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

File Location

`src/utils/open_ai_calls.py`

Overview

The `open_ai_calls.py` file implements a comprehensive set of functions for interacting with OpenAI and other LLM APIs. It provides various methods for making API calls with different parameters, formats, and handling mechanisms. This file serves as the foundation for the system's LLM capabilities, supporting different models, response formats, and usage scenarios.

Key Responsibilities

- Initialize API clients with appropriate authentication
- Make API calls to OpenAI, Groq, Deepseek, and other LLM providers
- Support different response formats (text, JSON)
- Provide functions for chat sessions (written and spoken)
- Handle API errors and implement fallback mechanisms
- Process responses for use by the wider system
- Support image generation with DALL-E
- Implement prompt categorization

Core Functionality

API Key Management

```
# Set the API_KEY variable at the module level
from .get_api_keys import get_api_key

API_KEY = get_api_key('openai')
if API_KEY:
    os.environ['OPENAI_API_KEY'] = API_KEY
else:
    print("Failed to load API key.")

GROQ_API_KEY = get_api_key('groq')
os.environ['GROQ_API_KEY'] = GROQ_API_KEY
perplexity_key = "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"
```

Basic OpenAI Call

```
def run_open_ai(message, context, temperature=0.7, top_p=1.0):
    global sound_playing
    chat_log = []
    sound_path = play_chime('speech_dis')  # play_chime returns a valid path or None
    sound_thread = threading.Thread(target=lambda: None)  # Initialize thread with a
dummy function to handle cases where sound_path is None
    if sound_path:
```

```
        sound_thread = threading.Thread(target=play_sound_on_off, args=(sound_path,))
        sound_thread.start()
    response = openai.ChatCompletions.create(
        model="o3-mini",
        messages=[
            {"role": "system", "content": context},
            {"role": "user", "content": [{"type": "text", "text": message}]},
        ],
        temperature=temperature,
        top_p=top_p,
        frequency_penalty=0.0,
        presence_penalty=0.0,
    )
    AI_response = response.choices[0].message.content
    chat_log.append({'role': 'assistant', 'content': AI_response.strip('\n').strip()})
    if sound_thread.is_alive():
        sound_thread.join()
    return AI_response
```

Enhanced No-Streaming OpenAI Call

The file contains multiple implementations of this function, with the most recent being the most robust:

```
def run_open_ai_ns(message, context, temperature=0.7, top_p=1.0, model="o3-mini",
max_tokens=500):
    import time
    import json
    print("DEBUG: Starting OpenAI API call")
    print("DEBUG: Model:", model)
    print("DEBUG: Context:", context)
    print("DEBUG: Message:", message)
    start_time = time.time()
    try:
        if model == "o3-mini":
            # For o3-mini, the temperature parameter is not supported.
            # The user message is passed as a simple string.
            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,
            )
            elapsed = time.time() - start_time
            print(f"open_ai_call.py: \n DEBUG [o3-mini]: API call completed in
{elapsed:.2f} seconds")
            print("DEBUG [o3-mini]: Full API response:", response)
            AI_response = response.choices[0].message.content
            print("DEBUG [o3-mini]: Extracted AI response:", AI_response)
```

```
            return AI_response

        elif 'gpt' in model:
            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,
            )
            elapsed = time.time() - start_time
            print(f"open_ai_call.py: \n DEBUG [GPT]: API call completed in
{elapsed:.2f} seconds")
            print("DEBUG [GPT]: Full API response:", response)
            AI_response = response.choices[0].message.content
            print("DEBUG [GPT]: Extracted AI response:", AI_response)
            return AI_response

        # Handle other model types (o1, studio, groq, deepseek, sonar, test)
        # ...
    except Exception as e:
        elapsed = time.time() - start_time
        print(f"open_ai_call.py: \n DEBUG: API call failed after {elapsed:.2f}
seconds: {e}")
        # Fallback: If we get an unsupported_parameter error, try minimal parameters
        if "unsupported_parameter" in str(e).lower():
            print("DEBUG: Retrying with minimal parameters...")
            minimal_params = {
                "model": model,
                "messages": [
                    {"role": "system", "content": context},
                    {"role": "user", "content": message}
                ]
            }
            try:
                fallback_start_time = time.time()
                print("DEBUG: Sending fallback request with minimal parameters...")
                response = openai.chat.completions.create(**minimal_params)
                fallback_elapsed = time.time() - fallback_start_time
                print(f"open_ai_call.py: \n DEBUG: Fallback successful in
{fallback_elapsed:.2f} seconds")
                AI_response = response.choices[0].message.content
                print("DEBUG: Fallback extracted AI response:", AI_response)
                return AI_response
            except Exception as fallback_error:
                fallback_elapsed = time.time() - fallback_start_time
                print(f"open_ai_call.py: \n DEBUG: Fallback also failed after
{fallback_elapsed:.2f} seconds: {fallback_error}")
```

```
                return '{}'  # Return empty JSON object as last resort
        return '{}'
```

