsulates the logic for creating the RAR agent, making it reusable and maintainable.

create_cjc_agent() -> RunnableSerializable

This function creates the Cross Job Comparison (CJC) agent. This agent compares job descriptions and provides a match result. It uses a pre-defined prompt and an LLM to generate structured output in the form of `CrossJobMatchResult`.

1. Prompt Loading: It retrieves the CJC agent prompt from the cache. If the prompts are not cached, it loads them from the `prompts.yaml` file and caches them.

2. LLM Initialization: It initializes the LLM using the `initialize_llm` function, specifying the "mistral-large-latest" model.

3. Structured Output: It configures the LLM to produce structured output using the `CrossJobMatchResult` data model.

4. Prompt Template: It creates a chat prompt template from the loaded prompt.

5. Agent Chain: It chains the prompt template with the LLM to create the agent chain.

Example:

```
cjc_agent = create_cjc_agent()
# Now you can use the 'cjc_agent' to compare job descriptions.
```

This function encapsulates the logic for creating the CJC agent, ensuring consistency and reusability.

Documentation generated on [fill-info: Date].

For further assistance, contact [fill-info: Author/Team Name] at [fill-info: Contact Email/Details].