AI Agent Coordinator System - Complete Technical Documentation

Overview

The AI Agent Coordinator is a multi-agent system for energy modeling that automatically handles complex workflows. Think of it like having a team of specialists: Nova (the manager), Emil (the engineer), Lola (the writer), and Ivan (the generator). They work together to understand what you want, build energy models, and create reports.

Keywords: multi-agent system, energy modeling, PLEXOS, parameter collection, streamlit, CLI, force_cli, workflow automation

How the System Works: A Simple Example

Let's trace through what happens when you type: **"build a wind model for spain and write a report"**

Step 1: Nova Understands Your Request

Nova is like a smart project manager. It reads your request and figures out what needs to be done.

Keywords: intent detection, task creation, prompt parsing, Nova agent

File: `src/agents/nova.py` **, Function:** `create_task_list_from_prompt_async` **(lines 90-150)**

```
async def create_task_list_from_prompt_async(self, prompt: str) -> List[Task]:
    # Your input: "build a wind model for spain and write a report"

    # 1. Nova splits this into separate tasks
    # File: src/agents/nova.py, Function: identify_multiple_intents_async (lines 250-300)
    intents = await self.identify_multiple_intents_async(prompt)
    # Result: [
    #   {"intent": "build a wind model for spain"},
    #   {"intent": "write a report"}
    # ]

    # 2. Nova decides which agent should handle each task
    # File: src/agents/nova.py, Function: _create_task_with_category (lines 150-200)
    for intent in intents:
        if "model" in intent_text and "energy" related words:
            task.agent = "Emil"  # Emil handles energy modeling
            task.function_name = "process_emil_request"
        elif "report" in intent_text:
            task.agent = "Lola"  # Lola handles report writing
            task.function_name = "write_report"
```

Intent Classification Logic: File: `src/agents/nova.py` **, Function:** `_create_task_with_category` **(lines 180-210)**

```
 # Nova categorizes your request
model_keywords = ['model', 'build', 'create', 'generate', 'design', 'construct']
energy_keywords = ['energy', 'solar', 'wind', 'hydro', 'electricity', 'power']

# If your prompt contains both model AND energy words:
if (any(word in prompt for word in model_keywords) and
    any(word in prompt for word in energy_keywords)):
    # This goes to Emil (the energy modeling expert)
    agent = "Emil"
```

Step 2: Parameter Detection and Collection

This is where the system gets smart about missing information. Each agent knows what information it needs to do its job.

Keywords: parameter validation, missing parameters, Emil requirements, parameter extraction

How Emil Checks for Missing Parameters

File: `src/agents/emil.py` **, Function:** `verify_parameters_async` **(lines 35-50)**

```
async def verify_parameters_async(self, function_name: str, task_args: dict) -> dict:
    """Emil checks if he has everything needed to build a model"""

    if function_name == 'process_emil_request':
        # Emil needs these 3 things to build an energy model:
        missing = []

        if not task_args.get('location'):
            missing.append('location')  # Where should the model be for?

        if not task_args.get('generation'):
            missing.append('generation')  # What type of energy? (wind, solar, etc.)

        # energy_carrier defaults to 'electricity' if not specified

        if missing:
            return {
                "success": False,
                "missing": missing,
                "message": f"I need: {', '.join(missing)}"
            }

        return {"success": True, "message": "I have everything I need!"}
```

The Parameter Extraction Process

Before asking you for parameters, the system tries to extract them from your original request using AI.

Keywords: LLM parameter extraction, automatic parameter detection, location correction, generation type detection

File: `app.py` **, Function:** `extract_model_parameters_with_llm_correction` **(lines 140-184)**

```
async def extract_model_parameters_with_llm_correction(prompt):
    """Smart parameter extraction from your text"""

    # Your prompt: "build a wind model for spain"
    params = {
        "locations": [],
        "generation_types": [],
        "energy_carriers": []
    }

    # Step 1: Look for location names
    prompt_lower = prompt.lower()  # "build a wind model for spain"

    # File: app.py, Global constant LOCATIONS (lines 62-85)
    for location in LOCATIONS:  # ['Spain', 'France', 'Germany', ...]
        patterns = [
            f" for {location.lower()}",  # Matches "for spain"
            f" in {location.lower()}",   # Matches "in spain"
            f" {location.lower()} "      # Matches " spain "
        ]
        if any(pattern in prompt_lower for pattern in patterns):
            params["locations"].append(location)  # Found: ["Spain"]

    # Step 2: If location is misspelled, use AI to correct it
    if not params["locations"]:
        # File: utils/open_ai_utils.py, Function: run_open_ai_ns_async
        # AI corrects "spian" → "Spain", "craotia" → "Croatia"
        corrected = await run_open_ai_ns_async(prompt, correction_context)
        if corrected != "Unknown":
            params["locations"] = [corrected]

    # Step 3: Look for energy types
    generation_keywords = ['solar', 'wind', 'hydro', 'nuclear', 'thermal']
    for gen_type in generation_keywords:
        if gen_type in prompt_lower:  # Found "wind" in prompt
            params["generation_types"].append(gen_type)  # ["wind"]

    return params  # {"locations": ["Spain"], "generation_types": ["wind"], ...}
```

Step 3: When Parameters Are Missing - The UI Collection Process

If the system can't figure out what you want, it shows you a form to fill in the missing information.

Keywords: parameter collection logic, missing parameter detection, UI forms

The Parameter Collection Logic

File: `app.py`, Class: `StreamlitParameterCollector`, Function: `needs_parameters` (lines 185-220)

```
@staticmethod
def needs_parameters(task_args, function_name):
    """Check if Emil is missing required information"""

    if function_name != 'process_emil_request':
        return False, []  # Other agents might not need extra params

    missing = []

    # Check 1: Does Emil know what type of energy generation?
    has_generation = (
        task_args.get('generation') or           # Direct: "generation": "wind"
        task_args.get('generation_type') or     # Alternative name
        task_args.get('generation_types')       # From extraction: ["wind"]
    )
    if not has_generation:
        missing.append('generation')

    # Check 2: Does Emil know the location?
    has_location = (
        task_args.get('location') and task_args.get('location') != 'Unknown' or
        (task_args.get('locations') and
         task_args.get('locations')[0] != 'Unknown')
    )
    if not has_location:
        missing.append('location')

    return len(missing) > 0, missing  # True if parameters missing
```

