Nova AI System: Code Improvements Summary

This document summarizes the key code improvements made to fix the model → analyze → report pipeline. Each section includes the file location, function name, code, and a brief explanation of the fix.

1. Parameter Extraction (`functions_registery.py`)

python

```python
@log_function_call
def extract_model_parameters(prompt):
    """Extract energy modeling parameters from the prompt using keyword matching."""
    print("Extracting model parameters from prompt...")
    prompt_lower = prompt.lower()
    params = {"locations": [], "generation_types": [], "energy_carriers": [], "model_type": "si

    # Extract locations - FIX: Better pattern matching for locations
    found_locations = []
    for loc in LOCATIONS:
        # Look for "in [location]" or "for [location]" patterns
        if f" in {loc.lower()}" in prompt_lower or f" for {loc.lower()}" in prompt_lower:
            found_locations.append(loc)
        # Also check for exact match
        elif loc.lower() in prompt_lower:
            found_locations.append(loc)

    # Make sure to prioritize specific location mentions
    params["locations"] = list(set(found_locations))

    # Extract generation types - FIX: Better pattern matching
    found_gen_types = []
    for gen in GENERATION_TYPES.keys():
        # Look for "[gen] power" or similar patterns
        if f"{gen} power" in prompt_lower or f"{gen} generation" in prompt_lower:
            found_gen_types.append(gen)
        # Also check for exact match
        elif gen in prompt_lower:
            found_gen_types.append(gen)

    params["generation_types"] = found_gen_types

    # Set defaults if nothing found
    if not params["locations"]:
        params["locations"] = ["Unknown"]
    if not params["generation_types"]:
        params["generation_types"] = ["solar"]  # Default to solar

    return params
```

Improvement Summary: Enhanced parameter extraction to better identify location ("in Spain") and generation type ("solar power") patterns. Added more robust pattern matching and default values to ensure essential parameters are always present.

2. Model Creation (`functions_registery.py`)

python

```python
@log_function_call
def create_simple_xml(location_name, gen_type_name, carrier_name, output_xml):
    """Create a simple XML model file when PLEXOS is not available."""
    try:
        import xml.etree.ElementTree as ET
        root = ET.Element("PLEXOSModel", version="7.4")
        root.set("xmlns", "https://custom-energy-model/non-plexos/v1")
        comment = ET.Comment("This is a placeholder XML file and is not a PLEXOS database")
        root.append(comment)

        # Add metadata
        metadata = ET.SubElement(root, "Metadata")
        timestamp = ET.SubElement(metadata, "Timestamp")
        timestamp.text = datetime.datetime.now().isoformat()

        modeltype = ET.SubElement(metadata, "ModelType")
        modeltype.text = gen_type_name

        location_elem = ET.SubElement(metadata, "Location")
        location_elem.text = location_name

        carrier_elem = ET.SubElement(metadata, "EnergyCarrier")
        carrier_elem.text = carrier_name

        # XML file creation code...

        # FIXED: Return the correct parameters in the right order
        return {
            "status": "success",
            "message": f"Created {gen_type_name} {carrier_name} model for {location_name}",
            "file": output_xml,
            "location": location_name,  # FIXED: This was incorrectly using gen_type_name
            "generation_type": gen_type_name,
            "energy_carrier": carrier_name
        }
    except Exception as e:
        print(f"❌ Error creating simple XML: {str(e)}")
        return {"status": "error", "message": f"Failed to create simple XML: {str(e)}"}
```

Improvement Summary: Fixed a critical parameter mix-up where the location and generation type were swapped in the returned dictionary. The location field was incorrectly being set to the generation type, causing confusion in downstream components.

3. Analysis Function (`functions_registery.py`)

python

```python
@log_function_call
def analyze_results(kb, prompt=None, analysis_type="basic", model_file=None, model_details=None
    """Analyzes energy model results."""
    print(f"Emil analyzing model with {analysis_type} analysis...")

    # If file_path wasn't provided, try to get from KB
    if model_file is None:
        model_file = kb.get_item("latest_model_file")

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

    # Extract model information (if available)
    location = model_details.get('location', 'Unknown location') if model_details else 'Unknowr

    # FIXED: Use generation_type explicitly and use fallback to 'generation' field
    generation = model_details.get('generation_type',
             model_details.get('generation', 'Unknown generation type')) if model_details else

    energy_carrier = model_details.get('energy_carrier', 'Unknown carrier') if model_details el

    # Added debugging
    print(f"Analysis using: location={location}, generation={generation}, energy_carrier={energ

    # Create analysis results with correct parameter values
    result = {
        "status": "success",
        "message": f"Completed {analysis_type} analysis of {generation} {energy_carrier} model
        "key_findings": [
            f"Model contains {len(regions)} region(s): {', '.join(regions) if regions else 'Nor
            f"Model contains {len(nodes)} node(s)",
            f"Primary generation type: {generation}",
            f"Primary energy carrier: {energy_carrier}"
        ],
        # More result fields...
    }
    # Rest of function...
```

Improvement Summary: Fixed parameter handling in the analysis function to correctly access the generation_type from model_details with proper fallbacks. Added debugging output to trace parameter values and ensure proper data flow between components.

