File Breakdown: `src/core/functions_registery.py`

File Location

`src/core/functions_registery.py`

Overview

The `functions_registery.py` file defines the core functions that can be called by AI agents in the system. It implements energy modeling, analysis, reporting, and utility functions that form the functional backbone of the framework. This module acts as a registry for all available functions, mapping function names to actual Python implementations, and provides a consistent interface for agents to access functionality.

Key Responsibilities

- Define core functions for energy modeling and analysis
- Create, run, and analyze energy models
- Generate reports based on model results
- Extract parameters from user prompts
- Create different types of energy models (single location, multi-location, etc.)
- Register functions to make them available to agents
- Map function names to actual implementations

Core Functionality

Function Registration

A function map loader to register and retrieve functions:

```
# ----------------------------------------------------------------------
# Initialize Function Map Loader
# ----------------------------------------------------------------------

# Create a function map loader instance
function_loader = FunctionMapLoader()

# Register all available functions
function_loader.register_functions({
    "build_plexos_model": build_plexos_model,
    "run_plexos_model": run_plexos_model,
    "analyze_results": analyze_results,
    "write_report": write_report,
    "generate_python_script": generate_python_script,
    "extract_model_parameters": extract_model_parameters,
    "create_single_location_model": create_single_location_model,
    "create_simple_xml": create_simple_xml,
    "create_multi_location_model": create_multi_location_model,
    "create_simple_multi_location_xml": create_simple_multi_location_xml,
    "create_comprehensive_model": create_comprehensive_model,
    "create_simple_comprehensive_xml": create_simple_comprehensive_xml,
    "process_emil_request": process_emil_request
})

# Load function maps from CSV files
NOVA_FUNCTIONS = function_loader.load_function_map("Nova")
EMIL_FUNCTIONS = function_loader.load_function_map("Emil")
IVAN_FUNCTIONS = function_loader.load_function_map("Ivan")
LOLA_FUNCTIONS = function_loader.load_function_map("Lola")
```

Energy Model Creation

Functions for creating energy models:

```
@log_function_call
def create_single_location_model(kb, location, generation,
energy_carrier="electricity"):
    """
    Create an energy model for a specific location.
    Fixed to correctly handle parameter ordering.

    Parameters:
      - kb (KnowledgeBase): The knowledge base
      - location (str): The location for the model.
      - generation (str): The generation type.
      - energy_carrier (str): Energy carrier.

    Returns:
      - dict: Information about the created model.
    """
    try:
        # Ensure proper capitalization and type handling
        location_cap = location.capitalize()
        generation_cap = generation.lower()  # Keep generation type lowercase
        carrier_cap = energy_carrier.capitalize()

        print(f"Creating model for {generation_cap} {carrier_cap} in {location_cap}")

        # Determine paths
        script_dir = os.path.dirname(os.path.abspath(__file__))
        project_dir = os.path.abspath(os.path.join(script_dir, ".."))  # Go up one
level from core
        models_dir = os.path.join(script_dir, "models")
        os.makedirs(models_dir, exist_ok=True)

        # Create a timestamp for the filename
        timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
        model_name = f"{location_cap}_{generation_cap}_{carrier_cap}_{timestamp}"
        safe_filename = ''.join(c if c.isalnum() or c in ['-', '_'] else '_' for c in
model_name)
        output_xml = os.path.join(models_dir, f"{safe_filename}.xml")
        # Check if PLEXOS is available and if the basefile exists
        basefile_path = os.path.join(project_dir, "basefile.xml")
        if os.path.exists(basefile_path) and PLEXOS_AVAILABLE:
            # PLEXOS processing code would go here
            pass
        else:
            print("PLEXOS not available or basefile not found, creating simple
XML...")
            return create_simple_xml(location_cap, generation_cap, carrier_cap,
output_xml)
    except Exception as e:
        print(f"□ Error creating model: {str(e)}")
        return {"status": "error", "message": f"Failed to create model: {str(e)}"}
```