The Parameter Collection Form

File: `app.py`, Function: `show_parameter_form` (lines 285-350)

```
def show_parameter_form(missing_params, task_args):
    """Show a user-friendly form to collect missing information"""

    st.info("🤖 I need some additional information to complete your request:")

    collected_params = {}

    with st.form("parameter_collection_form"):
        st.markdown("### Please provide the following details:")

        # If generation type is missing
        if 'generation' in missing_params:
            st.markdown("**Generation Type** - What type of energy?")
            # File: app.py, Global constant GENERATION_TYPES (lines 50-60)
            generation_options = ['solar', 'wind', 'hydro', 'thermal', 'bio', 'nuclear']
            collected_params['generation'] = st.selectbox(
                "Select generation type:",
                options=generation_options,
                help="Choose the type of energy generation for your model"
            )

        # If location is missing
        if 'location' in missing_params:
            st.markdown("**Location** - Which country/region?")
            common_locations = ['Spain', 'France', 'Germany', 'Italy', 'UK']
            location_input = st.selectbox(
                "Select a location:",
                options=['Other...'] + common_locations
            )

            if location_input == 'Other...':
                collected_params['location'] = st.text_input(
                    "Enter location:",
                    placeholder="e.g., Denmark, Sweden, Poland"
                )
            else:
                collected_params['location'] = location_input

        # Submit button
        submitted = st.form_submit_button("🚀 Continue with these parameters")

        if submitted:
            # Store the collected parameters
            st.session_state.collected_parameters = collected_params
            st.session_state.parameters_ready = True
            st.rerun()  # Restart processing with new parameters
```

Parameter Collection Methods: force_cli True vs False

The system supports two different ways to collect missing parameters from users. The key difference is controlled by the `force_cli` parameter, which determines whether to use web forms (Streamlit) or command-line prompts.

Keywords: force_cli, parameter collection modes, Streamlit forms, CLI prompts, user input methods

Real-World Example Comparison

Let's examine the same prompt **"build a wind model for spain"** in both modes:

Streamlit Mode (force_cli=False) - Web Interface

Keywords: streamlit parameter collection, web forms, automatic extraction, no user interaction

When running with `streamlit run app.py`, the system uses `force_cli=False`:

```
 # Console output from Streamlit mode:
# File: app.py, Function: process_prompts_with_ui_params execution
 OpenAI categorization: Energy Model
 Task args before extraction: {'prompt': 'build a wind model for spain', 'full_prompt': 'build a wind model for spain'}
Extracting model parameters from prompt...
Extracted parameters: {'locations': ['Spain'], 'generation_types': ['wind'], 'energy_carriers': ['electricity'], 'model_type': 'sing
 Task args after LLM-enhanced extraction: {'prompt': 'build a wind model for spain', 'full_prompt': 'build a wind model for spain',
 Checking task_args: {'prompt': 'build a wind model for spain', 'full_prompt': 'build a wind model for spain', 'generation_types':
 Missing parameters: []
 Needs parameters: False, Missing: []
```

What happened:

1. System automatically extracted ALL parameters from the prompt
2. Location: "Spain" was found and correctly identified
3. Generation: "wind" was found and correctly identified
4. Energy carrier: "electricity" was set as default
5. No user interaction required - processing continues immediately
6. Model builds successfully without asking for additional input

CLI Mode (force_cli=True) - Command Line

Keywords: CLI parameter collection, command line prompts, manual user input, parameter prompting

When running with `python src/main.py`, the system uses `force_cli=True`:

```
 # Console output from CLI mode:
# File: src/main.py, Function: interactive_async_main execution
 OpenAI categorization: Energy Model
 Context handover: Nova → Emil
   Task: build a wind model for spain...

 Emil needs: ['generation']

 PARAMETER COLLECTION MODE:
 Auto-detection Mode (will use CLI)
Using CLI for parameter collection

===== Nova needs input for process_emil_request =====

Nova: I need the 'generation' for this task.
Description: The generation type (e.g., solar, wind, hydro, thermal, bio)
Examples: solar, wind, hydro, etc.
Please enter generation: wind    # ← User types "wind"

===== Parameters collected successfully =====
```

What happened:

1. System couldn't automatically extract the "generation" parameter
2. Location: "Spain" was probably extracted but not shown in this output
3. Generation: "wind" was NOT automatically detected
4. System prompted user to manually enter "generation"
5. User had to type "wind" manually
6. After manual input, processing continued successfully

The force_cli Parameter Implementation

Keywords: force_cli implementation, parameter collection dispatcher, mode selection

The Parameter Collection Dispatcher

File: `src/agents/parameter_collection.py`, Function: `get_missing_parameters_async` (lines 30-65)

```
 import asyncio
import importlib.util
from utils.function_logger import log_function_call

# Check if streamlit is available at import time
streamlit_available = importlib.util.find_spec("streamlit") is not None

# Import the appropriate modules based on availability
if streamlit_available:
    try:
        # File: src/agents/simplified_parameter_collection.py
        from .simplified_parameter_collection import get_missing_parameters_simple_async as st_get_params
    except ImportError:
        streamlit_available = False  # If import fails, fall back to CLI

# Always import CLI version as fallback
# File: src/agents/cli_parameter_collection.py
from .cli_parameter_collection import get_missing_parameters_cli_async

@log_function_call
async def get_missing_parameters_async(function_name: str, missing_params: list,
                                       initial_args: dict = None, force_cli: bool = True) -> dict:
    """
    This is the main dispatcher that chooses between Streamlit and CLI parameter collection.
    Keywords: dispatcher, parameter collection routing, force_cli logic
    """

    print("\n PARAMETER COLLECTION MODE:")

    # The key decision: force_cli determines the collection method
    if streamlit_available and not force_cli:
        print(" Using Streamlit for parameter collection")
        try:
            return await st_get_params(function_name, missing_params, initial_args)
        except Exception as e:
            print(f" Streamlit collection failed: {e}, falling back to CLI")
            return await get_missing_parameters_cli_async(function_name, missing_params, initial_args)
    else:
        print(" Using CLI for parameter collection")
        return await get_missing_parameters_cli_async(function_name, missing_params, initial_args)
```

