***2 - 1/4/2025***

Based on the conversation, here are the key issues that were causing the system not to work as originally expected:

* **API Key Configuration:** The OpenAI API key wasn’t being correctly loaded—either from the get\_api\_keys.py module or via the OPENAI\_API\_KEY environment variable. This misconfiguration led to immediate API call failures.
* **Missing or Inconsistent Parameter Verification:** The Emil agent was missing the necessary verify\_parameters (and its async counterpart) methods. This absence caused errors like:  
  + "'Emil' object has no attribute 'verify\_parameters'"
  + "Missing required parameters: args, kwargs"  
     Without proper verification, the energy model tasks couldn’t gather the required inputs.
* **Async Workflow Discrepancies:** There was a mismatch between synchronous and asynchronous method implementations. For example, functions like create\_task\_list\_from\_prompt\_async and identify\_multiple\_intents\_async weren’t consistently available or correctly integrated. This led to issues where the system didn’t prompt interactively as expected.
* **Interactive Parameter Collection Not Triggering:** The previous (synchronous) implementation prompted the user to provide missing energy model parameters. In the async version, this interactive collection wasn’t occurring, meaning that when a prompt like "build a model" was received, the system defaulted to using empty parameters instead of asking the user for the missing details.
* **Categorization and Function Mapping Problems:** Even though the CSV-based function map clearly defines an “Energy Model” category, the categorization logic (via open\_ai\_categorisation\_async) sometimes did not select it. This discrepancy caused the system to misroute the task or fail to trigger the correct interactive parameter collection workflow.
* **Lack of Parameter Validation and Correction:** Common issues like typos in energy model parameters (e.g., "finalnd" instead of "Finland" or "solatr" instead of "solar") weren’t being corrected. A validation function (such as validate\_and\_correct\_energy\_params) was needed to handle these cases and ensure the parameters were properly formatted before processing.

Each of these issues contributed to the overall failure of the system to operate as intended, especially in handling the "build a model" task in an asynchronous context.

***Things done***  
  
**Understanding Asynchronous Programming in Your Agent System**

Based on your code and test output, I can see you've implemented an asynchronous framework for your multi-agent system. Let me explain how asynchronous programming works and assess your implementation.

## **How Asynchronous Programming Works**

Asynchronous programming allows operations to execute concurrently without blocking the main program flow. This is particularly valuable for:

* I/O-bound operations (network requests, file access)
* Handling multiple operations simultaneously
* Improving responsiveness in interactive applications

In Python, asynchronous programming is built around:

1. **Coroutines** (async def functions that can pause with await)
2. **Event Loop** (manages scheduled coroutines)
3. **Tasks** (units of work scheduled on the event loop)
4. **Gather** (runs multiple coroutines concurrently and waits for all to complete)

## **Your Asynchronous Implementation**

You've built a solid foundation for asynchronous processing:

### **Strengths:**

1. **Dual Mode Architecture**: You maintain both sync (main()) and async (async\_main()) paths

**Parallel Prompt Processing**: You use asyncio.gather() to process multiple user prompts concurrently:  
  
 task\_creation\_coroutines = [

nova.create\_task\_list\_from\_prompt\_async(prompt) for prompt in prompts

]

all\_task\_lists = await asyncio.gather(\*task\_creation\_coroutines)

1. **Async Agent Methods**: Your agent classes have async versions of key methods:  
   * handle\_task\_async()
   * verify\_parameters\_async()
   * identify\_multiple\_intents\_async()
2. **Thread-Safe Knowledge Base**: Your KnowledgeBase class uses asyncio.Lock() for thread safety in async operations
3. **Async API Calls**: You've implemented async OpenAI API calls (run\_open\_ai\_ns\_async())

### **Test Results Analysis**

Your test with three prompts demonstrates partial concurrency:

Processing 3 prompts concurrently...

The system successfully processes three different prompts:

1. "What is 2 \* 50" (math calculation)
2. "build a model" (energy model creation)
3. "What is 50% of 100" (another math calculation)

All three finish and return results, showing the async framework is functioning.