Function for creating a simple XML model when PLEXOS is not available:

```
@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.

    Parameters:
      - location_name (str): Location name.
      - gen_type_name (str): Generation type.
      - carrier_name (str): Energy carrier.
      - output_xml (str): Output file path.

    Returns:
      - dict: Result information.
    """
    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
        ET.SubElement(metadata, "Format").text = "Custom XML (Not PLEXOS)"

        # Add entities
        entities = ET.SubElement(root, "Entities")

        # Add region
        regions = ET.SubElement(entities, "Regions")
        region = ET.SubElement(regions, "Region", id="1", name=location_name)

        # Add node
        nodes = ET.SubElement(entities, "Nodes")
        node = ET.SubElement(nodes, "Node", id="1",
                            name=f"{location_name}_{gen_type_name}_{carrier_name}",
                            type=gen_type_name,
                            carrier=carrier_name,
                            region=location_name)

        # Write the XML to file
        with open(output_xml, 'w', encoding='utf-8') as f:
            f.write('<?xml version="1.0" encoding="utf-8"?>\n')
            f.write('<!-- NOT A PLEXOS DATABASE. -->\n')
            xml_str = ET.tostring(root, encoding='utf-8').decode('utf-8')
            f.write(xml_str)

        print(f" Created simple XML: {output_xml}")
        return {
            "status": "success",
            "message": f"Created {gen_type_name} {carrier_name} model for
{location_name}",
            "file": output_xml,
            "location": location_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)}"}
```

Parameter Extraction

Function for extracting energy model parameters from user prompts:

```
@log_function_call
def extract_model_parameters(prompt):
    """
    Extract energy modeling parameters from the prompt using keyword matching.

    Parameters:
        - prompt (str): The user's prompt.

    Returns:
        - dict: Structured parameters including locations, generation_types,
energy_carriers, and model_type.
    """
    print("Extracting model parameters from prompt...")
    prompt_lower = prompt.lower()
    params = {"locations": [], "generation_types": [], "energy_carriers": [],
"model_type": "single"}

    # Extract locations
    found_locations = [loc.lower() for loc in LOCATIONS if loc.lower() in
prompt_lower]
    params["locations"] = list(set(found_locations))

    # Extract generation types from the keys in GENERATION_TYPES
    found_gen_types = [gen for gen in GENERATION_TYPES.keys() if gen in prompt_lower]
    params["generation_types"] = found_gen_types

    # Extract energy carriers (simplified example)
    carriers = ["electricity", "hydrogen", "methane"]
    found_carriers = [carrier for carrier in carriers if carrier in prompt_lower]
    params["energy_carriers"] = found_carriers

    if any(word in prompt_lower for word in ["multiple", "several", "separate"]):
        params["model_type"] = "multiple"

    print("Extracted parameters:", params, "\n")
    return params
```

Analysis and Reporting

Function for analyzing model results:

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

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

    Returns:
        dict: Analysis 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")
```

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

    # Check if we have a model to analyze
    if not model_file or not os.path.exists(model_file):
        print(f"No model file found to analyze.")
        result = {
            "status": "error",
            "message": "No model file found for analysis",
            "key_findings": [
                "Analysis could not proceed due to missing model file."
            ]
        }
        kb.set_item("latest_analysis_results", result)
        return result

    try:
        # Extract model information (if available)
        location = model_details.get('location', 'Unknown location') if model_details
else 'Unknown location'
        generation = model_details.get('generation', 'Unknown generation type') if
model_details else 'Unknown generation type'
        energy_carrier = model_details.get('energy_carrier', 'Unknown carrier') if
model_details else 'Unknown carrier'

        print(f"Analyzing model for {location} with {generation} generation and
{energy_carrier} carrier")

        # Parse XML to extract model data
        try:
            import xml.etree.ElementTree as ET
            tree = ET.parse(model_file)
            root = tree.getroot()

            # Extract regions, nodes, etc.
            regions = []
            nodes = []

            # Look for regions
            region_elements = root.findall(".//Region") or root.findall(".//*
[@name='Region']")
            for region in region_elements:
                regions.append(region.get('name'))

            # Look for nodes
            node_elements = root.findall(".//Node") or root.findall(".//*
[@name='Node']")
            for node in node_elements:
                nodes.append({
                    'name': node.get('name'),
                    'type': node.get('type'),
                    'carrier': node.get('carrier'),
                    'region': node.get('region')
                })

            # Create analysis results based on extracted data
            result = {
                "status": "success",
                "message": f"Completed {analysis_type} analysis of {generation}
{energy_carrier} model for {location}",
                "key_findings": [
                    f"Model contains {len(regions)} region(s): {', '.join(regions) if
regions else 'None'}",
                    f"Model contains {len(nodes)} node(s)",
                    f"Primary generation type: {generation}",
                    f"Primary energy carrier: {energy_carrier}"
                ],
                "recommendations": [
                    f"Consider expanding {location} model with additional generation
types",
                    f"Compare results with historical data from {location}",
                    f"Run sensitivity analysis on key parameters for {generation}
generation"
                ],
                "summary": f"The {generation} {energy_carrier} model for {location}
has been successfully analyzed using {analysis_type} analysis approach.",
                "data": {
                    "regions": regions,
                    "nodes": nodes,
                    "generation_type": generation,
                    "energy_carrier": energy_carrier,
                    "location": location
                }
            }

            # Store the analysis results for use by reporting functions
            kb.set_item("latest_analysis_results", result)
            kb.set_item("final_report", f"Analysis completed for {location} model. Key
findings: {', '.join(result['key_findings'][:2])}")

            print(f"Analysis complete with {len(result['key_findings'])} findings.")
            return result

        except Exception as xml_error:
            print(f"Error parsing model file: {str(xml_error)}")
            # Provide basic analysis based on model_details
            result = {
                "status": "partial",
                "message": f"Partial analysis of {generation} {energy_carrier} model
for {location}",
                "key_findings": [
                    f"Model was created for {location}",
                    f"Model uses {generation} generation type",
                    f"Model uses {energy_carrier} as energy carrier"
                ],
                "recommendations": [
                    "Consider running a more detailed analysis",
                    "Validate model parameters against industry standards"
                ],
                "summary": f"Basic analysis of {generation} {energy_carrier} model for
{location}, without detailed structure extraction."
            }
            kb.set_item("latest_analysis_results", result)
            kb.set_item("final_report", f"Basic analysis completed for {location}
model.")
            return result

    except Exception as e:
        print(f"Error during analysis: {str(e)}")
        result = {
            "status": "error",
            "message": f"Error during analysis: {str(e)}",
            "key_findings": [
                "Analysis encountered errors."
            ]
        }
        kb.set_item("latest_analysis_results", result)
        return result
```