How force_cli Works in Different Contexts

Keywords: force_cli contexts, streamlit app behavior, CLI app behavior, context-dependent defaults

Context 1: Streamlit Web App (force_cli=False by default)

File: `app.py` , Function: `process_prompts_with_ui_params` (lines 400-600)

```
# In the Streamlit app, force_cli is effectively False
# because we handle parameter collection directly in the UI

def process_prompts_with_ui_params(prompts_text: str):
    """Main processing in Streamlit app - Keywords: streamlit processing, web UI parameter collection"""

    for task in tasks:
        if task.agent == "Emil":
            # 1. Try automatic parameter extraction first
            # File: app.py, Function: extract_model_parameters_with_llm_correction
            original_params = loop.run_until_complete(
                extract_model_parameters_with_llm_correction(task.args.get('prompt', ''))
            )

            # 2. Add extracted parameters to task
            if original_params.get('generation_types'):
                task.args['generation'] = original_params['generation_types'][0]
            if original_params.get('locations'):
                task.args['location'] = original_params['locations'][0]

            # 3. Check if parameters are still missing
            # File: app.py, Class: StreamlitParameterCollector, Function: needs_parameters
            needs_params, missing_params = StreamlitParameterCollector.needs_parameters(
                task.args, task.function_name
            )

            if needs_params:
                # 4. Show web form (NOT CLI) - this is the Streamlit way
                # File: app.py, Function: show_parameter_form
                show_parameter_form(missing_params, task.args)  # Shows web form
                return []  # Wait for user to fill the form
```

Context 2: CLI Application (force_cli=True by default)

File: `src/main.py`, Function: `interactive_async_main` (lines 100-400)

```
# In the CLI version, agents call parameter collection with force_cli=True

async def interactive_async_main():
    """Main CLI application - Keywords: CLI processing, command line parameter collection"""

    # When Emil needs parameters in CLI mode
    for task in tasks:
        agent = agents.get(task.agent)
        result = await agent.handle_task_async(task)  # This triggers parameter collection
```

How Emil calls parameter collection in CLI mode: File: `src/agents/emil.py`, Function: `handle_task_async` (lines 60-85)

```
async def handle_task_async(self, task: Task):
    """Emil's main work function - Keywords: Emil parameter handling, CLI parameter requests"""

    # Validate parameters
    # File: src/agents/emil.py, Function: verify_parameters_async
    validation = await self.verify_parameters_async(task.function_name, task.args)

    # If parameters are missing
    while not validation["success"] and validation.get("missing"):
        print(f"❓ Emil needs: {validation['missing']}")

        # This call chooses CLI collection because force_cli=True in CLI context
        # File: src/agents/parameter_collection.py, Function: get_missing_parameters_async
        collected = await get_missing_parameters_async(
            task.function_name,
            validation["missing"],
            task.args,
            force_cli=True  # ← This forces CLI collection in the CLI app
        )

        task.args.update(collected)
        validation = await self.verify_parameters_async(task.function_name, task.args)
```

CLI Parameter Collection Implementation

Keywords: CLI parameter collection, command line prompts, user input, asyncio input

File: `src/agents/cli_parameter_collection.py` , **Function:** `get_missing_parameters_cli_async` **(lines 10-80)**

```
async def get_missing_parameters_cli_async(function_name: str, missing_params: list,
                                       initial_args: dict = None) -> dict:
    """
    Command-line parameter collection - asks user to type responses
    Keywords: CLI prompts, command line input, parameter descriptions
    """
    collected_args = initial_args.copy() if initial_args else {}

    print(f"\n===== Nova needs input for {function_name} =====")

    # Helpful descriptions for each parameter
    param_descriptions = {
        "location": "The geographic location for the energy model (e.g., UK, France, Spain, etc.)",
        "generation": "The generation type (e.g., solar, wind, hydro, thermal, bio)",
        "energy_carrier": "Energy carrier to model (e.g., electricity, hydrogen, methane)",
        "prompt": "Detailed prompt describing the task",
        "analysis_type": "Type of analysis: basic, detailed, or comprehensive"
    }

    param_examples = {
        "location": "Examples: UK, France, Germany, Spain, 'all'",
        "generation": "Examples: solar, wind, hydro, thermal, bio, nuclear",
        "energy_carrier": "Examples: electricity (default), hydrogen, methane"
    }

    # Ask user for each missing parameter
    for param in missing_params:
        description = param_descriptions.get(param, f"The {param} input required")
        examples = param_examples.get(param, "No examples available")

        print(f"\nNova: I need the '{param}' for this task.")
        print(f"Description: {description}")
        print(f"{examples}")

        # Get user input (this blocks until user types something)
        value = await asyncio.to_thread(
            input,
            f"Please enter {param}: "
        )

        collected_args[param] = value.strip()

    print(f"\n===== Parameters collected successfully =====")
    return collected_args
```

Streamlit Parameter Collection Implementation

Keywords: Streamlit parameter collection, web forms, session state, UI forms

File: `src/agents/simplified_parameter_collection.py` , **Function:** `get_missing_parameters_simple_async` **(lines 10-80)**

```
async def get_missing_parameters_simple_async(function_name: str, missing_params: list,
                                              initial_args: dict = None) -> dict:
    """
    Streamlit parameter collection - uses web forms and session state
    Keywords: streamlit forms, session state, web UI collection
    """
    collected_args = initial_args.copy() if initial_args else {}

    print(f"Nova needs input for {function_name}...")

    # Check if we have collected parameters in session state from previous form submission
    if ('collected_parameters' in st.session_state and
        st.session_state.collected_parameters):

        # Use the parameters that were collected from the web form
        for param in missing_params:
            if param in st.session_state.collected_parameters:
                collected_args[param] = st.session_state.collected_parameters[param]

        # Clear the collected parameters so they don't get reused
        st.session_state.collected_parameters = {}

        # If we got all the parameters we needed, return them
        if all(param in collected_args for param in missing_params):
            return collected_args

    # Store what parameters we need in session state for the UI to display
    st.session_state.pending_parameters = {
        'function': function_name,
        'missing': missing_params,
        'descriptions': {
            param: param_descriptions.get(param, f"The {param} input required")
            for param in missing_params
        },
        'examples': {
            param: param_examples.get(param, "No examples available")
            for param in missing_params
        },
        'initial_args': collected_args
    }

    # Return empty dict to signal that parameters are pending collection via UI
    # The Streamlit app will detect this and show the parameter form
    return {}
```

