# **Enhanced Workflow Mapping with Code References**

## **1. Initial User Interaction**

**Workflow Component:** User Prompt **File:** main.py **Function:** main()

# In main.py

def main():

# Initialize system components

kb = KnowledgeBase()

task\_manager = TaskManager(kb)

# ... other initialization

# Capture user prompt

user\_prompt = input("\n\nUser prompt: ")

# Process the prompt through the enhanced workflow

# ...

**Explanation:** Entry point where the system receives user instructions. This prompt will be analyzed and broken down into executable tasks using the enhanced task creation pipeline.

## **2. Prompt Analysis & Initial Categorization**

**Workflow Component:** Coordination Agent (Nova) **File:** nova.py **Class:** Nova (extends BaseAgent) **Function:** categorize\_prompt()

# In nova.py

class Nova(BaseAgent):

@log\_function\_call

def categorize\_prompt(self, prompt: str) -> str:

"""

Analyzes and categorizes the user prompt using LLM.

"""

from utils.open\_ai\_utils import open\_ai\_categorisation

# Use the LLM to categorize the prompt

csv\_path = os.path.join(os.path.dirname(\_\_file\_\_), "Nova\_function\_map\_enhanced.csv")

category = open\_ai\_categorisation(prompt, csv\_path)

print(f"Prompt categorized as: {category}")

return category

**Explanation:** Nova first analyzes the prompt using an LLM to determine its general category (math problem, energy model, website, etc.). This is the initial step in the enhanced workflow.

## **3. Metadata Extraction**

**Workflow Component:** Extract Metadata from Prompt **File:** parameter\_analyzer.py **Class:** ParameterAnalyzer **Function:** extract\_parameters\_from\_prompt()

# In parameter\_analyzer.py

class ParameterAnalyzer:

def \_\_init\_\_(self, function\_registry=None):

self.function\_registry = function\_registry

@log\_function\_call

def extract\_parameters\_from\_prompt(self, prompt: str, category: str = None) -> Dict[str, Any]:

"""

Extract parameters from the user's prompt based on category.

"""

parameters = {}

# For energy modeling tasks

if category and category.lower() in ["energy model"]:

# Import the specialized extraction function

from core.functions\_registery import extract\_model\_parameters

energy\_params = extract\_model\_parameters(prompt)

parameters.update(energy\_params)

# For website tasks

elif category and category.lower() in ["open a website", "website"]:

# Extract website URL or search terms

url\_match = re.search(r'https?://\S+', prompt)

if url\_match:

parameters["url"] = url\_match.group(0)

else:

terms = re.sub(r'^(open|navigate to|go to|visit|browse)\s+', '', prompt, flags=re.IGNORECASE)

parameters["search\_terms"] = terms

# For math problems

elif category and category.lower() in ["do\_maths", "math", "calculation"]:

parameters["math\_expression"] = prompt

# Extract generic parameters like dates, locations, numbers

# ...

return parameters

**Explanation:** The ParameterAnalyzer extracts structured parameters from the user's prompt, using specialized extraction logic based on the category. This enhances the previous approach by separating parameter extraction from task creation.

## **4. Centralized Task List Creation**

**Workflow Component:** Centralized Task List Creator **File:** task\_creator.py **Class:** TaskListCreator **Function:** create\_task\_list\_from\_prompt()

# In task\_creator.py

class TaskListCreator:

def \_\_init\_\_(self, llm\_service, parameter\_analyzer):

self.llm\_service = llm\_service

self.parameter\_analyzer = parameter\_analyzer

@log\_function\_call

def create\_task\_list\_from\_prompt(self, prompt: str) -> List[Task]:

"""

Creates a comprehensive task list from the user prompt with hierarchical structure.

