File Breakdown: `src/main.py`

File Location

`src/main.py`

Overview

The `main.py` file serves as the entry point for the multi-agent framework. It initializes the system, registers functions, creates agent instances, and handles user interactions. This file coordinates the overall workflow, from receiving user input to processing it through the appropriate agents and returning the results.

Key Responsibilities

- Import and initialize all required modules and components
- Set up the knowledge base for shared data storage
- Register available functions with the function loader
- Initialize all agent instances with proper configurations
- Process user prompts through Nova's task creation
- Execute tasks with the appropriate specialized agents
- Collect and present results back to the user
- Support both synchronous and asynchronous operation modes
- Handle single and multi-prompt processing
- Manage session history and data persistence

Core Functionality

Imports and Configuration

The file begins by importing all necessary components and setting configuration flags:

```
# Standard library imports
import asyncio
import os
import sys
import time
import datetime
import re
# Agent imports
from agents import Nova, Emil, Ivan, Lola
from agents.base_agent import BaseAgent

# Core functionality imports
from core.functions_registery import (
    build_plexos_model,
    run_plexos_model,
    analyze_results,
    write_report,
    generate_python_script,
    extract_model_parameters,
    create_single_location_model,
```

```
    create_simple_xml,
    create_multi_location_model,
    create_simple_multi_location_xml,
    create_comprehensive_model,
    create_simple_comprehensive_xml,
    process_emil_request,
    EMIL_FUNCTIONS,
    IVAN_FUNCTIONS,
    LOLA_FUNCTIONS
)
from core.task_manager import Task
from core.knowledge_base import KnowledgeBase

# Utility imports
from utils.csv_function_mapper import FunctionMapLoader
from utils.function_logger import log_function_call
from utils.do_maths import do_maths
from utils.general_knowledge import answer_general_question
from utils.open_ai_utils import (
    ai_chat_session,
    ai_spoken_chat_session,
    run_open_ai_ns,
    run_open_ai_ns_async,
    open_ai_categorisation,
    open_ai_categorisation_async
)

# === CONFIGURATION FLAG ===
# Set to True to enable asynchronous processing of multiple prompts
# Set to False to use the original synchronous processing # not working currently
USE_ASYNC_MODE = True
AUTO_DUMP_KB = True  # Set to False to disable automatic KB dumps as txt file
```

Synchronous Main Function

The original synchronous implementation for processing a single prompt:

```
@log_function_call
def main():
    """
    Main entry point for the Nova AI Coordinator system.
    Initializes agents, loads function maps (with fallbacks), processes the user
prompt,
    and executes the resulting task list.
    """
    # 1. Initialize the knowledge base
    kb = KnowledgeBase()

    # 2. Load function maps from CSV files
    try:
        print("Loading function maps from CSV files...")
        function_loader = FunctionMapLoader()
```

```
        # Register 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,
            "do_maths": do_maths,
            "answer_general_question": answer_general_question,
            "ai_chat_session": ai_chat_session,
            "ai_spoken_chat_session": ai_spoken_chat_session
        })

        # Load function maps from CSV for each agent
        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")

        # Use fallbacks if CSV loading fails
        # ...
    except Exception as e:
        print(f"Error loading CSV function maps: {str(e)}")
        print("Falling back to hardcoded function maps")
        # Fallback implementation...

    # 3. Initialize agents with loaded function maps.
    nova = Nova("Nova", kb, nova_functions)
    emil = Emil("Emil", kb, emil_functions)
    ivan = Ivan("Ivan", kb, ivan_functions)
    lola = Lola("Lola", kb, lola_functions)

    # DEBUGGING: Print Nova's registered functions
    print(f"REGISTERED FUNCTIONS FOR NOVA: {list(nova_functions.keys())}")

    # 4. Capture user prompt.
    user_prompt = input("\n\nUser prompt: ")

    # 5. Nova creates a top-level task list from the prompt.
    top_level_tasks = nova.create_task_list_from_prompt(user_prompt)

    # 6. Execute tasks in series with improved handling for multiple top-level tasks.
    print(f"\nNova will process {len(top_level_tasks)} tasks from your request:")
    for i, task in enumerate(top_level_tasks, 1):
```

```
        print(f"\n Task {i}/{len(top_level_tasks)}: {task.name}")

        # Process task and subtasks...

    # 7. Retrieve final output from knowledge base.
    final_report = kb.get_item("final_report")
    print("\n===== FINAL OUTPUT =====\n")
    print(final_report if final_report else "No final report created.")
```