Function for writing reports based on model analysis:

```
@log_function_call
def write_report(kb: KnowledgeBase, prompt=None, style="executive_summary",
model_file=None, model_details=None, analysis_results=None):
    """
    Writes a report based on model and analysis results.

    Parameters:
        kb (KnowledgeBase): Knowledge base
        prompt (str): Original user prompt
        style (str): Report style ("executive_summary", "technical_report", etc.)
        model_file (str): Path to the model file
        model_details (dict): Details about the model
        analysis_results (dict): Results from previous analysis

    Returns:
        str: The generated report
    """
    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")

    # Check if we have the necessary information to write a report
    if not model_details:
        return "Error: No model details available for report generation."

    if not analysis_results:
        # Create a basic analysis result if none exists
        analysis_results = {
            "key_findings": [
                f"Model was created successfully for {model_details.get('location',
'unknown location')}"
            ],
            "recommendations": [
                "Consider running a detailed analysis on this model",
                "Validate with historical data"
            ],
            "summary": f"A {model_details.get('generation_type', 'energy')} model was
created for {model_details.get('location', 'unknown location')}."
        }

    # Extract key information
    location = model_details.get('location', 'Unknown location')
    generation = model_details.get('generation_type', 'Unknown generation type')
    energy_carrier = model_details.get('energy_carrier', 'Unknown carrier')

    # Determine report type based on style
    if style == "executive_summary":
        report = generate_executive_summary(prompt, model_details, analysis_results,
location, generation, energy_carrier)
    elif style == "technical_report":
        report = generate_technical_report(prompt, model_details, analysis_results,
location, generation, energy_carrier)
    elif style == "presentation_report":
        report = generate_presentation_report(prompt, model_details, analysis_results,
location, generation, energy_carrier)
    else:
        # Default to executive summary
        report = generate_executive_summary(prompt, model_details, analysis_results,
location, generation, energy_carrier)

    # Store the final report in the knowledge base
    kb.set_item("latest_report", report)
    kb.set_item("final_report", report)  # This is used as the final output

    return report
```