"""

# Determine prompt category

category = self.categorize\_prompt(prompt)

# Extract parameters

parameters = self.parameter\_analyzer.extract\_parameters\_from\_prompt(prompt, category)

# Create initial high-level task

high\_level\_task = self.create\_high\_level\_task(category, prompt, parameters)

# Break down into subtasks (recursive)

self.break\_down\_task(high\_level\_task)

# Create dependency graph

self.create\_dependency\_graph([high\_level\_task])

return [high\_level\_task]

def break\_down\_task(self, task, max\_depth=3, current\_depth=0):

"""

Recursively breaks down tasks into subtasks.

"""

if current\_depth >= max\_depth:

return

# Generate subtasks based on task type

if task.function\_name == "process\_emil\_request":

# Create modeling subtasks

task.add\_subtask(Task(

name="Extract Energy Parameters",

description="Extract energy modeling parameters from prompt",

agent="Emil",

function\_name="extract\_model\_parameters",

args={"prompt": task.args.get("prompt", "")}

))

task.add\_subtask(Task(

name="Create Energy Model",

description="Create energy model with parameters",

agent="Emil",

function\_name="create\_comprehensive\_model",

args={} # Will be filled in during execution

))

# ... more subtasks

# Continue breaking down each subtask

for subtask in task.sub\_tasks:

self.break\_down\_task(subtask, max\_depth, current\_depth + 1)

**Explanation:** The TaskListCreator builds a comprehensive task hierarchy with high-level goal tasks, mid-level functional tasks, and atomic function-level tasks. This is a major enhancement over the previous flat task structure.

## **5. Dependency Graph Creation**

**Workflow Component:** Dependency Graph Creation **File:** task\_creator.py **Class:** TaskListCreator **Function:** create\_dependency\_graph()

# In task\_creator.py (continued)

class TaskListCreator:

# ... other methods

@log\_function\_call

def create\_dependency\_graph(self, tasks: List[Task]):

"""

Creates a dependency graph between tasks and subtasks.

"""

for task in tasks:

# Set dependencies between subtasks (if any)

for i, subtask in enumerate(task.sub\_tasks):

if i > 0:

# Each subtask depends on the previous one

previous = task.sub\_tasks[i-1]

subtask.dependencies.append(previous.id)

# Process nested subtasks

self.create\_dependency\_graph(task.sub\_tasks)

**Explanation:** This component creates dependencies between tasks, ensuring they execute in the correct order. This is critical for the enhanced workflow with parallel and sequential execution.

## **6. Complete Parameter Analysis**

**Workflow Component:** Complete Parameter Analysis **File:** parameter\_analyzer.py **Class:** ParameterAnalyzer **Function:** identify\_missing\_parameters()

# In parameter\_analyzer.py (continued)

class ParameterAnalyzer:

# ... other methods

@log\_function\_call

def identify\_missing\_parameters(self, function\_name: str, existing\_params: Dict[str, Any]) -> List[str]:

"""

Identifies which parameters are missing for a function.

"""

# Get required parameters for the function

required\_params = self.get\_parameter\_requirements(function\_name)

# Find missing parameters

missing = [param for param in required\_params if param not in existing\_params]

return missing

def get\_parameter\_requirements(self, function\_name: str) -> List[str]:

"""

Gets the required parameters for a function using introspection.

"""

if not self.function\_registry or function\_name not in self.function\_registry:

# Default requirements for known functions

if function\_name == "process\_emil\_request":

return ["prompt", "location", "generation", "energy\_carrier"]

elif function\_name == "open\_website":

return ["prompt"]

# ... other known functions

return []

# Use introspection to analyze function signature

function = self.function\_registry.get\_function(function\_name)

import inspect

sig = inspect.signature(function)

# Identify required parameters (no default value, not self/kb)

return [p.name for p in sig.parameters.values()

if p.default == inspect.Parameter.empty

and p.name != 'self' and p.name != 'kb']

**Explanation:** This enhanced parameter analysis is more thorough, using function introspection to precisely identify what parameters are required for each function, improving the user experience.