JSON Response Format

```
def run_open_ai_json(message, context, temperature=None, top_p=1.0, model="o3-mini"):
    """
    Call OpenAI API to get a JSON-formatted response.
    Modified to handle models that don't support certain parameters.
    Enhanced with detailed logging to verify API calls.

    Parameters:
    - message (str): The message to send to the API
    - context (str): System context/instructions
    - temperature (float, optional): Temperature parameter for models that support it
    - top_p (float): Top-p sampling parameter
    - model (str): The model to use

    Returns:
    - str: The JSON response content
    """
    import json
    import time
    import uuid
    import logging

    # Generate a unique request ID
    request_id = str(uuid.uuid4())[:8]

    print(f"open_ai_call.py: \n \n [Request {request_id}] Preparing OpenAI API call
to model: {model}")
    print(f"open_ai_call.py: \n  Context length: {len(context)} chars")
    print(f"open_ai_call.py: \n  Message length: {len(message)} chars")

    # Base params that should work with all models
    params = {
        "model": model,
        "response_format": {"type": "json_object"},
        "messages": [
            {"role": "system", "content": context},
            {"role": "user", "content": [{"type": "text", "text": message}]},
        ],
    }

    # Log which parameters we're using
    used_params = ["model", "response_format", "messages"]

    # Conditionally add parameters based on model
    if 'o3' not in model.lower():  # o3-mini doesn't support temperature
        if temperature is not None:
            params["temperature"] = temperature
            used_params.append("temperature")
```

```
    if top_p is not None:
        params["top_p"] = top_p
        used_params.append("top_p")

    # These parameters may not be supported by all models, add conditionally
    if 'gpt' in model.lower() or 'claude' in model.lower():
        params["frequency_penalty"] = 0.0
        params["presence_penalty"] = 0.0
        used_params.extend(["frequency_penalty", "presence_penalty"])

    # Start timer
    start_time = time.time()

    try:
        # Actually make the API call
        response = openai.chat.completions.create(**params)

        # Calculate elapsed time
        elapsed_time = time.time() - start_time

        # Extract response
        AI_response = response.choices[0].message.content

        # Log success
        print(f"open_ai_call.py: \n  [Request {request_id}] OpenAI API call
successful in {elapsed_time:.2f} seconds")

        # Try to validate it's actually JSON
        try:
            json.loads(AI_response)
        except json.JSONDecodeError:
            print(f"open_ai_call.py: \n  Warning: Response is not valid JSON")

        return AI_response
    except Exception as e:
        # Calculate elapsed time even for failures
        elapsed_time = time.time() - start_time

        print(f"open_ai_call.py: \n  [Request {request_id}] Error in OpenAI API call
after {elapsed_time:.2f} seconds")
        print(f"open_ai_call.py: \n  Error type: {type(e).__name__}")
        print(f"open_ai_call.py: \n  Error details: {str(e)}")

        # Fallback: Try again with minimal parameters if we got parameter errors
        if "unsupported_parameter" in str(e).lower():
            print(f"open_ai_call.py: \n  [Request {request_id}] Retrying with minimal
parameters...")

            # Create minimal params and retry
            # ...
```

```
        # Return empty JSON object on error
        return '{}'
```