Asynchronous Interactive Main Function

The enhanced asynchronous implementation for interactive use and multiple prompts:

```
@log_function_call
async def interactive_async_main():
    """
    Interactive asynchronous main entry point for the Nova AI Coordinator system.
    FIXED version with better empty line handling, enhanced categorization and
    improved history query handling.
    """
    # 1. Initialize the knowledge base - this will persist across interactions
    kb = KnowledgeBase(storage_path="knowledge_db", use_persistence=True)
    # Create a session at the start
    session_id = kb.create_session()
    print(f"Created new session: {session_id}")

    # CRITICAL FIX: Initialize session history storage explicitly with reports key
    if not kb.get_item("session_history"):
        kb.set_item("session_history", {
            "sessions": [],
            "math_questions": [],
            "general_questions": [],
            "energy_models": [],
            "reports": []  # Make sure reports is included
        })

    try:
        print("Loading function maps from CSV files...")
        function_loader = FunctionMapLoader()
        # Register available functions
        function_loader.register_functions({ /* function registrations */ })
        # Load function maps
        # ...
    except Exception as e:
        print(f"Error loading CSV function maps: {str(e)}")
        print("Falling back to hardcoded function maps")
        # Fallback implementation...

    # 3. Initialize agents with loaded function maps
    nova = Nova("Nova", kb, nova_functions)
    emil = Emil("Emil", kb, emil_functions)
    ivan = Ivan("Ivan", kb, ivan_functions)
    lola = Lola("Lola", kb, lola_functions)
```

```python
# Map of agents for easy lookup
agents = {
    "Nova": nova,
    "Emil": emil,
    "Ivan": ivan,
    "Lola": lola
}

# DEBUGGING: Print Nova's registered functions
print(f"REGISTERED FUNCTIONS FOR NOVA: {list(nova_functions.keys())}")
# Interactive loop - continue until user explicitly exits
session_count = get_next_session_id(kb)
print("\n===== Welcome to Nova AI Coordinator (Interactive Async Mode) =====")
print("Type 'exit' or 'quit' when you're done to exit the program.\n")

while True:
    print(f"\n\n===== Session {session_count} =====")
    print("***************\nAsyncronous Mode\n***************")
    print("Enter multiple prompts (type 'done' when finished, 'exit' to quit):")

    # 4. Get multiple prompts from user
    prompts = []
    while True:
        user_input = input("> ")
        if user_input.lower() in ['done', 'finished']:
            break
        elif user_input.lower() in ['exit', 'quit']:
            print("\nExiting Nova AI Coordinator. Goodbye!")
            return  # Exit the entire function

        # Handle KB dump command
        if user_input.lower() in ['dump kb', 'kb dump', 'save kb']:
            dump_file = dump_knowledge_base_to_text(kb)
            print(f"\nKnowledge base has been saved to: {dump_file}")
            continue

        # FIX: Clean up prompts and SKIP empty prompts
        if user_input.startswith('>'):
            user_input = user_input[1:].strip()

        # Skip empty lines
        if not user_input.strip():
            print("Skipping empty input line...")
            continue

        prompts.append(user_input)

    if not prompts:
        print("No prompts entered. Starting next session.")
        continue
```

```
        # CRITICAL FIX: PRE-PROCESS PROMPTS TO IDENTIFY HISTORY QUERIES
        processed_prompts = []
        for prompt in prompts:
            prompt_lower = prompt.lower()
            # Check for history query indicators
            is_history_query = (/* history query detection logic */)

            if is_history_query:
                # Flag history queries with a special prefix
                processed_prompts.append("__HISTORY__:" + prompt)
                print(f"DEBUG: Detected history query: {prompt}")
            else:
                processed_prompts.append(prompt)

        # 5. Process each prompt concurrently
        print(f"\nProcessing {len(processed_prompts)} prompts concurrently...")

        # Create tasks for all prompts simultaneously
        task_creation_coroutines = [
            nova.create_task_list_from_prompt_async(prompt) for prompt in
processed_prompts
        ]
        all_task_lists = await asyncio.gather(*task_creation_coroutines)

        # Process each prompt's tasks sequentially, but all prompts concurrently
        processing_coroutines = []
        for prompt_idx, task_list in enumerate(all_task_lists):
            # Remove the history prefix for display purposes
            display_prompt = prompts[prompt_idx]

            # Process the task list
            processing_coroutines.append(
                process_prompt_tasks(prompt_idx, display_prompt, task_list, agents,
kb)
            )

        all_results = await asyncio.gather(*processing_coroutines)

        # 6. Display all results
        for prompt_idx, result in enumerate(all_results):
            print(f"\n\n===== RESULTS FOR PROMPT {prompt_idx + 1}:
{prompts[prompt_idx]} =====")
            print(result)

        # 7. ENHANCED: Explicitly categorize and store session data
        session_data = {
            "id": session_count,
            "timestamp": datetime.datetime.now().isoformat(),
            "prompts": prompts,
            "results": all_results
        }
```

```
        # Process and store session data in knowledge base
        # ...

        # Increment session counter for next iteration
        session_count += 1
        # Archive old sessions periodically (every 5 sessions)
        if session_count % 5 == 0:
            kb.archive_old_sessions(days_threshold=7)
            print("Archived old sessions (>7 days)")
        # Ask if user wants to continue to the next session
        print("\n===== Session complete =====")
        print("Ready for next session. Enter prompts or type 'exit' to quit.")
```