## **7. Front-loaded User Interaction**

**Workflow Component:** Front-loaded User Interaction **File:** user\_interaction.py **Class:** UserInteractionManager **Function:** collect\_missing\_parameters()

# In user\_interaction.py

class UserInteractionManager:

def \_\_init\_\_(self, llm\_service):

self.llm\_service = llm\_service

@log\_function\_call

def collect\_missing\_parameters(self, missing\_params: List[str], context: str = None) -> Dict[str, Any]:

"""

Collects all missing parameters at once through conversational interaction.

"""

print(f"\nNova will collect some information about your {context or 'request'}...")

collected\_params = {}

for param in missing\_params:

# Generate a conversational question for this parameter

question = self.generate\_parameter\_question(param, context)

# Ask the user

print(f"\nNova: {question}")

user\_response = input("> ").strip()

# Store the response

collected\_params[param] = user\_response

return collected\_params

def generate\_parameter\_question(self, param\_name: str, context: str = None) -> str:

"""

Generates a conversational question using LLM.

"""

# Context for the parameter

param\_contexts = {

"location": "Ask about geographic locations for energy modeling. Options include UK, France, Spain...",

"generation": "Ask about energy generation types. Options include solar, wind, hydro...",

"energy\_carrier": "Ask about energy carriers. Options include electricity, hydrogen, methane..."

# ... other parameters

}

# Create prompt for LLM

context\_text = param\_contexts.get(param\_name, f"Ask about the {param\_name} parameter")

prompt = f"Ask the user to provide the {param\_name} for their {context or 'request'}."

system\_prompt = (

f"You are Nova, a friendly AI assistant for {context or 'general tasks'}. "

f"Ask ONE clear, conversational question to get specific information. "

f"Specific guidance: {context\_text}"

)

# Generate question using LLM

return self.llm\_service.generate\_text(prompt, system\_prompt)

**Explanation:** This component collects all required parameters at once, before execution begins, preventing multiple interruptions. It uses the LLM to generate natural-sounding questions.

## **8. Global Execution Optimization**

**Workflow Component:** Global Execution Optimization **File:** execution\_optimizer.py **Class:** ExecutionOptimizer **Function:** optimize\_task\_execution()

# In execution\_optimizer.py

class ExecutionOptimizer:

@log\_function\_call

def optimize\_task\_execution(self, tasks: List[Task]) -> Dict[str, Any]:

"""

Analyzes tasks to find the optimal execution strategy.

"""

# Analyze task complexity and interdependencies

task\_analysis = self.analyze\_tasks(tasks)

# Select execution strategy (parallel or sequential)

strategy = self.select\_execution\_strategy(task\_analysis)

if strategy == "parallel":

# Identify tasks that can run in parallel

parallel\_groups = self.identify\_parallel\_opportunities(tasks)

return {

"strategy": "parallel",

"groups": parallel\_groups

}

else:

# Optimize sequential order

ordered\_tasks = self.optimize\_sequential\_order(tasks)

return {

"strategy": "sequential",

"ordered\_tasks": ordered\_tasks

}

def analyze\_tasks(self, tasks: List[Task]) -> Dict[str, Any]:

"""

Analyzes task complexity, dependencies, and resources required.

"""

# ... implementation

def select\_execution\_strategy(self, task\_analysis: Dict[str, Any]) -> str:

"""

Determines whether parallel or sequential execution is optimal.

"""

# ... implementation

**Explanation:** This new component analyzes the entire task structure to determine the optimal execution strategy, considering task dependencies, complexity, and resource requirements.

## **9. Dynamic Task Priority Queue**

**Workflow Component:** Dynamic Task Priority Queue **File:** task\_scheduler.py **Class:** TaskPriorityQueue **Functions:** add\_task(), get\_next\_task()

# In task\_scheduler.py

class TaskPriorityQueue:

def \_\_init\_\_(self):

self.tasks = [] # List of (task, priority) tuples

@log\_function\_call

def add\_task(self, task: Task, priority: int = 0):

"""

Adds a task to the priority queue.