Chat Sessions

```
def ai_chat_session(prompt=None):
    client = OpenAI(base_url="http://localhost:1234/v1", api_key="lm-studio")
    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}"},
    ]
    while True:
        completion = client.chat.completions.create(
            model="bartowski/Phi-3-medium-128k-instruct-GGUF",
            messages=history,
            temperature=0.7,
            stream=True,
        )
        new_message = {"role": "assistant", "content": ""}
        for chunk in completion:
            if chunk.choices[0].delta.content:
                print(chunk.choices[0].delta.content, end="", flush=True)
                new_message["content"] += chunk.choices[0].delta.content
        history.append(new_message)
        print()
        history.append({"role": "user", "content": input("> ")})
```

Prompt Categorization

```
def open_ai_categorisation(question, function_map, level=None):
    """
    Categorize user questions into specific functions using OpenAI.

    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)

    categories = load_category_descriptions(function_map)

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

```
    # Enhanced instruction for energy modeling tasks with more explicit guidance
    additional_instruction = (
        "IMPORTANT: If the prompt contains ANY of the following: "
        "- Any form of 'run a model', 'create model', 'execute model' "
        "- Any reference to energy modeling "
        "- Words such as 'model', 'solar', 'energy', 'generation', 'run model',
'electricity' "
        "- Any request to simulate or analyze energy systems "
        "THEN you must categorize it as 'Energy Model'. "
        "The Energy Model category takes priority over Data analysis for any model-
related queries."
    )

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

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

    try:
        response = openai.chat.completions.create(
            model="o3-mini",
            messages=[
                {"role": "system", "content": system_msg},
                {"role": "user", "content": [{"type": "text", "text": question}]}
            ],
            top_p=1.0
            # Removed temperature parameter as it is unsupported
        )
        category = response.choices[0].message.content.strip()

        # Enhanced fallback logic to catch misclassifications
        model_related_keywords = ['model', 'run', 'solar', 'energy', 'generation',
'electricity']
        model_phrases = ['run a model', 'create model', 'execute model', 'run model']

        # Check for model-related phrases or keywords
        is_model_related = any(phrase in question.lower() for phrase in model_phrases)
or \
                           (any(word in question.lower() for word in
model_related_keywords) and 'model' in question.lower())

        # Override if it's clearly model-related but was misclassified
        if is_model_related and category != "Energy Model":
            print(f"open_ai_call.py: \n  Overriding '{category}' to 'Energy Model' -
```

```
detected model-related query")
            category = "Energy Model"
        # Also keep the original fallback for Uncategorized
        elif category.lower() == 'uncategorized' and any(word in question.lower() for
word in model_related_keywords):
            category = "Energy Model"

        print(f"open_ai_call.py: \n  OpenAI Response (open_ai_calls.py): {category}")
        return category
    except Exception as e:
        print(f"open_ai_call.py: \n  Error in OpenAI categorization: {str(e)}")
        return 'Uncategorized'
```

Key Features

1. **Multiple Model Support**: Works with various LLM providers (OpenAI, Groq, Deepseek, Perplexity)
2. **Response Format Options**: Supports text and JSON response formats
3. **Streaming Support**: Implements streaming responses for interactive chat sessions
4. **Error Handling**: Provides robust error handling with detailed logging
5. **Fallback Mechanisms**: Implements parameter reduction for API compatibility
6. **Local Model Support**: Works with locally hosted models via LM Studio
7. **Categorization**: Implements specialized handling for prompt categorization
8. **Parameter Flexibility**: Adapts API parameters based on model requirements
9. **Performance Monitoring**: Tracks and reports API call performance metrics

Integration

- Used throughout the system for LLM-based capabilities
- Serves as the foundation for intent detection and categorization
- Enables general knowledge question answering
- Powers interactive chat sessions
- Provides the backing for Nova's coordination capabilities

Workflow

1. Initialize API clients with appropriate authentication
2. Receive requests for LLM processing
3. Format requests based on model requirements
4. Make API calls with appropriate error handling
5. Process and return responses
6. For errors, implement fallback strategies and detailed logging
7. For specialized tasks (categorization, chat), implement specific workflows

Implementation Notes

- Contains multiple implementations of similar functions, likely due to iterative development
- Includes detailed debugging output for troubleshooting
- Handles various model-specific requirements and limitations
- Uses ID-based request tracking for monitoring multi-request workflows
- Implements comprehensive error handling for API reliability
- Supports both local and cloud-based LLM services