File Breakdown: `src/utils/open_ai_utils.py`

File Location

`src/utils/open_ai_utils.py`

Overview

The `open_ai_utils.py` file implements utilities for interacting with OpenAI's API and other LLM services. It provides functions for making API calls, running inference, categorizing prompts, and handling both synchronous and asynchronous operations. This module serves as a bridge between the multi-agent system and external LLM capabilities.

Key Responsibilities

- Make API calls to OpenAI and other LLM providers
- Provide both synchronous and asynchronous versions of API functions
- Categorize user prompts based on function capabilities
- Handle chat sessions for natural conversations
- Implement error handling and retries for API failures
- Support different LLM models with appropriate parameters
- Format API requests and process responses

Core Functionality

API Calls

The core function for making synchronous API calls to OpenAI:

```
@log_function_call
def run_open_ai_ns(message, context, temperature=0.7, top_p=1.0, model="o3-mini",
max_tokens=500, verbose=False):
    """
    Call OpenAI API to generate a response based on the input message and context.
    This is the most complete implementation of the function with reduced verbosity.

    Parameters:
    - message (str): The message to send to the API
    - context (str): System context/instructions
    - temperature (float): Controls randomness in generation
    - top_p (float): Top-p sampling parameter
    - model (str): The model to use
    - max_tokens (int): Maximum number of tokens to generate
    - verbose (bool): Whether to print debug information

    Returns:
    - str: The API response content
    """
    import time

    # Only print debug info if verbose is True
```

```
    if verbose:
        print(f"Starting OpenAI API call to model: {model}")
        print(f"Context length: {len(context)} chars")
        print(f"Message length: {len(message)} chars")

    start_time = time.time()
    try:
        if model == "o3-mini":
            # For o3-mini, the temperature parameter is not supported.
            response = openai.chat.completions.create(
                model=model,
                messages=[
                    {"role": "system", "content": context},
                    {"role": "user", "content": message}
                ],
                top_p=top_p,
                frequency_penalty=0.0,
                presence_penalty=0.0,
            )
            AI_response = response.choices[0].message.content

            if verbose:
                elapsed = time.time() - start_time
                print(f"API call completed in {elapsed:.2f} seconds")

            return AI_response

        # Handle other models...
        elif 'gpt' in model.lower():
            response = openai.chat.completions.create(
                model=model,
                messages=[
                    {"role": "system", "content": context},
                    {"role": "user", "content": message}
                ],
                temperature=temperature,
                top_p=top_p,
                frequency_penalty=0.0,
                presence_penalty=0.0,
            )
            AI_response = response.choices[0].message.content

            if verbose:
                elapsed = time.time() - start_time
                print(f"API call completed in {elapsed:.2f} seconds")

            return AI_response

        # Other model handlers...
        else:
            return "Unsupported model type"
```

```
    except Exception as e:
        elapsed = time.time() - start_time
        print(f"❌ API call failed after {elapsed:.2f} seconds: {str(e)}")

        # Fallback: If an unsupported_parameter error occurs, retry with minimal
parameters.
        if "unsupported_parameter" in str(e).lower():
            if verbose:
                print("Retrying with minimal parameters...")

            minimal_params = {
                "model": model,
                "messages": [
                    {"role": "system", "content": context},
                    {"role": "user", "content": message}
                ]
            }
            try:
                fallback_start_time = time.time()
                response = openai.chat.completions.create(**minimal_params)
                AI_response = response.choices[0].message.content
                return AI_response
            except Exception as fallback_error:
                print(f"❌ Fallback also failed: {str(fallback_error)}")
                return '{}'  # Last resort: return empty result
        return '{}'  # Return empty result on error
```