4. Report Storage (`knowledge_base.py`)

python

```python
def store_report(self, report, prompt=None, model_details=None):
    """Store a report in both the session history and current session data."""
    # Get current session
    current_session = self.get_item("current_session")
    if not current_session:
        current_session = self.create_session()

    # Get session data
    session_data = self.get_item(f"session_{current_session}")

    if session_data:
        # Create report entry
        report_entry = {
            "timestamp": datetime.datetime.now().isoformat(),
            "report": report,
            "prompt": prompt or "No prompt provided"
        }

        # Add to reports_generated list in session data
        if "reports_generated" not in session_data:
            session_data["reports_generated"] = []

        session_data["reports_generated"].append(report_entry)
        self.set_item(f"session_{current_session}", session_data, category="sessions")

        # Now update the main session history
        history = self.get_item("session_history") or {}

        # Ensure reports list exists
        if "reports" not in history:
            history["reports"] = []

        # Create history entry
        history_entry = {
            "session_id": session_data.get("id", current_session),
            "prompt": prompt or "No prompt provided",
            "result": report,
            "model_type": model_details.get("generation_type", "Unknown"),
            "location": model_details.get("location", "Unknown"),
            "timestamp": datetime.datetime.now().isoformat()
        }

        # Add to reports list in history
        history["reports"].append(history_entry)

        # Update history
```

```
        self.set_item("session_history", history)

        print(f"Report stored in session {current_session} and session history")
        print(f"Session now has {len(session_data['reports_generated'])} reports")
        print(f"History now has {len(history['reports'])} reports")

        return True

    return False
```

Improvement Summary: Added a dedicated method to store reports in both the session-specific data and the global session history. This ensures reports are properly tracked in the knowledge base at all levels, fixing the issue where reports weren't appearing in the session history.

5. Report Writing Integration (`functions_registery.py`)

python

```
@log_function_call
def write_report(kb: KnowledgeBase, prompt=None, style="executive_summary", model_file=None, mc
    """Writes a report based on model and analysis results."""
    print(f"Writing report in {style} style")

    # Get model information from KB if not provided
    if model_file is None:
        model_file = kb.get_item("latest_model_file")

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

    if analysis_results is None:
        analysis_results = kb.get_item("latest_analysis_results")

    # Generate the report...

    # Store the final report in the knowledge base
    kb.set_item("latest_report", report)
    kb.set_item("final_report", report)

    # FIXED: Use the new method to store reports in both places
    kb.store_report(report, prompt, model_details)

    return report
```

Improvement Summary: Modified the report writing function to use the new `store_report` method, ensuring reports are properly saved to both the session-specific data and the global session history.

6. Task Hierarchy Detection (`nova.py`)

python

```python
@log_function_call
async def create_task_list_from_prompt_async(self, prompt: str) -> List[Task]:
    """Parses the high-level user prompt into a list of tasks."""
    # Other code...

    # Check if we have a sequence of tasks (model → analyze → report)
    has_model_task = any("model" in intent_info["intent"].lower() for intent_info in multiple_i
    has_analyze_task = any(any(word in intent_info["intent"].lower() for word in ["analyze", "a
                        for intent_info in multiple_intents)
    has_report_task = any(any(word in intent_info["intent"].lower() for word in ["report", "wri
                       for intent_info in multiple_intents)

    # If we have all three kinds of tasks, build a hierarchical structure
    if has_model_task and (has_analyze_task or has_report_task):
        print("DETECTED: Sequential task chain for model → analyze → report")

        # First, find and process the model task
        model_intent = next((intent_info for intent_info in multiple_intents
                         if "model" in intent_info["intent"].lower()), None)

        if model_intent:
            # Categorize and create the model task
            model_intent_text = model_intent["intent"]
            model_category = await open_ai_categorisation_async(model_intent_text, csv_path)
            model_task = await self.create_task_for_category(model_intent_text, model_category)

            # Find the analysis task if it exists
            analysis_intent = next((intent_info for intent_info in multiple_intents
                                if any(word in intent_info["intent"].lower()
                                      for word in ["analyze", "analysis", "assess"])), None)

            # Create and add the analysis task as a subtask
            if analysis_intent:
                analysis_intent_text = analysis_intent["intent"]
                analysis_category = await open_ai_categorisation_async(analysis_intent_text, cs
                analysis_task = await self.create_task_for_category(analysis_intent_text, analy

                # Force the analysis task to be assigned to Emil
                analysis_task.agent = "Emil"
                analysis_task.function_name = "analyze_results"

                # Add analysis task as subtask to model task
                model_task.sub_tasks.append(analysis_task)

                # Find the report task if it exists
                report_intent = next((intent_info for intent_info in multiple_intents
```

```
                                    if any(word in intent_info["intent"].lower()
                                           for word in ["report", "write", "document"])), None)

                # Create and add the report task as a subtask of analysis
                if report_intent:
                    report_intent_text = report_intent["intent"]
                    report_category = await open_ai_categorisation_async(report_intent_text, cs
                    report_task = await self.create_task_for_category(report_intent_text, repor

                    # Force the report task to be assigned to Lola
                    report_task.agent = "Lola"
                    report_task.function_name = "write_report"

                    # Add report task as subtask to analysis task
                    analysis_task.sub_tasks.append(report_task)

            # Return the single model task with its hierarchical subtasks
            return [model_task]

    # For non-sequential tasks or if the sequential structure couldn't be created,
    # process each intent separately...
```