The Complete Parameter Flow

Keywords: parameter flow, extraction to collection, complete workflow

Here's how everything works together:

File: `app.py`, **Function:** `process_prompts_with_ui_params` **(lines 400-600)**

```
def process_prompts_with_ui_params(prompts_text: str):
    """Main processing function that handles parameter collection
    Keywords: main processing, parameter workflow, complete flow"""

    # 1. Nova creates tasks from your prompt
    # File: src/agents/nova.py, Function: create_task_list_from_prompt_async
    tasks = loop.run_until_complete(agents["Nova"].create_task_list_from_prompt_async(prompt))

    for task in tasks:
        if task.agent == "Emil" and task.function_name == "process_emil_request":

            # 2. Try to extract parameters from your original text
            # File: app.py, Function: extract_model_parameters_with_llm_correction
            original_params = loop.run_until_complete(
                extract_model_parameters_with_llm_correction(task.args.get('prompt', ''))
            )

            # 3. Add extracted parameters to the task
            if original_params.get('generation_types'):
                task.args['generation'] = original_params['generation_types'][0]  # "wind"
            if original_params.get('locations'):
                task.args['location'] = original_params['locations'][0]          # "Spain"

            # 4. Check if we have collected parameters from previous UI interaction
            if (hasattr(st.session_state, 'parameters_ready') and
                st.session_state.parameters_ready):
                # Use the parameters you filled in the form
                user_params = st.session_state.collected_parameters.copy()
                task.args.update(user_params)  # Add form data to task

                # Clear the parameters so they don't get reused
                st.session_state.collected_parameters = {}
                st.session_state.parameters_ready = False

            else:
                # 5. Check if parameters are still missing
                # File: app.py, Class: StreamlitParameterCollector, Function: needs_parameters
                needs_params, missing_params = StreamlitParameterCollector.needs_parameters(
                    task.args, task.function_name
                )

                if needs_params:
                    # 6. Show the parameter collection form
                    st.session_state.awaiting_parameters = True
                    # File: app.py, Function: show_parameter_form
                    show_parameter_form(missing_params, task.args)
                    return []  # Wait for user to fill the form

            # 7. Now Emil has all the parameters he needs!
            # Continue with model creation...
```

Detailed Agent Responsibilities

Keywords: agent responsibilities, task coordination, specialized functions

Nova Agent - The Smart Coordinator

Keywords: Nova agent, task coordination, intent detection, agent assignment

What Nova Does:

- Reads your requests and understands what you want
- Splits complex requests into individual tasks
- Decides which agent should handle each task
- Handles simple questions (math, general knowledge) directly

Nova's Decision Making Process: File: `src/agents/nova.py` , Function: `_create_task_with_category` (lines 150-200)

```
async def _create_task_with_category(self, intent_text: str):
    """Nova decides which agent should handle your request
    Keywords: agent assignment, task categorization, intent routing"""

    # Check if it's a math question first
    math_patterns = [
        r'\d+\s*[\+\-\*\/]\s*\d+',          # "2+2", "10*5"
        r'what\s+is\s+\d+\s*[\+\-\*\/]',    # "what is 2+2"
        r'calculate\s+\d+'                   # "calculate 25"
    ]

    is_math = any(re.search(pattern, intent_text.lower()) for pattern in math_patterns)

    if is_math:
        return Task(agent="Nova", function_name="do_maths")

    # For other requests, categorize them
    # File: utils/open_ai_utils.py, Function: open_ai_categorisation_async
    category = await open_ai_categorisation_async(intent_text, csv_path)

    # Map categories to agents
    agent_mapping = {
        "energy model": "Emil",           # Energy modeling goes to Emil
        "copywriting and proofreading": "Lola",  # Writing goes to Lola
        "general knowledge": "Nova",      # Questions go to Nova
        "math and logic": "Nova"          # Math goes to Nova
    }

    agent = agent_mapping.get(category, "Nova")
```

Emil Agent - The Energy Modeling Expert

Keywords: Emil agent, energy modeling, PLEXOS models, parameter validation

What Emil Does:

- Builds PLEXOS energy models (these are complex engineering simulations)
- Extracts technical parameters from natural language
- Validates that all required information is present
- Creates model files with timestamps

Emil's Parameter Requirements: File: `src/agents/emil.py` , Function: `handle_task_async` (lines 85-130)

```
async def handle_task_async(self, task: Task):
    """Emil's main work function
    Keywords: Emil workflow, model creation, parameter handling"""

    # Extract missing parameters from your description
    # File: src/agents/emil.py, Function: extract_energy_parameters_from_prompt (lines 25-35)
    extracted = await extract_energy_parameters_from_prompt(task.args["prompt"])
    for key, value in extracted.items():
        task.args.setdefault(key, value)  # Add extracted params to task

    # Validate parameters
    # File: src/agents/emil.py, Function: verify_parameters_async
    validation = await self.verify_parameters_async(task.function_name, task.args)

    # If parameters are missing, ask for them
    while not validation["success"] and validation.get("missing"):
        print(f" Emil needs: {validation['missing']}")
        # File: src/agents/parameter_collection.py, Function: get_missing_parameters_async
        collected = await get_missing_parameters_async(
            task.function_name,
            validation["missing"],
            task.args
        )
        task.args.update(collected)  # Add collected params
        validation = await self.verify_parameters_async(task.function_name, task.args)

    # Now Emil has everything he needs!
    if task.function_name == "process_emil_request":
        location = task.args["location"]           # "Spain"
        generation = task.args["generation"]       # "wind"
        energy_carrier = task.args.get("energy_carrier", "electricity")

        # Create a unique filename
        timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
        model_name = f"{location}_{generation}_{energy_carrier}_{timestamp}.xml"
        # Result: "Spain_wind_electricity_20241229_143022.xml"

        # Build the actual model (this calls PLEXOS software)
        # File: src/agents/emil.py, Function: build_plexos_model_with_base (lines 145-170)
        if build_plexos_model_with_base(location, generation, energy_carrier, model_path):
            result = {
                "status": "success",
                "message": f"Created {generation} {energy_carrier} model for {location}",
                "file": model_path,
                "location": location,
                "generation_type": generation,
                "energy_carrier": energy_carrier
            }
```