"""

self.tasks.append((task, priority))

self.tasks.sort(key=lambda x: x[1], reverse=True)

@log\_function\_call

def get\_next\_task(self) -> Optional[Task]:

"""

Gets the highest priority task from the queue.

"""

if not self.tasks:

return None

return self.tasks.pop(0)[0]

@log\_function\_call

def update\_priority(self, task\_id: str, new\_priority: int):

"""

Updates the priority of a task in the queue.

"""

for i, (task, \_) in enumerate(self.tasks):

if task.id == task\_id:

self.tasks[i] = (task, new\_priority)

break

# Re-sort the queue

self.tasks.sort(key=lambda x: x[1], reverse=True)

**Explanation:** This component manages task execution order based on dynamic priorities, allowing the system to adapt as execution progresses.

## **10. Execution Strategy Selection**

**Workflow Component:** Execution Strategy Selection **File:** task\_execution.py **Class:** TaskExecutor **Function:** execute\_tasks()

# In task\_execution.py

class TaskExecutor:

def \_\_init\_\_(self, agent\_registry, kb):

self.agent\_registry = agent\_registry

self.kb = kb

@log\_function\_call

def execute\_tasks(self, execution\_plan: Dict[str, Any]) -> List[Any]:

"""

Executes tasks according to the execution plan.

"""

strategy = execution\_plan.get("strategy", "sequential")

if strategy == "parallel":

# Execute tasks in parallel groups

return self.execute\_parallel\_tasks(execution\_plan.get("groups", []))

else:

# Execute tasks in sequence

return self.execute\_sequential\_tasks(execution\_plan.get("ordered\_tasks", []))

@log\_function\_call

def execute\_parallel\_tasks(self, task\_groups: List[List[Task]]) -> List[Any]:

"""

Executes groups of tasks in parallel.

"""

all\_results = []

for group in task\_groups:

# Use threading to execute tasks in parallel

import threading

threads = []

results = [None] \* len(group)

for i, task in enumerate(group):

thread = threading.Thread(

target=self.\_execute\_task\_thread,

args=(task, i, results)

)

threads.append(thread)

thread.start()

# Wait for all threads to complete

for thread in threads:

thread.join()

all\_results.extend(results)

return all\_results

@log\_function\_call

def execute\_sequential\_tasks(self, tasks: List[Task]) -> List[Any]:

"""

Executes tasks in sequence.

"""

results = []

for task in tasks:

result = self.execute\_single\_task(task)

results.append(result)

return results

def \_execute\_task\_thread(self, task: Task, index: int, results: List[Any]):

"""

Helper function for executing a task in a thread.

"""

result = self.execute\_single\_task(task)

results[index] = result

**Explanation:** This component handles both parallel and sequential execution strategies, adapting to the characteristics of the tasks.

## **11. Function Execution Monitor**

**Workflow Component:** Function Execution Monitor **File:** function\_monitor.py **Class:** FunctionMonitor **Function:** monitor\_execution()

# In function\_monitor.py

class FunctionMonitor:

def \_\_init\_\_(self, error\_handler):

self.error\_handler = error\_handler

@log\_function\_call

def monitor\_execution(self, function, args, task=None, context=None):

"""

Monitors the execution of a function, handling errors and timeouts.

"""

import time

start\_time = time.time()

try:

# Set up timeout monitoring if needed

timeout = args.pop("\_timeout", None)

if timeout:

# Implementation with signal or threading.Timer

pass

# Execute the function

result = function(\*\*args)

# Record performance metrics

execution\_time = time.time() - start\_time

print(f"Function {function.\_\_name\_\_} executed in {execution\_time:.2f} seconds")

return {

"status": "success",

"result": result,

"execution\_time": execution\_time

}

except Exception as e:

# Record error

execution\_time = time.time() - start\_time

print(f"Error executing {function.\_\_name\_\_}: {str(e)}")

# Handle the error

if self.error\_handler:

return self.error\_handler.handle\_error(e, task, context)

else:

return {

"status": "error",

"error": str(e),

"execution\_time": execution\_time

}

**Explanation:** This new component monitors function execution, detecting errors and timeouts, and collecting performance metrics.