## **Areas for Improvement**

While your implementation is working, there are some limitations:

**Sequential Task Execution**: Within each prompt, tasks are processed sequentially rather than concurrently:  
  
 # Process each task in this prompt SEQUENTIALLY

for i, task in enumerate(task\_list, 1):

# ...

**Blocking User Input**: The interactive input for the energy model parameters blocks the event loop:  
  
 Nova: Could you please tell me which type of energy generation...

> bio

1. This creates a synchronous barrier in your otherwise async flow.
2. **Mixed Async/Sync Functions**: Some functions like process\_emil\_request are still synchronous, which can block the event loop when called from async functions.
3. **Per-Call API Sessions**: Each API call creates a new session, which is less efficient than reusing sessions.

## **How Close Are You?**

Your implementation is approximately 70-80% of the way to a fully asynchronous system. You've:

* Successfully implemented the framework for async processing
* Made most key components async-compatible
* Demonstrated concurrent processing of multiple prompts

The main limitations are:

* Interactive user input creating synchronous points
* Sequential task processing within each prompt
* Some remaining synchronous functions

## **Recommendations**

1. Consider using a message queue for user interactions to avoid blocking

Make task processing within a prompt concurrent where possible:  
  
 # Process tasks in a prompt concurrently

task\_coroutines = [agent.handle\_task\_async(task) for task in tasks]

results = await asyncio.gather(\*task\_coroutines)

1. Complete the async conversion of remaining synchronous functions

Overall, you've built a solid async framework that works well for concurrent prompt processing. The remaining limitations are common challenges in interactive agent systems and would require more advanced techniques to overcome.

Current Flow

# **Complete Flow Breakdown of Your Asynchronous Agent System**

Below is a detailed breakdown of your agent system's architecture and execution flow, including file relationships, code mappings, and the processing sequence.

## **Key Files and Their Roles**

| **File** | **Description** | **Key Components** |
| --- | --- | --- |
| **main.py** | Entry point | main(), async\_main(), process\_prompt\_tasks() |
| **base\_agent.py** | Base agent class | BaseAgent, handle\_task(), handle\_task\_async() |
| **nova.py** | Primary coordinator agent | Nova, create\_task\_list\_from\_prompt\_async(), identify\_multiple\_intents\_async() |
| **emil.py** | Energy modeling agent | Emil, verify\_parameters\_async(), handle\_task\_async() |
| **ivan.py** | Python script generation | Ivan, handle\_task\_async() |
| **lola.py** | Report writing agent | Lola, handle\_task\_async() |
| **knowledge\_base.py** | Data storage | KnowledgeBase, set\_item\_async(), get\_item\_async() |
| **task\_manager.py** | Task definition | Task, add\_subtask(), mark\_completed() |
| **functions\_registery.py** | Available functions | process\_emil\_request(), build\_plexos\_model(), create\_comprehensive\_model() |
| **open\_ai\_utils.py** | OpenAI integration | run\_open\_ai\_ns\_async(), open\_ai\_categorisation\_async() |
| **do\_maths.py** | Math calculation | do\_maths() |
| **general\_knowledge.py** | General Q&A | answer\_general\_question() |
| **csv\_function\_mapper.py** | CSV to function mapping | FunctionMapLoader, load\_function\_map() |
| **function\_logger.py** | Logging | log\_function\_call decorator |

## **Asynchronous Execution Flow**

### **1. Program Initialization**

# main.py

if USE\_ASYNC\_MODE:

asyncio.run(async\_main()) # Asynchronous entry point

### **2. Function Registration and Agent Setup**

# main.py -> async\_main()

function\_loader = FunctionMapLoader()

function\_loader.register\_functions({...}) # Register available functions

nova\_functions = function\_loader.load\_function\_map("Nova") # Load from CSV

emil\_functions = function\_loader.load\_function\_map("Emil")

# Initialize agents

nova = Nova("Nova", kb, nova\_functions)

emil = Emil("Emil", kb, emil\_functions)

# Map of agents for easy lookup

agents = {"Nova": nova, "Emil": emil, "Ivan": ivan, "Lola": lola}

### **3. Prompt Collection**

# main.py -> async\_main()

prompts = []

while True:

user\_input = input("> ")

if user\_input.lower() == 'done':

break

prompts.append(user\_input)