Lola Agent - The Report Writer

Keywords: Lola agent, report writing, documentation, context retrieval

What Lola Does:

- Takes model results and writes reports
- Creates executive summaries and technical documentation
- Uses context from previous agents (like Emil's model results)

How Lola Gets Information from Other Agents: File: `src/agents/lola.py` , Function: `handle_task_async` (lines 25-80)

```
async def handle_task_async(self, task: Task):
    """Lola writes reports using information from other agents
    Keywords: report generation, context retrieval, inter-agent communication"""

    # Get context from the session (information passed from Emil)
    session_context = task.session_context or {}

    if task.function_name == "write_report":
        # Lola looks for information left by Emil
        model_file = (session_context.get("latest_model_file") or
                      self.kb.get_item("latest_model_file"))
        model_details = (session_context.get("latest_model_details") or
                         self.kb.get_item("latest_model_details"))
        analysis_results = (session_context.get("latest_analysis_results") or
                            self.kb.get_item("latest_analysis_results"))

        # If Emil left information in the session context
        if "emil" in session_context:
            model_file = model_file or session_context["emil"].get("model_file")
            model_details = model_details or session_context["emil"].get("model_details")

        # Create the report
        # File: core/functions_registery.py, Function: write_report (lines 200-250)
        result = await asyncio.to_thread(
            global_write_report,
            self.kb,
            style=task.args.get("style", "executive_summary"),
            prompt=task.args.get("prompt", ""),
            model_file=model_file,              # Emil's model file
            model_details=model_details,        # Emil's model configuration
            analysis_results=analysis_results   # Emil's analysis results
        )
```

Ivan Agent - The Content Generator

Keywords: Ivan agent, content generation, Python scripts, visualization

What Ivan Does:

- Generates Python scripts and code
- Creates visualizations and charts
- Handles image generation tasks

File: `src/agents/ivan.py` , Function: `handle_task_async` (lines 10-80)

```
async def handle_task_async(self, task: Task):
    """Ivan handles content generation tasks
    Keywords: content generation, script generation, visualization"""

    if task.function_name == "generate_image":
        # File: src/agents/ivan.py, Function: _handle_image_task (lines 50-80)
        return await self._handle_image_task(task)

    if task.function_name in self.function_map:
        func = self.function_map[task.function_name]

        # File: core/functions_registery.py, various functions
        result = await asyncio.to_thread(func, self.kb, **task.args)
        task.result = result

        # Store results in knowledge base
        # File: core/knowledge_base.py, Function: set_item_async
        await self.kb.set_item_async(f"ivan_{task.function_name}_result", result)
        return result
```

Data Storage and Context Passing

Keywords: data storage, context passing, knowledge base, session management

How Information Flows Between Agents

The system uses several mechanisms to pass information between agents:

1. Knowledge Base Storage

Keywords: knowledge base, persistent storage, data persistence

File: `core/knowledge_base.py` , **Class:** `KnowledgeBase` **(lines 20-100)**

```
# Emil saves information for other agents to use
# File: src/agents/emil.py, Function: handle_task_async (lines 125-135)
await self.kb.set_item_async("latest_model_file", model_path)
await self.kb.set_item_async("latest_model_location", location)
await self.kb.set_item_async("latest_model_generation_type", generation)
await self.kb.set_item_async("emil_result", result, category="energy_models")

# Later, Lola retrieves this information
# File: src/agents/lola.py, Function: handle_task_async (lines 35-45)
model_file = self.kb.get_item("latest_model_file")
location = self.kb.get_item("latest_model_location")
```

2. Session Context (Agent-to-Agent Communication)

Keywords: session context, context handover, agent communication

File: `main.py` , **Function:** `interactive_async_main` **(lines 200-250)**

```
# Before sending a task to an agent, update the context
task.session_context.update({
    "latest_model_file": kb.get_item("latest_model_file"),
    "latest_model_details": kb.get_item("latest_model_details"),
    "location": kb.get_item("latest_model_location"),
    "generation_type": kb.get_item("latest_model_generation_type"),
    "energy_carrier": kb.get_item("latest_model_energy_carrier")
})

# Show what's being passed
print(f"🔄 Context handover: Emil → Lola")
print(f"   Model file: {os.path.basename(task.session_context.get('latest_model_file', ''))}")
print(f"   Location: {task.session_context.get('location')}")
print(f"   Generation: {task.session_context.get('generation_type')}")
```

3. Task Arguments (Within Same Task)

Keywords: task arguments, parameter storage, task data

File: `core/task_manager.py` , **Class:** `Task` **(lines 10-30)**

```
# Task arguments carry information within the same request
task.args = {
    "prompt": "build a wind model for spain",
    "location": "Spain",           # Extracted or collected from user
    "generation": "wind",          # Extracted or collected from user
    "energy_carrier": "electricity", # Default or collected from user
    "full_prompt": "build a wind model for spain"
}
```

4. Session Management

Keywords: session management, session data, persistence

File: `core/session_manager.py` , **Class:** `SessionManager` **(lines 20-100)**

```
def create_session(self):
    """Create a new session with unique ID and file"""
    session_id = f"session_{datetime.now().strftime('%Y%m%d_%H%M%S')}"
    session_file = os.path.join(self.base_path, f"{session_id}.json")

    session_data = {
        "id": session_id,
        "metadata": {
            "created_at": datetime.now().isoformat(),
            "session_active": True,
            "context_open": True
        },
        "prompts": [],
        "parameters": [],
        "results": []
    }
```

Configuration and Setup

Keywords: configuration, setup, generation types, locations, function mapping

Generation Types and Locations

File: `app.py` **, Global Constants (lines 50-85)**

```
# Supported energy generation types
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"]
}

# Supported locations (countries and regions)
LOCATIONS = [
    # EU members
    "Austria", "Belgium", "Bulgaria", "Croatia", "Cyprus", "Czech Republic",
    "Denmark", "Estonia", "Finland", "France", "Germany", "Greece",
    # ... more countries ...

    # Common abbreviations and alternate names
    "UK", "Great Britain", "Czechia", "Holland"
]
```

