File Breakdown: `src/agents/nova.py`

Overview

The `nova.py` file implements the Nova agent, which functions as the central coordination agent in a multi-agent system. Nova is responsible for parsing user requests, identifying intents, creating appropriate tasks, and delegating them to specialized agents (Emil, Ivan, and Lola).

Key Responsibilities

- Parsing and understanding user prompts
- Detecting multiple intents within a single request
- Creating task lists for execution
- Delegating tasks to appropriate specialized agents
- Collecting required parameters when necessary
- Managing website opening functionality

Core Functionality

Class Definition

```
class Nova(BaseAgent):
    # Inherits from BaseAgent class
    # Implements coordination logic for the multi-agent system
```

Task Handling

Nova can handle tasks both synchronously and asynchronously:

```
@log_function_call
def handle_task(self, task: Task):
    """
    Handles a task by executing the appropriate function based on the task's function_name.
    """
    print(f"Nova handling task: {task.name} with function: {task.function_name}")

    if task.function_name and task.function_name in self.function_map:
        # Function exists in function map, call it
        func = self.function_map[task.function_name]
        try:
            result = func(self.kb, **task.args)
            # Store the result in the task object
            task.result = result
            # Also store it in the knowledge base with a unique key
            result_key = f"{task.function_name}_result_{id(task)}"
            self.kb.set_item(result_key, result)
            return result
        except Exception as e:
            error_msg = f"Error executing {task.function_name}: {str(e)}"
            print(f"ERROR: {error_msg}")
```

```
            # Store error in task
            task.result = error_msg
            return error_msg
    else:
        # Handle case where function is not found
        # ...
```

Asynchronous version:

```
@log_function_call
async def handle_task_async(self, task: Task):
    """
    Asynchronous version of handle_task.
    """
    print(f"Nova handling task: {task.name} with function: {task.function_name}")
    import time
    start_time = time.time()

    # Log the start of task execution
    self.kb.log_interaction(f"Task: {task.name}", "Starting execution",
                            agent="Nova", function=task.function_name)

    if task.function_name and task.function_name in self.function_map:
        # Function exists in function map
        func = self.function_map[task.function_name]

        try:
            # Check if there's an async version available
            async_func = self.async_function_map.get(task.function_name)
            if async_func:
                # Use the async version
                result = await async_func(self.kb, **task.args)
            else:
                # Run the synchronous function in a thread pool
                result = await asyncio.to_thread(func, self.kb, **task.args)

            # Store the result and log completion
            # ...
        except Exception as e:
            # Handle errors...
```

Intent Recognition

Nova can identify multiple intents within a single user prompt:

```
@log_function_call
async def identify_multiple_intents_async(self, prompt: str) -> List[Dict[str, str]]:
    """
    Asynchronous method to identify multiple intents in a single prompt.
    """
    # Skip multi-intent detection for very short prompts
    if len(prompt.split()) < 5:
        return [{"intent": prompt}]
```

```
    # Use LLM to identify multiple intents with explicit formatting instructions
    context = (
        "You are an intent detection assistant. Your task is to identify if a prompt
contains "
        "multiple separate requests or intents. If it does, break it down into
separate intents. "
        "If the prompt is a single cohesive request, return it as a single
intent.\n\n"
        "Common indicators of multiple intents include: 'and', 'also', 'plus',
'additionally', etc.\n\n"
        "Example: 'open website for France government and build energy model for
France' contains two separate intents.\n\n"
        "IMPORTANT: You must respond with valid JSON. Always format your response as
follows:\n"
        "```json\n"
        "{\n"
        "  \"intents\": [\n"
        "    {\"intent\": \"first intent\"},\n"
        "    {\"intent\": \"second intent\"}\n"
        "  ]\n"
        "}\n"
        "```\n"
    )

    try:
        # Attempt to get JSON response from LLM
        json_response = await run_open_ai_ns_async(prompt, context)

        # Parse the JSON response and extract intents
        # ...
    except Exception as e:
        print(f"Error detecting multiple intents: {str(e)}")
        # Fallback to basic splitting
        # ...
```

Task Creation

Based on identified intents and their categories, Nova creates appropriate tasks:

```
@log_function_call
async def create_task_list_from_prompt_async(self, prompt: str) -> List[Task]:
    """
    Asynchronous version of create_task_list_from_prompt.
    Parses the high-level user prompt into a list of tasks.
    """
    tasks = []
    from utils.open_ai_utils import open_ai_categorisation_async
    import os

    # Add logging at the beginning
    self.kb.log_interaction(prompt, "Processing prompt to create tasks",
                            agent="Nova", function="create_task_list_from_prompt_async")
```

```
    # First, check if this is a history-related query
    prompt_lower = prompt.lower()

    # Look for history query patterns
    is_history_query = False

    # Check for past tense indicators + question words
    past_indicators = ["did", "was", "asked", "had", "previous", "earlier", "last"]
    question_words = ["what", "which", "when", "where", "how", "tell me"]

    # If we have both a past indicator and a question word, it's likely a history
query
    if (any(word in prompt_lower for word in past_indicators) and
        any(word in prompt_lower for word in question_words)):
        # Create history query task
        # ...

    # Otherwise, proceed with normal intent detection and categorization
    # ...

    # Process each intent separately
    for intent_info in multiple_intents:
        intent_text = intent_info["intent"]

        # Categorize this specific intent using async method
        category = await open_ai_categorisation_async(intent_text, csv_path)
        print(f"Intent '{intent_text}' categorized as: {category}")

        # Create a task based on the category
        task = await self.create_task_for_category(intent_text, category)
        tasks.append(task)

    return tasks
```