## **12. Error Classification and Resolution**

**Workflow Component:** Error Classification, Error Resolution **File:** error\_handler.py **Class:** ErrorHandler **Functions:** classify\_error(), determine\_resolution\_strategy()

# In error\_handler.py

class ErrorHandler:

def \_\_init\_\_(self, agent\_registry):

self.agent\_registry = agent\_registry

@log\_function\_call

def handle\_error(self, error: Exception, task: Task, context: Dict[str, Any] = None) -> Dict[str, Any]:

"""

Handles an error that occurred during task execution.

"""

# Classify the error

error\_class = self.classify\_error(error)

# Determine resolution strategy

resolution = self.determine\_resolution\_strategy(error\_class, task)

# Execute resolution

if resolution == "branch":

return self.handle\_branch(task, error)

elif resolution == "alternative":

return self.handle\_alternative(task, error)

elif resolution == "new\_function":

return self.handle\_new\_function(task, error)

else:

# Default: just return the error

return {

"status": "error",

"message": str(error),

"error\_class": error\_class

}

@log\_function\_call

def classify\_error(self, error: Exception) -> str:

"""

Classifies error into types.

"""

error\_str = str(error)

if "not found" in error\_str or "not defined" in error\_str:

return "missing\_function"

elif "missing required" in error\_str or "required parameter" in error\_str:

return "parameter\_error"

elif "timeout" in error\_str:

return "timeout"

elif "permission" in error\_str or "access" in error\_str:

return "permission\_error"

elif "connection" in error\_str or "network" in error\_str:

return "network\_error"

else:

return "execution\_error"

@log\_function\_call

def determine\_resolution\_strategy(self, error\_class: str, task: Task) -> str:

"""

Determines the best resolution strategy for the error.

"""

if error\_class == "missing\_function":

return "new\_function"

elif error\_class == "parameter\_error":

return "branch"

elif error\_class in ["network\_error", "timeout"]:

return "alternative"

else:

# Use IVAN for complex errors

if self.agent\_registry and self.agent\_registry.get\_agent("Ivan"):

return "new\_function"

else:

return "error"

**Explanation:** These components provide sophisticated error handling, classifying errors and selecting appropriate resolution strategies.

## **13. IVAN Integration for Error Handling**

**Workflow Component:** IVAN (for Error Analysis) **File:** ivan.py **Class:** Ivan (extends BaseAgent) **Function:** analyze\_error()

# In ivan.py

class Ivan(BaseAgent):

@log\_function\_call

def analyze\_error(self, error: Exception, context: Dict[str, Any]) -> Dict[str, Any]:

"""

Analyzes an error and proposes solutions.

"""

error\_message = str(error)

error\_type = type(error).\_\_name\_\_

# Get relevant context for the error

task = context.get("task")

function\_name = task.function\_name if task else None

args = task.args if task else {}

# Use LLM to analyze the error

prompt = (

f"Analyze this error and propose a solution:\n"

f"Error Type: {error\_type}\n"

f"Error Message: {error\_message}\n"

f"Function: {function\_name}\n"

f"Arguments: {json.dumps(args, indent=2)}"

)

system\_prompt = (

"You are Ivan, an expert in debugging and problem-solving. "

"Analyze the error and propose a detailed solution. "

"Format your response as a JSON with 'analysis', 'solution', and 'code\_fix' fields."