Asynchronous API Calls

Asynchronous version for non-blocking API calls:

```
@log_function_call
async def run_open_ai_ns_async(message, context, temperature=0.7, top_p=1.0,
model="o3-mini", max_tokens=500, verbose=False):
    """
    Asynchronous version of run_open_ai_ns.
    Uses aiohttp for non-blocking API calls.

    Parameters:
        message (str): The message to send to the API
        context (str): System context/instructions
        temperature (float): Controls randomness in generation
        top_p (float): Top-p sampling parameter
        model (str): The model to use
        max_tokens (int): Maximum number of tokens to generate
        verbose (bool): Whether to print debug information

    Returns:
        str: The API response content
    """
    import time

    # Only print debug info if verbose is True
```

```
    if verbose:
        print(f"Starting async OpenAI API call to model: {model}")
        print(f"Context length: {len(context)} chars")
        print(f"Message length: {len(message)} chars")

    start_time = time.time()

    # Create the API request parameters
    params = {
        "model": model,
        "messages": [
            {"role": "system", "content": context},
            {"role": "user", "content": message}
        ]
    }

    # Add model-specific parameters
    if model != "o3-mini":  # o3-mini doesn't support temperature
        params["temperature"] = temperature

    # Add top_p if provided
    if top_p is not None:
        params["top_p"] = top_p

    # Other parameters for certain models
    if 'gpt' in model.lower() or 'claude' in model.lower():
        params["frequency_penalty"] = 0.0
        params["presence_penalty"] = 0.0

    # Headers for the API request
    API_KEY = get_api_key('openai')

    headers = {
        "Content-Type": "application/json",
        "Authorization": f"Bearer {API_KEY}"
    }

    try:
        # Make the API call using aiohttp
        async with aiohttp.ClientSession() as session:
            async with session.post(
                "https://api.openai.com/v1/chat/completions",
                json=params,
                headers=headers
            ) as response:
                if response.status == 200:
                    response_json = await response.json()
                    AI_response = response_json["choices"][0]["message"]["content"]

                    if verbose:
                        elapsed = time.time() - start_time
                        print(f"API call completed in {elapsed:.2f} seconds")
```

```
                    return AI_response
                else:
                    error_text = await response.text()
                    print(f"API error: {response.status} - {error_text}")

                    # Try fallback with minimal parameters
                    if "unsupported_parameter" in error_text.lower():
                        if verbose:
                            print("Retrying with minimal parameters...")

                        # Create minimal params
                        minimal_params = {
                            "model": model,
                            "messages": params["messages"]
                        }

                        # Try again with minimal params
                        async with session.post(
                            "https://api.openai.com/v1/chat/completions",
                            json=minimal_params,
                            headers=headers
                        ) as fallback_response:
                            if fallback_response.status == 200:
                                fallback_json = await fallback_response.json()
                                return fallback_json["choices"][0]["message"]
["content"]
                            else:
                                return '{}'  # Return empty result on error
                    return '{}'  # Return empty result on error
    except Exception as e:
        elapsed = time.time() - start_time
        print(f"❌ API call failed after {elapsed:.2f} seconds: {str(e)}")
        return '{}'  # Return empty result on error
```

Prompt Categorization

Function for categorizing user prompts into specific function types:

```
@log_function_call
async def open_ai_categorisation_async(question, function_map, level=None):
    """
    Asynchronous version of open_ai_categorisation.

    Parameters:
        question (str or object): The user's question or prompt
        function_map (str): Path to the function map CSV file
        level (str, optional): Task level ('task list' or None)

    Returns:
        str: The categorized function key
    """
    if not isinstance(question, str):
```

```
        question = str(question)

    # Load category descriptions
    import pandas as pd

    try:
        df = pd.read_csv(function_map)
        categories = dict(zip(df['Key'], df['Description']))
    except Exception as e:
        print(f"Error loading categories from {function_map}: {str(e)}")
        categories = {}

    # Add general_question to categories if not present
    if "general_question" not in categories:
        categories["general_question"] = "General knowledge questions, basic
arithmetic, or factual queries"

    if level == 'task list' and 'Create task list' in categories:
        categories.pop('Create task list')

    # Energy model keywords for pre-check
    energy_model_keywords = [
        'energy model', 'model energy', 'build model',
        'create model', 'design model', 'energy system',
        'power system', 'renewable model', 'electricity model',
        'build a model'
    ]

    # Pre-check for energy model
    if any(keyword in question.lower() for keyword in energy_model_keywords):
        print(f"  Pre-check identified energy modeling request: '{question}'")
        return "Energy Model"

    # System message for the API
    category_info = ", ".join([f"'{key}': {desc}" for key, desc in
categories.items()])

    system_msg = (
        f"You are an assistant trained to categorize questions into specific
functions. "
        f"Here are the available categories with descriptions: {category_info}. "
        f"If none of the categories are appropriate, categorize as 'general_question'.
"
        f"Please respond with only the category from the list given, with no
additional text."
    )

    # Import math patterns for pre-check
    import re
    math_patterns = [
        r'\d+\s*[\+\-\*\/]\s*\d+',  # Simple operations like 2+2
        r'what\s+is\s+\d+\s*[\+\-\*\/]\s*\d+',  # "What is 2+2"
```

```
        r'calculate\s+\d+',  # "Calculate 25"
    ]

    # Pre-check for math
    if any(re.search(pattern, question.lower()) for pattern in math_patterns):
        print(f"🔢 Pre-check identified math problem: '{question}'")
        return "do_maths"

    # Check for website patterns
    website_patterns = [
        r'open\s+.*website',
        r'go\s+to\s+.*site',
        r'visit\s+.*page',
        r'browse\s+to',
        r'open\s+.*page'
    ]

    if any(re.search(pattern, question.lower()) for pattern in website_patterns):
        print(f"🌐 Pre-check identified website opening: '{question}'")
        return "Open a website"

    try:
        # Call OpenAI API using async version
        response = await run_open_ai_ns_async(question, system_msg)
        category = response.strip()

        # Clean up response
        category = category.replace("'", "").replace("\"", "").strip()

        # Final check for energy model
        if category.lower() in ['uncategorized', 'general_question'] and \
           any(keyword in question.lower() for keyword in energy_model_keywords):
            category = "Energy Model"

        print(f"🤖 OpenAI categorization: {category}")
        return category
    except Exception as e:
        print(f"❌ Error in OpenAI categorization: {str(e)}")

        # Fallback based on keywords in question
        lower_question = question.lower()
        if any(re.search(pattern, lower_question) for pattern in website_patterns):
            return "Open a website"
        elif any(re.search(pattern, lower_question) for pattern in math_patterns):
            return "do_maths"
        elif any(keyword in lower_question for keyword in energy_model_keywords):
            return "Energy Model"
        return 'general_question'
```

