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

File Location

`src/agents/emil.py`

Overview

The `emil.py` file implements the Emil agent, which specializes in energy modeling and analysis. Emil is responsible for creating energy models, running simulations, analyzing results, and providing in-depth technical expertise on energy systems. It extends the BaseAgent class and adds specific functionality for parameter verification and energy model operations.

Key Responsibilities

- Verify parameters for energy modeling functions
- Create and manage energy models for different locations
- Process comprehensive energy model requests
- Analyze model results and extract insights
- Store model data in the knowledge base
- Handle both synchronous and asynchronous task execution

Core Functionality

Class Definition

```
class Emil(BaseAgent):
    # Emil agent extends BaseAgent with energy modeling capabilities
```

Parameter Verification

Emil provides both synchronous and asynchronous methods for verifying function parameters:

```
@log_function_call
def verify_parameters(self, function_name: str, task_args: dict) -> dict:
    """
    Synchronous method to verify parameters for a given function.

    Parameters:
        function_name (str): The function to verify parameters for
        task_args (dict): The provided task arguments

    Returns:
        dict: Verification results with 'success' flag and 'missing' parameters
    """
    # If the function exists in the function map, check its parameters
    if function_name in self.function_map:
        func = self.function_map[function_name]

        # Get the function's parameter names (excluding self/kb)
```

```
        sig = inspect.signature(func)
        required_params = [
            p.name for p in sig.parameters.values()
            if p.default == inspect.Parameter.empty and p.name not in ['self', 'kb']
        ]

        # Check which required parameters are missing
        missing_params = [
            param for param in required_params
            if param not in task_args
        ]

        if missing_params:
            return {
                "success": False,
                "message": f"Missing required parameters: {',
'.join(missing_params)}",
                "missing": missing_params
            }

        return {"success": True}

    # If function not found, return an error
    return {
        "success": False,
        "message": f"Function {function_name} not found in function map",
        "missing": []
    }
```

The asynchronous version includes special handling for certain function types:

```
@log_function_call
async def verify_parameters_async(self, function_name: str, task_args: dict) -> dict:
    """
    Asynchronous method to verify parameters for a given function.
    """
    # Special handling for process_emil_request
    if function_name == 'process_emil_request':
        # If a prompt is provided, consider it a valid call
        if task_args.get('prompt'):
            return {
                "success": True,
                "missing": [],
                "message": "Prompt provided for Emil request"
            }

    # Special handling for analyze_results
    if function_name == 'analyze_results':
        # For analysis tasks, we don't need explicit parameters
        # All data should be retrieved from the knowledge base
        return {
            "success": True,
```

```
            "missing": [],
            "message": "Analysis tasks don't require explicit parameters"
        }

    # Standard parameter verification follows...
```