Function Mapping Configuration

File: `src/agents/Nova_function_map_enhanced.csv`

```
Key,Function,Type,Name,Args,Description
Energy Model,assistants.emil.main,Assistant,emil,prompt,Engineering. Model Building (PLEXOS)
copywriting and proofreading,assistants.lola.main,Assistant,lola,prompt,Communications expert
do_maths,utils.do_maths.do_maths,Agent,do_maths,prompt,Do math problems
general knowledge,utils.general_knowledge.answer_general_question,Agent,Nova,prompt,Answer general questions
```

File: `utils/csv_function_mapper.py` **, Class:** `FunctionMapLoader` **(lines 10-80)**

```
class FunctionMapLoader:
    """Loads function mappings from CSV files"""

    def load_function_map(self, agent_name: str) -> dict:
        """Load function map for specific agent"""
        csv_path = os.path.join("src/agents", f"{agent_name}_function_map_enhanced.csv")

        if os.path.exists(csv_path):
            df = pd.read_csv(csv_path)
            function_map = {}
            for _, row in df.iterrows():
                function_map[row['Key']] = row['Function']
            return function_map
        return {}
```

Complete Example Walkthrough with File References

Keywords: complete example, walkthrough, step-by-step, full workflow, file references, function mapping

Let's trace through: **"build a wind model for spain and write a report"** with exact file and function references

Step 1: Initial Processing

Keywords: initial processing, prompt reception, Nova coordination

```
 # 1. User input received by Nova
# File: main.py, Function: interactive_async_main (lines 180-200) OR
# File: app.py, Function: main (lines 600-650)
prompt = "build a wind model for spain and write a report"

# 2. Nova identifies multiple intents
# File: src/agents/nova.py, Function: identify_multiple_intents_async (lines 250-300)
intents = [
    {"intent": "build a wind model for spain"},
    {"intent": "write a report"}
]

# 3. Nova creates tasks
# File: src/agents/nova.py, Function: create_task_list_from_prompt_async (lines 90-150)
model_task = Task(
    name="Handle Intent: build a wind model for spain",
    agent="Emil",
    function_name="process_emil_request",
    args={"prompt": "build a wind model for spain"}
)

report_task = Task(
    name="Handle Intent: write a report",
    agent="Lola",
    function_name="write_report",
    args={"prompt": "write a report"}
)

# 4. Nova creates workflow: model_task → report_task
# File: src/agents/nova.py, Function: create_task_list_from_prompt_async (lines 220-240)
model_task.sub_tasks.append(report_task)
```

Step 2: Parameter Extraction and Collection

Keywords: parameter extraction, LLM enhancement, parameter validation

```
 # 5. System tries to extract parameters from text
# File: app.py, Function: extract_model_parameters_with_llm_correction (lines 140-184)
extracted_params = {
    "locations": ["Spain"],      # Found "spain" in prompt
    "generation_types": ["wind"], # Found "wind" in prompt
    "energy_carriers": ["electricity"] # Default value
}

# 6. Update task with extracted parameters
# File: app.py, Function: process_prompts_with_ui_params (lines 450-480)
model_task.args.update({
    "location": "Spain",
    "generation": "wind",
    "energy_carrier": "electricity"
})

# 7. Emil validates parameters
# File: src/agents/emil.py, Function: verify_parameters_async (lines 35-50)
validation = verify_parameters_async("process_emil_request", model_task.args)
# Result: {"success": True, "message": "All parameters present"}
```

Step 3: Model Creation by Emil

Keywords: Emil model creation, PLEXOS model building, timestamp generation

```
 # 8. Emil creates the energy model
# File: src/agents/emil.py, Function: handle_task_async (lines 85-130)
location = "Spain"
generation = "wind"
energy_carrier = "electricity"

# 9. Generate unique filename
# File: src/agents/emil.py, Function: handle_task_async (lines 110-115)
timestamp = "20241229_143022"
model_name = f"{location}_{generation}_{energy_carrier}_{timestamp}.xml"
# Result: "Spain_wind_electricity_20241229_143022.xml"

# 10. Emil builds the model and saves result
# File: src/agents/emil.py, Function: handle_task_async (lines 115-140)
# Calls: File: src/agents/emil.py, Function: build_plexos_model_with_base (lines 145-170)
# Which calls: File: src/agents/plexos_base_model_final.py, Function: process_base_model_task
result = {
    "status": "success",
    "message": "Created wind electricity model for Spain",
    "file": "/path/to/Spain_wind_electricity_20241229_143022.xml",
    "location": "Spain",
    "generation_type": "wind",
    "energy_carrier": "electricity"
}

# 11. Save to knowledge base for other agents
# File: src/agents/emil.py, Function: handle_task_async (lines 125-135)
# Calls: File: core/knowledge_base.py, Function: set_item_async
kb.set_item("latest_model_file", result["file"])
kb.set_item("latest_model_location", result["location"])
kb.set_item("latest_model_generation_type", result["generation_type"])
```

Step 4: Context Handover to Lola

Keywords: context handover, session context, agent communication, data passing

```
 # 12. Prepare context for report task
# File: main.py, Function: interactive_async_main (lines 250-280) OR
# File: app.py, Function: process_prompts_with_ui_params (lines 580-620)
report_task.session_context.update({
    "latest_model_file": "Spain_wind_electricity_20241229_143022.xml",
    "latest_model_location": "Spain",
    "latest_model_generation_type": "wind",
    "latest_model_energy_carrier": "electricity"
})

# 13. System shows handover
# File: main.py, Function: interactive_async_main (lines 290-310) OR
# File: app.py, Function: show_enhanced_handover (lines 350-400)
print("🔄 Context handover: Emil → Lola")
print("   Model file: Spain_wind_electricity_20241229_143022.xml")
print("   Location: Spain, Generation: wind, Carrier: electricity")
```

Step 5: Report Generation by Lola

Keywords: report generation, Lola agent, document creation, context retrieval

```
 # 14. Lola generates report using Emil's results
# File: src/agents/lola.py, Function: handle_task_async (lines 25-80)
# Calls: File: core/functions_registery.py, Function: write_report (lines 200-250)
report_result = write_report(
    kb=knowledge_base,
    style="executive_summary",
    model_file="Spain_wind_electricity_20241229_143022.xml",
    model_details={
        "location": "Spain",
        "generation_type": "wind",
        "energy_carrier": "electricity"
    }
)

# 15. Final result compilation and display
# File: main.py, Function: interactive_async_main (lines 350-400) OR
# File: app.py, Function: display_results (lines 650-750)
final_results = [
    ("Handle Intent: build a wind model for spain", emil_result, "Emil"),
    ("Handle Intent: write a report", report_result, "Lola")
]
```