Chat Session Implementation

Method for handling chat sessions:

```python
@log_function_call
def ai_chat_session(kb, prompt=None):
    """
    Handles a general AI chat session.

    Parameters:
        kb (KnowledgeBase): The knowledge base
        prompt (str, optional): Initial prompt to start the conversation

    Returns:
        str: The result of the chat session
    """
    client = OpenAI(base_url="http://localhost:1234/v1", api_key="lm-studio")
    print(f"Starting chat session with prompt: {prompt}")

    history = [
        {"role": "system", "content": "You are a friendly, intelligent assistant. You always provide well-reasoned answers that are both correct and helpful. Keep your responses concise and to the point."},
        {"role": "user", "content": f"Hello, you have been requested. Here is the prompt: {prompt}"},
    ]

    try:
        # Single response mode since we're in a script context
        completion = client.chat.completions.create(
            model="bartowski/Phi-3-medium-128k-instruct-GGUF",
            messages=history,
            temperature=0.7
        )

        response = completion.choices[0].message.content

        # Store the result in the knowledge base
        kb.set_item("chat_result", response)
        kb.set_item("final_report", response)  # This will be displayed at the end

        return response
    except Exception as e:
        error_msg = f"Error in chat session: {str(e)}"
        print(error_msg)
        return error_msg
```

Key Features

1. **Multiple LLM Support**: Works with different models and API endpoints
2. **Async/Sync API**: Provides both synchronous and asynchronous implementations
3. **Error Handling**: Implements robust error handling and fallback mechanisms
4. **Pattern Recognition**: Uses regex patterns for pre-checking certain query types
5. **Model-Specific Parameters**: Handles different parameter requirements for various models
6. **API Optimization**: Minimizes unnecessary parameters to avoid API errors

7. **Performance Monitoring**: Tracks API call duration for performance monitoring
8. **Chat Sessions**: Implements chat sessions for conversational interactions

Integration

- Used by all agents to access LLM capabilities
- Supports the prompt categorization system for routing queries
- Enables general knowledge question answering
- Powers chat sessions for conversational interactions
- Provides error handling and fallbacks for API issues

Workflow

1. Receive a message and context (system prompt)
2. Determine appropriate parameters based on the target model
3. Make the API call with appropriate error handling
4. Process and return the response
5. For categorization:
    - Pre-check common patterns (math, website, energy model)
    - Fall back to the LLM for more complex categorization
    - Clean and validate the response
6. For chat sessions:
    - Manage conversation history
    - Generate responses using the appropriate model
    - Store results in the knowledge base

Implementation Notes

- Uses the OpenAI client library for API access
- Implements aiohttp for asynchronous API calls
- Provides detailed error information for debugging
- Optimizes API parameters based on model requirements
- Supports local model servers for development and testing
- Uses pattern matching for more efficient categorization
- Implements both function-specific and general-purpose helpers