Task Handling

Emil's asynchronous task handling implementation with specialized processing for different task types:

```
@log_function_call
async def handle_task_async(self, task: Task):
    """
    Asynchronous version of handle_task for Emil agent.
    Enhanced to better handle analysis tasks and log interactions.
    """
    print(f"Emil handling task asynchronously: {task.name}")

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

    # Special handling for analyze_results function
    if task.function_name == "analyze_results":
        # Retrieve model information from knowledge base
        model_file = self.kb.get_item("latest_model_file")
        model_details = self.kb.get_item("latest_model_details")

        # Call the analyze_results function directly with parameters from KB
        try:
            analyze_func = self.function_map.get("analyze_results")
            if not analyze_func:
                error_message = "analyze_results function not found in Emil's function
map"
                print(error_message)

                # Log the error
                self.kb.log_interaction(f"Task: {task.name}", error_message,
agent="Emil", function="analyze_results")

                task.result = error_message
                return error_message

            result = await asyncio.to_thread(
                analyze_func,
                self.kb,
                prompt=task.args.get("prompt", ""),
                analysis_type=task.args.get("analysis_type", "basic"),
                model_file=model_file,
                model_details=model_details
            )
```

```
            # Store results in knowledge base
            task.result = result
            await self.kb.set_item_async("latest_analysis_results", result,
category="analyses")

            # Store in session history
            # ... (additional code for session history management)

            return result
        except Exception as e:
            error_message = f"Error analyzing results: {str(e)}"
            print(error_message)

            # Log the error
            self.kb.log_interaction(f"Task: {task.name}", error_message, agent="Emil",
function="analyze_results")

            task.result = error_message
            return error_message

    # For other function types, use the regular logic
    if task.function_name and task.function_name in self.function_map:
        func = self.function_map[task.function_name]

        # Verify parameters using async method
        verification = await self.verify_parameters_async(task.function_name,
task.args)

        if not verification["success"]:
            # Handle missing parameters
            error_message = f"Error: {verification['message']}"
            print(error_message)

            # Log the parameter validation error
            self.kb.log_interaction(f"Task: {task.name}", error_message, agent="Emil",
function=task.function_name)

            # Store the error in the knowledge base
            await self.kb.set_item_async("emil_error", error_message)

            # Store error in the task result
            task.result = error_message

            # Return the error message
            return error_message

        # All parameters are present, so call the function
        task_args = task.args.copy()
        try:
            # Run the function in a thread pool since it's synchronous
            result = await asyncio.to_thread(func, self.kb, **task_args)
```

```
            # Store the result
            task.result = result

            # Handle specific function types with categorization
            if task.function_name == "process_emil_request":
                # Store with energy_models category
                await self.kb.set_item_async("emil_result", result,
category="energy_models")

                # Record in the session
                # ... (additional code for session management)

                # Also store the latest model details in standard keys for other
agents to use
                if isinstance(result, dict):
                    if 'file' in result:
                        await self.kb.set_item_async("latest_model_file",
result['file'])

                    # Store the entire result dict for comprehensive access
                    await self.kb.set_item_async("latest_model_details", result,
category="models")

                    # Store specific model attributes for easy access
                    for key in ['location', 'generation', 'generation_type',
'energy_carrier']:
                        if key in result:
                            await self.kb.set_item_async(f"latest_model_{key}",
result[key])
            else:
                # Generic result handling for other function types
                # ... (additional code for generic result handling)

            # Log successful completion
            self.kb.log_interaction(f"Task: {task.name}", "Task completed
successfully",
                                    agent="Emil", function=task.function_name)

            return result
        except Exception as e:
            error_message = f"Error executing {task.function_name}: {str(e)}"
            print(error_message)

            # Log the error
            self.kb.log_interaction(f"Task: {task.name}", error_message, agent="Emil",
function=task.function_name)

            task.result = error_message
            return error_message
    else:
        message = f"Emil has no function for task: {task.name}"
        print(message)
```

```
        # Log the error
        self.kb.log_interaction(f"Task: {task.name}", message, agent="Emil",
function=task.function_name)

        task.result = message
        return message
```

Key Features

1. **Parameter Verification**: Robust validation of required parameters before function execution
2. **Special Case Handling**: Custom logic for specific task types like analysis and model requests
3. **Knowledge Base Integration**: Stores model files, details, and analysis results for other agents
4. **Session Management**: Keeps track of model creation and analysis in the current session
5. **Error Handling**: Comprehensive error handling with detailed logging
6. **Asynchronous Processing**: Uses asyncio for non-blocking task execution

Integration

- Communicates with the knowledge base to store model details
- Works with the function registry to access energy modeling functions
- Coordinates with Lola for report generation based on model results
- Uses asyncio to handle tasks asynchronously

Workflow

1. Receives a task from Nova related to energy modeling
2. Verifies that all required parameters are present
3. Executes the appropriate function (model creation, analysis, etc.)
4. Stores results in the knowledge base with appropriate categorization
5. Updates session history with the completed task
6. Returns results to the task manager

Implementation Notes

- Uses thread pools to run synchronous functions in asynchronous contexts
- Implements special handling for different function types
- Maintains standardized keys in the knowledge base for cross-agent access
- Provides detailed error messages and logging