Task Routing Based on Categories

```
@log_function_call
async def create_task_for_category(self, intent_text: str, category: str) -> Task:
    """
    Create a task for a specific intent based on its category.
    """
    print(f"Creating task for intent: '{intent_text}' (category: {category})")

    # Text pattern matching - to catch cases where categorization might miss
    intent_lower = intent_text.lower()

    # Check for image generation keywords
    image_keywords = ["image", "picture", "drawing", "sketch", "illustration",
"photo", "artwork", "draw"]
    creation_keywords = ["create", "generate", "make", "design", "produce"]
```

```
    is_image_request = category.lower() == "uncategorized" and (
        any(word in intent_lower for word in image_keywords) and
        any(word in intent_lower for word in creation_keywords)
    )

    if is_image_request:
        print(f"Detected image generation request: '{intent_text}'")
        # Route to Ivan for image generation
        return Task(
            name=f"Generate image: {intent_text[:30]}...",
            description=f"Generate image based on prompt",
            agent="Ivan",
            function_name="generate_image",
            args={"prompt": intent_text}
        )

    # Energy modeling - routes to Emil
    if category.lower() in ["energy model", "energy modeling", "build model"]:
        target_agent = "Emil"
        function_name = "process_emil_request"

        # Extract initial parameters from the prompt
        from core.functions_registery import extract_model_parameters
        params = extract_model_parameters(intent_text)
        task_args = {"prompt": intent_text}

        # Check missing parameters and collect them if needed
        # ...

    # Data Analysis - route to Emil
    elif category.lower() in ["data analysis", "analyze results", "analyse results",
"analysis"]:
        target_agent = "Emil"
        function_name = "analyze_results"
        # ...

    # Report writing - route to Lola
    elif category.lower() in ["write report", "write a report", "report writing",
"create report"]:
        target_agent = "Lola"
        function_name = "write_report"
        # ...

    # Create and return the final task
    return Task(
        name=f"Handle Intent: {intent_text[:30]}...",
        description=f"Process intent categorized as {category}",
        agent=target_agent,
        function_name=function_name,
        args=task_args
    )
```

Website Opening Implementation

```
def open_website(kb, prompt, input2="-"):
    """
    Dynamically generates a website URL from the prompt using the LLM.
    """
    from utils.open_ai_utils import run_open_ai_ns
    import re

    # Check if prompt contains a direct URL
    url_match = re.search(r'https?://\S+', prompt)
    if url_match:
        url = url_match.group(0)
        print(f"Found direct URL in prompt: {url}")
    else:
        # Build a system message that instructs the LLM to generate a valid URL.
        context = (
            "You are a URL generation assistant. "
            "Given a prompt like 'open the wikipedia website' or 'open the website for
OpenAI', "
            "output only a valid URL for that website. "
            "Be accurate with government websites. For example:"
            "- 'france government website' should return
'https://www.gouvernement.fr/'"
            "- 'spain government website' should return
'https://www.lamoncloa.gob.es/'"
            "Do not include explanations or additional text, just the URL."
        )

        # Use the LLM to generate the URL.
        url = run_open_ai_ns(prompt, context, model="o3-mini",
temperature=0.0).strip()

    # Validate URL: if empty or not starting with 'http', fall back to a Google search
URL.
    if not url or not url.startswith("http"):
        query = urllib.parse.quote(prompt)
        url = f"https://www.google.com/search?q={query}"

    print(f"Website URL: {url}")

    # Attempt to open the URL in a new browser tab
    opened = webbrowser.open_new_tab(url)
    print(f"webbrowser.open_new_tab returned: {opened}")

    # Store in knowledge base
    kb.set_item("last_opened_website", url)

    return f"Website opened: {url}"
```

Parameter Collection

```python
@log_function_call
async def get_energy_parameters_from_user_async(self, missing_params: List[str]) ->
Dict[str, Any]:
    """
    Asynchronous method to collect specific energy modeling parameters from the user.
    """
    import asyncio
    collected_args = {}

    print("\nNova needs to collect information about your energy model...")

    # Parameter descriptions
    param_descriptions = {
        "location": "The geographic location for the energy model (e.g., UK, France,
Spain, etc.)",
        "generation": "The generation type for the model (e.g., solar, wind, hydro,
thermal, bio)",
        "energy_carrier": "The energy carrier to model (e.g., electricity, hydrogen,
methane)"
    }

    # Parameter examples
    param_examples = {
        "location": "UK, France, Germany, or 'all' for all available locations",
        "generation": "solar, wind, hydro, thermal, bio, or 'all' for all types",
        "energy_carrier": "electricity (default), hydrogen, methane"
    }

    for param in missing_params:
        # Get description and examples from our predefined dictionaries
        description = param_descriptions.get(param, f"The {param} parameter")
        examples = param_examples.get(param, "No examples available")

        # Create a simple prompt
        print(f"\nNova: I need the '{param}' for this task.")
        print(f"Description: {description}")
        print(f"Examples: {examples}")

        # Get user input using asyncio to run input() in a thread
        user_response = await asyncio.to_thread(input, "> ")
        user_response = user_response.strip()

        # Store the response
        collected_args[param] = user_response

    return collected_args
```

Workflow

1. User submits a prompt to Nova
2. Nova identifies all intents in the prompt using `identify_multiple_intents_async`
3. For each intent, Nova:

    - Determines the appropriate category using `open_ai_categorisation_async`
    - Creates a task with the right agent assignment using `create_task_for_category`
    - Adds necessary parameters
4. Tasks are returned for execution by the task manager
5. Results are stored in the knowledge base

Integration

- Interfaces with the knowledge base to store and retrieve information
- Coordinates with specialized agents (Emil, Ivan, Lola)
- Uses LLM services for intent detection and categorization
- Maintains session history for context-aware responses