### **4. Concurrent Task Creation**

# main.py -> async\_main()

task\_creation\_coroutines = [

nova.create\_task\_list\_from\_prompt\_async(prompt) for prompt in prompts

]

all\_task\_lists = await asyncio.gather(\*task\_creation\_coroutines)

### **5. Parallel Prompt Processing**

# main.py -> async\_main()

processing\_coroutines = []

for prompt\_idx, task\_list in enumerate(all\_task\_lists):

processing\_coroutines.append(

process\_prompt\_tasks(prompt\_idx, prompts[prompt\_idx], task\_list, agents, kb)

)

all\_results = await asyncio.gather(\*processing\_coroutines)

## **Detailed Flow for a Single Prompt**

Let's trace a specific prompt: "build a model":

**Intent Identification** # nova.py

multiple\_intents = await self.identify\_multiple\_intents\_async(prompt)

# Uses OpenAI API to analyze if the prompt contains multiple requests

**Intent Categorization** # nova.py -> create\_task\_list\_from\_prompt\_async()

category = await open\_ai\_categorisation\_async(intent\_text, csv\_path)

# "build a model" -> "Energy Model"

**Task Creation** # nova.py -> create\_task\_for\_category()

# For "Energy Model" category:

target\_agent = "Emil"

function\_name = "process\_emil\_request"

task\_args = {"prompt": intent\_text}

**Parameter Collection** # nova.py -> get\_energy\_parameters\_from\_user\_async()

# Collects location, generation type, and energy carrier

question = await run\_open\_ai\_ns\_async(prompt, context)

user\_response = await asyncio.to\_thread(input, "> ")

**Task Delegation to Emil** # main.py -> process\_prompt\_tasks()

agent = agents.get(task.agent) # Gets Emil

await agent.handle\_task\_async(task)

**Parameter Verification** # emil.py -> handle\_task\_async()

verification = await self.verify\_parameters\_async(task.function\_name, task.args)

**Model Creation** # emil.py -> handle\_task\_async()

# Calls process\_emil\_request in a thread pool

result = await asyncio.to\_thread(func, self.kb, \*\*task\_args)

# functions\_registery.py -> process\_emil\_request()

# Creates comprehensive model with parameters

result = create\_comprehensive\_model(locations, generation\_types, carriers)

**Result Storage** # emil.py -> handle\_task\_async()

task.result = result

await self.kb.set\_item\_async("emil\_formatted\_result", formatted\_result)

await self.kb.set\_item\_async("emil\_result", result)

## **Flow for a Math Calculation Prompt**

For "What is 2 \* 50":

1. Categorized as "do\_maths" by open\_ai\_categorisation\_async()
2. Delegated to Nova's do\_maths function
3. Result calculated and stored in the knowledge base
4. Displayed to user: "The result of 2.0 \* 50.0 is 100.0"

## **Async vs. Sync Implementation Mapping**

| **Synchronous Method** | **Asynchronous Equivalent** |
| --- | --- |
| main() | async\_main() |
| handle\_task() | handle\_task\_async() |
| ask\_user\_for\_missing\_args() | ask\_user\_for\_missing\_args\_async() |
| create\_task\_list\_from\_prompt() | create\_task\_list\_from\_prompt\_async() |
| identify\_multiple\_intents() | identify\_multiple\_intents\_async() |
| verify\_parameters() | verify\_parameters\_async() |
| kb.set\_item() | kb.set\_item\_async() |
| run\_open\_ai\_ns() | run\_open\_ai\_ns\_async() |

## **Execution Model**

Your system uses a hybrid approach:

* **Parallel Across Prompts**: Different user prompts are processed concurrently
* **Sequential Within Prompts**: Tasks within a single prompt are processed sequentially
* **Thread Pool for Blocking Calls**: Uses asyncio.to\_thread() to run blocking functions

This is visualized as:

Prompt 1 ──> Task 1.1 ─> Task 1.2 ─> Task 1.3 ─┐

│

Prompt 2 ──> Task 2.1 ─> Task 2.2 ─────────────┼─> Results Gathering

│

Prompt 3 ──> Task 3.1 ─────────────────────────┘

The output in your test case shows this flow in action, with three prompts being processed concurrently, but the tasks within each prompt (especially for "build a model" which required user input) being processed sequentially.