Process Emil Request

High-level function to process energy modeling requests:

```python
@log_function_call
def process_emil_request(kb: KnowledgeBase, prompt: str = None, **kwargs):
    """
    This function wraps Emil's energy model processing logic.
    Enhanced with better location parsing.

    Parameters:
        - kb (KnowledgeBase): The knowledge base to store results
        - prompt (str, optional): The user's original prompt
        - **kwargs: Additional flexible parameters for model creation

    Returns:
        - dict: Information about the created model
    """
    # Ensure prompt is set
    if prompt is None:
        prompt = kwargs.get('prompt', 'Create an energy model')

    print(f"Processing Emil request with prompt: {prompt}")

    # Extract initial parameters from the prompt
    params = extract_model_parameters(prompt)

    # Prepare task arguments with the prompt
    task_args = {"prompt": prompt}

    # Check and set locations - improved handling
    locations = kwargs.get('location') or params.get('locations', [])
    if locations:
        # Convert string to list if needed
        if isinstance(locations, str):
            # Split by commas if present
            if ',' in locations:
                locations = [loc.strip() for loc in locations.split(',')]
            else:
                locations = [locations.strip()]

        # Store as comma-separated string
        task_args['location'] = ','.join(locations)

    # Check and set generation types
    generation_types = kwargs.get('generation') or params.get('generation_types', [])
    if generation_types:
        # Convert to list if it's a string
        if isinstance(generation_types, str):
            if ',' in generation_types:
                generation_types = [gt.strip() for gt in generation_types.split(',')]
            else:
                generation_types = [generation_types.strip()]
        task_args['generation'] = ','.join(generation_types)

    # Check and set energy carriers
    energy_carriers = kwargs.get('energy_carrier') or params.get('energy_carriers', 
[])
    if energy_carriers:
        # Convert to list if it's a string
        if isinstance(energy_carriers, str):
            if ',' in energy_carriers:
                energy_carriers = [ec.strip() for ec in energy_carriers.split(',')]
            else:
                energy_carriers = [energy_carriers.strip()]
        task_args['energy_carrier'] = ','.join(energy_carriers)

    # Determine model type based on parameters
    model_type = 'single'
    if len(locations) > 1 or len(generation_types) > 1:
        model_type = 'multiple'

    # Choose appropriate model creation function
    if model_type == 'single':
        # Use first (or default) values if available
        location = locations[0] if locations else 'all'
        generation = generation_types[0] if generation_types else 'all'
        energy_carrier = energy_carriers[0] if energy_carriers else 'electricity'

        # Fix for the location parsing issue
        # Make sure the location is treated as a single entity
        result = create_single_location_model(location, generation, energy_carrier)
    elif model_type == 'multiple':
        # If multiple locations or generation types, use comprehensive model with 
fixed parsing
        result = create_comprehensive_model(
            locations or ['all'],
            generation_types or ['all'],
            energy_carriers or ['electricity']
        )
    else:
        # Fallback to a single location model with default values
        result = create_single_location_model('all', 'all', 'electricity')

    # Store the result in the knowledge base
    kb.set_item("emil_result", result)

    return result
```

Constants and Configuration

Constants and configuration values used in the file:

```
# ----------------------------------------------------------------------
# Global Constants
# ----------------------------------------------------------------------

GENERATION_TYPES = {
    "wind": ["Onshore Wind", "Onshore Wind Expansion", "Offshore Wind Radial"],
    "solar": ["Solar PV", "Solar PV Expansion", "Solar Thermal Expansion",
              "Rooftop Solar Tertiary", "Rooftop Tertiary Solar Expansion"],
    "hydro": ["RoR and Pondage", "Pump Storage - closed loop"],
    "thermal": ["Hard coal", "Heavy oil"],
    "bio": ["Bio Fuels"],
    "other": ["Other RES", "DSR Industry"]
}

LOCATIONS = [
    "Spain", "UK", "France", "Germany", "Italy",
    "Portugal", "Belgium", "Netherlands", "Greece",
    "Croatia", "Sweden", "Norway", "Denmark",
    "Finland", "Ireland", "Switzerland", "Austria"
]
```

Key Features

1. **Modular Functions**: Organizes energy modeling capabilities into discrete functions
2. **Parameter Extraction**: Intelligently extracts model parameters from natural language prompts
3. **Model Creation**: Creates different types of energy models (single location, multi-location, comprehensive)
4. **Analysis**: Evaluates models to extract key insights and recommendations
5. **Report Generation**: Creates different styles of reports based on analysis results
6. **Function Registry**: Maps function names to implementations for agent access
7. **Error Handling**: Provides robust error handling and fallback mechanisms

Integration

- Called by Emil agent to process energy modeling tasks
- Used by Lola agent for report generation
- Integrated with the knowledge base for storing results and context
- Connected to the task management system through the function registry
- Provides the technical implementation behind agent capabilities

Workflow

1. Nova agent identifies an energy modeling intent in a user prompt
2. Parameters are extracted from the prompt using `extract_model_parameters`
3. The appropriate function is called via `process_emil_request`
4. An energy model is created using the appropriate model creation function
5. Model results are stored in the knowledge base
6. If requested, analysis is performed using `analyze_results`
7. If requested, reports are generated using `write_report`
8. Results are returned to the user and stored for future reference

Implementation Notes

- Checks for PLEXOS availability and falls back to simple XML if not present
- Uses XML-based model representation for compatibility

- Implements parameter handling for different formats (strings, lists, etc.)
- Provides multiple report styles for different audiences
- Organizes functions into logical groups based on functionality
- Uses logging decorators for tracking function calls
- Handles various edge cases and error conditions