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

File Location

`src/agents/lola.py`

Overview

The `lola.py` file implements the Lola agent, which specializes in copywriting, proofreading, and report generation. Lola extends the BaseAgent class to provide natural language capabilities, creating high-quality written content based on data from other agents. It serves as the system's communication expert, transforming technical information into well-structured reports and other written content.

Key Responsibilities

- Generate reports based on energy model analysis
- Perform proofreading and copywriting tasks
- Create different report styles (executive summaries, technical reports, etc.)
- Verify parameters for writing-related functions
- Handle task execution both synchronously and asynchronously
- Store written content in the knowledge base

Core Functionality

Class Definition

```
class Lola(BaseAgent):
    # Lola agent extends BaseAgent with copywriting and report generation capabilities
```

Result Analysis

Lola can analyze energy model results, complementing Emil's capabilities:

```
@log_function_call
def analyze_results(self, kb, prompt=None, analysis_type="basic", file_path=None,
model_details=None, **kwargs):
    """
    Analyzes energy model results.

    Parameters:
        kb (KnowledgeBase): Knowledge base
        prompt (str): The original prompt
        analysis_type (str): Type of analysis to perform
        file_path (str): Path to the model file to analyze
        model_details (dict): Details about the model

    Returns:
        dict: Analysis results
    """
    # If file_path wasn't provided, try to get from KB
    if file_path is None:
```

```
        file_path = kb.get_item("latest_model_file")

    if model_details is None:
        model_details = kb.get_item("latest_model_details")

    # Analyze the file...
    # Store results in KB for later report writing
    analysis_results = {"key_findings": [...], "recommendations": [...]}
    kb.set_item("latest_analysis_results", analysis_results)

    return analysis_results
```

Parameter Verification

Asynchronous parameter verification with special handling for report writing:

```
@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.

    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
    """
    # Special handling for write_report
    if function_name == 'write_report':
        # Report writing doesn't need explicit parameters
        # All necessary data should be in the knowledge base
        return {
            "success": True,
            "missing": [],
            "message": "Report tasks don't require explicit parameters"
        }

    # Standard parameter verification for other functions...
```

Task Handling

Asynchronous task handling with special focus on report writing:

```
@log_function_call
async def handle_task_async(self, task: Task):
    """
    Asynchronous version of handle_task for Lola agent.
    Enhanced with better logging and categorized storage.
    """
    print(f"Lola handling task asynchronously: {task.name}")

    # Log the start of task execution
```

```
    self.kb.log_interaction(f"Task: {task.name}", "Starting execution", agent="Lola",
function=task.function_name)

    # Special handling for report writing function
    if task.function_name == "write_report":
        # Retrieve model and analysis information from knowledge base
        model_file = self.kb.get_item("latest_model_file")
        model_details = self.kb.get_item("latest_model_details")
        analysis_results = self.kb.get_item("latest_analysis_results")

        try:
            # CRITICAL FIX: Import the write_report function directly
            # to ensure we're using the correct function signature
            from core.functions_registery import write_report as global_write_report

            # Call the function explicitly with the kb parameter first
            result = await asyncio.to_thread(
                global_write_report,  # Use the explicitly imported function
                self.kb,  # First positional parameter is kb
                style=task.args.get("style", "executive_summary"),
                prompt=task.args.get("prompt", ""),
                model_file=model_file,
                model_details=model_details,
                analysis_results=analysis_results
            )

            # Store the result
            task.result = result

            # Store in knowledge base with categorization
            await self.kb.set_item_async("latest_report", result, category="reports")
            await self.kb.set_item_async("final_report", result)

            # Record in the session history
            current_session = self.kb.get_item("current_session")
            if current_session:
                session_data = self.kb.get_item(f"session_{current_session}")
                if session_data:
                    if "reports_generated" not in session_data:
                        session_data["reports_generated"] = []

                    report_info = {
                        "timestamp": datetime.datetime.now().isoformat(),
                        "style": task.args.get("style", "executive_summary"),
                        "prompt": task.args.get("prompt", "")
                    }
                    session_data["reports_generated"].append(report_info)
                    self.kb.set_item(f"session_{current_session}", session_data,
category="sessions")

            # Log successful completion
            self.kb.log_interaction(f"Task: {task.name}", "Report generated
```

```
successfully",
                                agent="Lola", function="write_report")

            return result

        except Exception as e:
            error_message = f"Error writing report: {str(e)}"
            print(error_message)

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

            task.result = error_message
            return error_message

    # Standard task processing for non-report tasks...
```

Report Generation

Lola implements several report generation styles:

```
@log_function_call
def generate_executive_summary(self, prompt, model_details, analysis_results):
    """
    Generates an executive summary report.

    Parameters:
        prompt (str): Original user prompt
        model_details (dict): Details about the model
        analysis_results (dict): Analysis results

    Returns:
        str: The executive summary report
    """
    # Extract relevant information
    location = model_details.get('location', 'the specified location')
    generation = model_details.get('generation', 'energy')
    carrier = model_details.get('energy_carrier', 'electricity')
    key_findings = analysis_results.get('key_findings', ['No specific findings
available'])

    # Generate the report
    report = f"""
# Executive Summary: {generation.capitalize()} Energy Model for
{location.capitalize()}

## Overview
This report summarizes the energy model created for {location}, focusing on
{generation} generation with {carrier} as the primary energy carrier.

## Key Findings
"""
```

```
    # Add key findings as bullet points
    for finding in key_findings:
        report += f"- {finding}\n"

    # Add summary and recommendations
    report += f"""
## Summary
{analysis_results.get('summary', 'A model has been successfully created according to specifications.')}

## Recommendations
- Consider performing detailed simulations with this model
- Enhance the model with additional data sources
- Compare results with historical data for validation

Report generated based on user request: "{prompt}"
"""

    return report
```

Other report styles include technical reports and presentation-style reports.

Key Features

1. **Report Generation**: Creates different styles of reports based on energy model data
2. **Content Formatting**: Structures written content appropriately for different audiences
3. **Knowledge Base Integration**: Retrieves model details and analysis results from the knowledge base
4. **Session Tracking**: Records generated reports in the session history
5. **Parameter Verification**: Special handling for report writing parameters
6. **Error Handling**: Comprehensive error handling with detailed logging

Integration

- Communicates with the knowledge base to retrieve model and analysis data
- Coordinates with Emil for accessing energy model results
- Uses the function registry for report generation functions
- Records all generated content in the knowledge base
- Links reports to the specific model and analysis they're based on

Workflow

1. Receives a task from Nova related to report generation
2. Retrieves necessary data from the knowledge base:
    - Model file and details from Emil
    - Analysis results from Emil
3. Determines the appropriate report style
4. Generates the report with proper formatting
5. Stores the report in the knowledge base
6. Updates the session history

7. Returns the formatted report to the task manager

Implementation Notes

- Uses thread pools to run synchronous functions in asynchronous contexts
- Implements specialized report formats for different audiences
- Directly imports and uses the global write_report function to avoid reference issues
- Provides consistent report formatting across different report types
- Maintains clear separation between technical and executive content