Task Processing

Function for processing all tasks from a prompt:

```
@log_function_call
async def process_prompt_tasks(prompt_idx, prompt, task_list, agents, kb):
    """
    Process all tasks for a single prompt sequentially.
    Enhanced to properly handle task dependencies for modeling workflows.

    Parameters:
        prompt_idx (int): Index of the prompt being processed
        prompt (str): The prompt text
        task_list (list): List of tasks to process
        agents (dict): Dictionary of agent instances
        kb (KnowledgeBase): The knowledge base instance

    Returns:
        str: Combined result of all tasks
    """
    print(f"\n[Prompt {prompt_idx + 1}] Processing: '{prompt}'")
    print(f"[Prompt {prompt_idx + 1}] Created {len(task_list)} tasks")

    # Log this process in the knowledge base
    kb.log_interaction(prompt, f"Processing {len(task_list)} tasks", agent="System",
function="process_prompt_tasks")

    results = []

    # Detect if this is a modeling workflow (model building + analysis + report)
    modeling_workflow = False
    has_model_task = any(task.function_name == "process_emil_request" for task in
task_list)
    has_analysis_task = any(task.function_name == "analyze_results" for task in
task_list)
    has_report_task = any(task.function_name == "write_report" for task in task_list)

    if has_model_task and (has_analysis_task or has_report_task):
        modeling_workflow = True
        print(f"[Prompt {prompt_idx + 1}] Detected modeling workflow - will ensure
```

```
correct task order")

    # If this is a modeling workflow, make sure we process in the right order:
    # 1. Build the model first
    # 2. Analysis second
    # 3. Report writing last
    if modeling_workflow:
        # Reorder tasks: model building → analysis → report
        ordered_tasks = []

        # First, add model building tasks
        for task in task_list:
            if task.function_name == "process_emil_request":
                ordered_tasks.append(task)

        # Then, add analysis tasks
        for task in task_list:
            if task.function_name == "analyze_results":
                ordered_tasks.append(task)

        # Finally, add report tasks
        for task in task_list:
            if task.function_name == "write_report":
                ordered_tasks.append(task)

        # Add any remaining tasks
        for task in task_list:
            if task.function_name not in ["process_emil_request", "analyze_results",
"write_report"]:
                ordered_tasks.append(task)

        # Replace the original task list with the ordered one
        task_list = ordered_tasks
        print(f"[Prompt {prompt_idx + 1}] Reordered tasks to ensure proper workflow")

        # Log the reordering
        kb.log_interaction(prompt, "Reordered tasks for modeling workflow",
agent="System", function="process_prompt_tasks")

    # 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}")

        # Log the delegation
        kb.log_interaction(prompt, f"Delegating task {i}/{len(task_list)}: {task.name}
to {task.agent}",
                           agent="System", function="process_prompt_tasks")

        agent = agents.get(task.agent)
        if agent:
            try:
```

```
                # FIXED: Use the specific agent implementation based on agent name
                if task.agent == "Emil":
                    result = await agents["Emil"].handle_task_async(task)
                elif task.agent == "Lola":
                    result = await agents["Lola"].handle_task_async(task)
                elif task.agent == "Ivan":
                    result = await agents["Ivan"].handle_task_async(task)
                else:
                    result = await agents["Nova"].handle_task_async(task)

                # Store result and continue processing...
                # ...
            except Exception as e:
                error_msg = f"Error processing task: {str(e)}"
                print(f"[Prompt {prompt_idx + 1}] {error_msg}")
                results.append(error_msg)

                # Log the error
                kb.log_interaction(prompt, f"Error in task {i}: {str(e)}",
                                   agent="System", function="process_prompt_tasks")
        else:
            error_msg = f"No agent found for name {task.agent}"
            print(f"[Prompt {prompt_idx + 1}] {error_msg}")
            results.append(error_msg)

            # Log the error
            kb.log_interaction(prompt, f"Agent not found for task {i}: {task.agent}",
                               agent="System", function="process_prompt_tasks")

    # Log the completion
    kb.log_interaction(prompt, f"Completed processing {len(task_list)} tasks with
{len(results)} results",
                       agent="System", function="process_prompt_tasks")

    # Combine results
    if not results:
        return "No results found for any tasks."
    elif len(results) == 1:
        return results[0]
    else:
        combined = "\n\n" + "-"*40 + "\n\n".join([str(result) for result in results])
+ "\n\n" + "-"*40
        return combined
```