Detailed Function Call Chain with File References

Keywords: function call chain, execution flow, detailed tracing, file mapping

Entry Point Flow

Streamlit App Entry (force_cli=False)

```
 # 1. User clicks "Process Prompt" button
# File: app.py, Function: main (lines 600-650)

# 2. Streamlit calls process_prompts_with_ui_params
# File: app.py, Function: process_prompts_with_ui_params (lines 400-600)

# 3. Within processing, Nova creates task list
# File: src/agents/nova.py, Function: create_task_list_from_prompt_async (lines 90-150)
tasks = loop.run_until_complete(agents["Nova"].create_task_list_from_prompt_async(prompt))
```

CLI App Entry (force_cli=True)

```
# 1. User types prompts and enters "done"
# File: src/main.py, Function: interactive_async_main (lines 100-150)

# 2. CLI processes prompts in main loop
# File: src/main.py, Function: interactive_async_main (lines 200-350)

# 3. Nova creates task list (same as Streamlit)
# File: src/agents/nova.py, Function: create_task_list_from_prompt_async (lines 90-150)
task_lists = await asyncio.gather(*(nova.create_task_list_from_prompt_async(p) for p in prompts))
```

Task Processing Flow with Parameter Collection

Keywords: task processing, parameter collection flow, agent execution

```
# 1. For each task, check if it needs parameters
# File: app.py (Streamlit) OR src/agents/emil.py (CLI)
if task.agent == "Emil" and task.function_name == "process_emil_request":

    # 2. Extract parameters using LLM
    # File: app.py, Function: extract_model_parameters_with_llm_correction (lines 140-184)
    original_params = await extract_model_parameters_with_llm_correction(task.args.get('prompt', ''))

    # 3. Check if parameters are still missing
    # File: app.py, Class: StreamlitParameterCollector, Function: needs_parameters (lines 185-220)
    needs_params, missing_params = StreamlitParameterCollector.needs_parameters(task.args, task.function_name)

    # 4a. If using Streamlit and parameters missing:
    # File: app.py, Function: show_parameter_form (lines 285-350)
    show_parameter_form(missing_params, task.args)

    # 4b. If using CLI and parameters missing:
    # File: src/agents/parameter_collection.py, Function: get_missing_parameters_async (lines 30-60)
    # Calls: File: src/agents/cli_parameter_collection.py, Function: get_missing_parameters_cli_async
    collected = await get_missing_parameters_async(task.function_name, missing_params, task.args, force_cli=True)
```

Agent Execution Flow

Keywords: agent execution, task handling, agent workflow

```
# 1. Agent receives task for execution
# File: src/agents/emil.py, Function: handle_task_async (lines 55-130)
result = await agent.handle_task_async(task)

# 2. Within Emil's handler, parameter validation occurs
# File: src/agents/emil.py, Function: verify_parameters_async (lines 35-50)
validation = await self.verify_parameters_async(task.function_name, task.args)

# 3. If parameters valid, Emil processes the request
# File: src/agents/emil.py, Function: handle_task_async (lines 85-130)
if task.function_name == "process_emil_request":
    # Model creation logic here

# 4. Emil calls PLEXOS model building
# File: src/agents/emil.py, Function: build_plexos_model_with_base (lines 145-170)
# Calls: File: src/agents/plexos_base_model_final.py, Function: process_base_model_task
if build_plexos_model_with_base(location, generation, energy_carrier, model_path):

# 5. Results stored in knowledge base
# File: src/agents/emil.py, Function: handle_task_async (lines 125-135)
# Calls: File: core/knowledge_base.py, Function: set_item_async
await self.kb.set_item_async("emil_result", result, category="energy_models")
```

Context Handover Flow

Keywords: context handover, data passing, session management

```
 # 1. Before subtask execution, context is updated
# File: main.py, Function: interactive_async_main (lines 250-280)
subtask.session_context.update({
    "latest_model_file": kb.get_item("latest_model_file"),
    "latest_model_details": kb.get_item("latest_model_details"),
    # ... other context data
})

# 2. Context handover notification displayed
# File: main.py, Function: interactive_async_main (lines 290-310)
print(f"🔄 Context handover: {task.agent} → {subtask.agent}")

# 3. Subtask agent (Lola) receives context
# File: src/agents/lola.py, Function: handle_task_async (lines 25-50)
session_context = task.session_context or {}
model_file = session_context.get("latest_model_file")
```

Result Display Flow

Keywords: result display, output formatting, user interface

Streamlit Result Display

```
 # 1. Results collected and formatted
# File: app.py, Function: process_prompts_with_ui_params (lines 580-620)
results.append((task.name, result, task.agent))

# 2. Results displayed in UI
# File: app.py, Function: display_results (lines 650-750)
def display_results(results: List[tuple]):
    st.subheader("Results:")
    for task_name, result, agent in results:
        # Format and display each result
```

CLI Result Display

```
 # 1. Results collected in main loop
# File: src/main.py, Function: interactive_async_main (lines 300-350)
results.append((task.name, result, task.agent))

# 2. Results formatted and printed
# File: src/main.py, Function: interactive_async_main (lines 380-450)
print("\n\n********\nResults\n********")
for task_name, result, agent in results:
    # Format and print each result
```

Parameter Collection Detailed Flow by Mode

Keywords: parameter collection modes, detailed flow, mode-specific behavior

Streamlit Mode Parameter Collection Flow

Keywords: streamlit parameter flow, web form handling, session state management

```
 # 1. Parameter extraction attempted
# File: app.py, Function: extract_model_parameters_with_llm_correction (lines 140-184)
original_params = await extract_model_parameters_with_llm_correction(prompt)

# 2. Parameters added to task args
# File: app.py, Function: process_prompts_with_ui_params (lines 450-480)
if original_params.get('generation_types'):
    task.args['generation'] = original_params['generation_types'][0]

# 3. Check if parameters still missing
# File: app.py, Class: StreamlitParameterCollector, Function: needs_parameters (lines 185-220)
needs_params, missing_params = StreamlitParameterCollector.needs_parameters(task.args, task.function_name)

# 4. If parameters missing, show form
# File: app.py, Function: show_parameter_form (lines 285-350)
if needs_params:
    show_parameter_form(missing_params, task.args)
    return []  # Wait for form submission

# 5. Form submission triggers rerun
# File: app.py, Function: show_parameter_form (lines 340-350)
if submitted:
    st.session_state.collected_parameters = collected_params
    st.session_state.parameters_ready = True
    st.rerun()

# 6. On rerun, collected parameters are used
# File: app.py, Function: process_prompts_with_ui_params (lines 500-520)
if st.session_state.parameters_ready:
    user_params = st.session_state.collected_parameters.copy()
    task.args.update(user_params)
```