Improvement Summary: Enhanced the task creation logic to detect sequential task chains (model → analyze → report) and build a proper hierarchical task structure. This ensures tasks are executed in the correct order with proper data flow between steps.

7. Subtask Processing (`main.py`)

python

```python
async def process_subtasks(task, agents, results):
    """Process all subtasks of a task recursively and in sequence."""
    print(f"Processing {len(task.sub_tasks)} subtasks for {task.name}")

    for idx, subtask in enumerate(task.sub_tasks, 1):
        print(f"Subtask {idx}/{len(task.sub_tasks)}: {subtask.name}")
        print(f"Delegating to {subtask.agent}")

        agent = agents.get(subtask.agent)
        if agent:
            result = await agent.handle_task_async(subtask)
            results.append(result)

            # Process any sub-subtasks recursively
            if subtask.sub_tasks:
                await process_subtasks(subtask, agents, results)
        else:
            error = f"Agent {subtask.agent} not found for subtask {subtask.name}"
            print(f"ERROR: {error}")
            results.append(error)
```

Improvement Summary: Added a recursive function to process subtasks in sequence, ensuring hierarchical tasks are executed in the correct order. This maintains proper task dependencies and ensures data flows correctly between model creation, analysis, and report generation.

8. Task Processing in Main (`main.py`)

python

```python
async def process_prompt_tasks(prompt_idx, display_prompt, task_list, agents, kb):
    """Process all tasks for a single prompt sequentially."""
    print(f"[Prompt {prompt_idx + 1}] Processing: '{display_prompt}'")

    # Other code...

    # Process each task in sequence
    for i, task in enumerate(task_list, 1):
        print(f"[Prompt {prompt_idx + 1}] Task {i}/{len(task_list)}: {task.name}")
        print(f"[Prompt {prompt_idx + 1}] Delegating to {task.agent}")

        # Check if this task has subtasks
        if task.sub_tasks:
            print(f"[Prompt {prompt_idx + 1}] Task has {len(task.sub_tasks)} subtasks - process

            # First execute the main task
            agent = agents.get(task.agent)
            if agent:
                result = await agent.handle_task_async(task)
                results.append(result)

                # After main task completes, process subtasks
                await process_subtasks(task, agents, results)
        else:
            # Regular task processing...
```

Improvement Summary: Enhanced the main task processing function to check for subtasks and process hierarchical tasks correctly. This ensures the proper execution sequence for complex workflows like the model → analyze → report chain.

Detailed Flow Diagram

1. **User Input → Intent Detection**
   - Prompt: "Build an energy model, analyze it, write a report"
   - Nova detects multiple intents and their relationships
2. **Task Hierarchy Creation**
   - Model task (Parent)
     - Analysis task (Child)
       - Report task (Grandchild)
3. **Execution Flow**
   - Emil builds the model and saves details to KB

- Emil analyzes the model using data from KB
- Lola writes the report using both model and analysis data from KB

4. **Knowledge Base Storage**
   - Model details stored at each step
   - Session history updated with completed tasks
   - Reports properly categorized and stored

Results

✅ **Parameter Handling**: Countries and generation types correctly identified

✅ **Sequential Tasks**: Tasks properly execute in order with data flow

✅ **Report Generation**: Reports correctly generated and stored

✅ **Session History**: All information properly maintained

From the debug logs:

```
DEBUG: STORED REPORTS:
  1. 'Build an electricity energy model for solar power in UK...', Model: solar for Uk
(Session session_1746525145)
  2. 'Build an electricity energy model for solar power in UK...', Model: solar for Uk
(Session session_1746525278)
```

Next Steps

- Further improve parameter extraction for more complex requests
- Add validation checks to ensure model quality
- Enhance error handling across the task chain
- Develop more comprehensive reporting options
- Consider implementing a proper database backend for the knowledge base