Utility Functions

Several utility functions to support the main operations:

```
def dump_knowledge_base_to_text(kb, filename=None, silent=False):
    """
    Dumps the contents of the knowledge base to a human-readable text file.

    Parameters:
```

```
        kb (KnowledgeBase): The knowledge base object
        filename (str, optional): Custom filename. If None, uses timestamp
        silent (bool): Whether to suppress console output

    Returns:
        str: Path to the created file
    """
    # Implementation...

def extract_math_result(text):
    """
    Extract math calculation result from text.

    Parameters:
        text (str): Text containing math result

    Returns:
        str: Extracted result or original text
    """
    # Implementation...

def extract_model_result(text):
    """
    Extract energy model result from text.

    Parameters:
        text (str): Text containing model creation result

    Returns:
        str: Extracted result or original text
    """
    # Implementation...

def extract_country(text):
    """
    Extract country name from text.

    Parameters:
        text (str): Text to extract country from

    Returns:
        str or None: Extracted country or None
    """
    # Implementation...
```

Main Entry Point

The script's main entry point decides which mode to run based on configuration:

```
if __name__ == "__main__":
    # Check the configuration flag to determine which mode to run
    if USE_ASYNC_MODE:
        print("Running in INTERACTIVE ASYNCHRONOUS mode - multiple prompts can be
```

```
processed concurrently")
        asyncio.run(interactive_async_main())  # Use the new interactive version
    else:
        print("Running in SYNCHRONOUS mode - original single-prompt processing")
        main()
```

Key Features

1. **Dual Operation Modes**: Supports both synchronous and asynchronous processing
2. **Interactive Interface**: Provides an interactive interface for user prompts
3. **Multi-Prompt Processing**: Handles multiple prompts concurrently in async mode
4. **Knowledge Base Integration**: Uses a persistent knowledge base for data storage
5. **Agent Coordination**: Initializes and coordinates all specialized agents
6. **Task Workflow Management**: Detects and enforces proper task ordering for complex workflows
7. **Error Handling**: Implements comprehensive error handling and logging
8. **Session Management**: Tracks sessions and archives old data
9. **Utility Functions**: Provides helper functions for common operations

Integration

- Serves as the entry point for the entire application
- Coordinates all agent activities (Nova, Emil, Ivan, Lola)
- Interfaces with the knowledge base for data persistence
- Uses the function registry to access available functions
- Manages the user interface for input and output

Workflow

1. Initialize the knowledge base and create a new session
2. Load function maps and initialize agents
3. Accept one or more prompts from the user
4. Preprocess prompts to identify special cases (history queries, etc.)
5. Create tasks for each prompt using Nova's intent detection
6. Detect and handle complex workflows by reordering tasks as needed
7. Execute each task with the appropriate agent
8. Collect and combine results
9. Present the results to the user
10. Store session data in the knowledge base
11. Repeat until the user chooses to exit

Implementation Notes

- Uses asyncio for concurrent task processing
- Provides fallback mechanisms when function maps fail to load
- Implements special handling for history-related queries
- Detects and manages dependencies in energy modeling workflows
- Supports knowledge base dumping for backup purposes
- Automatically archives old sessions to manage storage
- Categories and stores different types of interactions (math, general knowledge, etc.)