CLI Mode Parameter Collection Flow

Keywords: CLI parameter flow, command line handling, asyncio input

```
# 1. Emil detects missing parameters during execution
# File: src/agents/emil.py, Function: handle_task_async (lines 70-85)
validation = await self.verify_parameters_async(task.function_name, task.args)
while not validation["success"] and validation.get("missing"):

# 2. Parameter collection dispatcher called
# File: src/agents/emil.py, Function: handle_task_async (lines 75-80)
collected = await get_missing_parameters_async(
    task.function_name,
    validation["missing"],
    task.args,
    force_cli=True  # This determines CLI mode
)

# 3. Dispatcher chooses CLI collection
# File: src/agents/parameter_collection.py, Function: get_missing_parameters_async (lines 30-60)
if streamlit_available and not force_cli:
    # Use Streamlit (not executed in CLI mode)
else:
    return await get_missing_parameters_cli_async(function_name, missing_params, initial_args)

# 4. CLI collection prompts user
# File: src/agents/cli_parameter_collection.py, Function: get_missing_parameters_cli_async (lines 10-60)
for param in missing_params:
    print(f"Nova: I need the '{param}' for this task.")
    value = await asyncio.to_thread(input, f"Please enter {param}: ")
    collected_args[param] = value.strip()

# 5. Collected parameters returned to Emil
# File: src/agents/emil.py, Function: handle_task_async (lines 80-85)
task.args.update(collected)
validation = await self.verify_parameters_async(task.function_name, task.args)
```

Key Dictionaries and Lists Storage Locations

Keywords: dictionaries, lists, storage locations, data structures, configuration data

1. GENERATION_TYPES Dictionary

File: `app.py`, Global Constant (lines 50-60)

- **Location**: Global constant in `app.py`
- **Purpose**: Maps generation types to specific PLEXOS categories
- **Usage**: Parameter validation and UI dropdowns
- **Keywords**: generation types, energy types, PLEXOS categories

2. LOCATIONS List

File: `app.py`, Global Constant (lines 62-85)

- **Location**: Global constant in `app.py`
- **Purpose**: Valid country/region names for energy models
- **Usage**: Location validation and correction
- **Keywords**: locations, countries, regions, geography

3. Function Maps

Files: `src/agents/*.csv` files

- **Location**: CSV files in `src/agents/` directory
- **Purpose**: Maps user intents to agent functions
- **Usage**: Loaded by `FunctionMapLoader` in `utils/csv_function_mapper.py`
- **Keywords**: function mapping, intent mapping, agent routing

4. Task Arguments

File: `core/task_manager.py`, Class: `Task` (lines 10-30)

- **Location**: Task object attribute

- **Purpose**: Stores parameters for individual tasks
- **Structure**: `{"prompt": str, "location": str, "generation": str, "energy_carrier": str}`
- **Keywords**: task arguments, task parameters, task data

5. Session Context

File: `core/task_manager.py`, Class: `Task` (lines 15-25)

- **Location**: Task object attribute
- **Purpose**: Carries context between agents
- **Structure**: `{"latest_model_file": str, "location": str, "generation_type": str, ...}`
- **Keywords**: session context, agent context, inter-agent data

6. Knowledge Base Storage

File: `core/knowledge_base.py`, Class: `KnowledgeBase` (lines 20-100)

- **Location**: Persistent storage managed by `KnowledgeBase` class
- **Purpose**: Long-term data persistence across sessions
- **Key Items**: `latest_model_file`, `latest_model_location`, `emil_result`, etc.
- **Keywords**: knowledge base, persistent storage, data persistence

7. Session Data

File: `core/session_manager.py`, Class: `SessionManager` (lines 20-100)

- **Location**: JSON files in `sessions/` directory
- **Purpose**: Complete session state including prompts, parameters, results
- **Structure**: `{"id": str, "metadata": dict, "prompts": list, "parameters": list, "results": list}`
- **Keywords**: session data, session management, session state

Summary: force_cli True vs False

Keywords: force_cli summary, parameter collection comparison, mode differences

The `force_cli` parameter fundamentally changes how users provide missing information:

force_cli=False (Streamlit Web Interface)

File: `app.py` - Default behavior in web interface

- ✅ **Better parameter extraction**: More likely to automatically extract parameters from prompts
- ✅ **User-friendly forms**: Dropdown menus, helpful descriptions, examples
- ✅ **No typing required**: Point-and-click interface
- ✅ **Visual feedback**: Progress bars, status indicators, rich UI
- ❌ **Requires web browser**: Must run Streamlit app
- ❌ **More complex setup**: Web server, session state management

force_cli=True (Command Line Interface)

File: `src/main.py` - Default behavior in CLI

- ✅ **Simple setup**: Just run Python script
- ✅ **No browser required**: Works in any terminal
- ✅ **Clear prompts**: Descriptive parameter requests
- ❌ **Manual typing**: User must type responses
- ❌ **Less extraction**: May miss parameters that Streamlit would catch
- ❌ **Basic interface**: Text-only, no visual feedback

The choice between them depends on your use case:

- **Use Streamlit (force_cli=False)** for end-user applications, demos, or when you want the best user experience
- **Use CLI (force_cli=True)** for automation, scripts, server environments, or when Streamlit isn't available

This comprehensive documentation provides a complete picture of how the AI Agent Coordinator system works, with specific file and function references, real-world comparisons, and clear explanations suitable for beginner to intermediate developers. The system's strength lies in its flexibility to work in both web and command-line environments while maintaining the same core functionality and intelligent parameter collection capabilities.