)

# Get analysis from LLM

analysis\_json = self.llm\_service.generate\_json(prompt, system\_prompt)

return {

"status": "analyzed",

"error": error\_message,

"analysis": analysis\_json.get("analysis", "No analysis available"),

"solution": analysis\_json.get("solution", "No solution available"),

"code\_fix": analysis\_json.get("code\_fix", "")

}

@log\_function\_call

def generate\_function(self, function\_spec: Dict[str, Any]) -> str:

"""

Generates a new function based on a specification.

"""

# Implementation

**Explanation:** Ivan now specializes in error analysis and resolution, using LLMs to diagnose errors and propose solutions.

## **14. Task Results Aggregation**

**Workflow Component:** Task Results Aggregation **File:** results\_manager.py **Class:** ResultsManager **Function:** aggregate\_results()

# In results\_manager.py

class ResultsManager:

def \_\_init\_\_(self, kb):

self.kb = kb

self.results = {}

@log\_function\_call

def store\_result(self, task\_id: str, result: Any):

"""

Stores a task result.

"""

self.results[task\_id] = result

@log\_function\_call

def aggregate\_results(self, task\_ids: List[str] = None) -> Dict[str, Any]:

"""

Aggregates results from multiple tasks.

"""

# If no task IDs provided, aggregate all results

if task\_ids is None:

task\_ids = list(self.results.keys())

# Collect results to aggregate

results\_to\_aggregate = [self.results.get(task\_id) for task\_id in task\_ids if task\_id in self.results]

# Perform aggregation based on result types

aggregated\_result = self.\_aggregate\_by\_type(results\_to\_aggregate)

# Store in knowledge base

self.kb.set\_item("aggregated\_result", aggregated\_result)

return aggregated\_result

def \_aggregate\_by\_type(self, results: List[Any]) -> Any:

"""

Aggregates results based on their types.

"""

# Filter out None results

results = [r for r in results if r is not None]

if not results:

return None

# Check if all results are of the same type

result\_types = set(type(r) for r in results)

if len(result\_types) == 1:

# All results are the same type

result\_type = next(iter(result\_types))

if issubclass(result\_type, str):

return "\n\n".join(results)

elif issubclass(result\_type, (dict, Dict)):

# Merge dictionaries

merged = {}

for result in results:

merged.update(result)

return merged

elif issubclass(result\_type, (list, List)):

# Concatenate lists

merged = []

for result in results:

merged.extend(result)

return merged

# ... handle other types

# Mixed types - convert to strings and join

return "\n\n".join(str(r) for r in results)

**Explanation:** This component intelligently combines results from multiple tasks, handling different data types appropriately.

## **15. Quality Control Analysis**

**Workflow Component:** Quality Control Analysis **File:** quality\_analyzer.py **Class:** QualityControlAnalyzer **Function:** analyze\_quality()

# In quality\_analyzer.py

class QualityControlAnalyzer:

def \_\_init\_\_(self, llm\_service):

self.llm\_service = llm\_service

@log\_function\_call

def analyze\_quality(self, results: Any, expectations: Dict[str, Any] = None) -> Dict[str, Any]:

"""

Analyzes the quality of results against expectations.

"""

# Detect any inconsistencies in the results

issues = self.detect\_inconsistencies(results)

# If expectations provided, check if they're met

if expectations:

meets\_expectations = self.evaluate\_satisfaction\_criteria(results, expectations)

if not meets\_expectations:

issues.append("Results do not meet expectations")

# Generate a quality score (0-100)

quality\_score = 100 - (len(issues) \* 10) # Deduct 10 points per issue

quality\_score = max(0, quality\_score) # Ensure non-negative

# If there are issues, generate a correction plan

correction\_plan = None

if issues:

correction\_plan = self.generate\_correction\_plan(issues, results)

return {

"satisfied": len(issues) == 0,

"quality\_score": quality\_score,

"issues": issues,

"correction\_plan": correction\_plan

}

@log\_function\_call

def detect\_inconsistencies(self, results: Any) -> List[str]:

"""

Detects inconsistencies in the results.

"""

issues = []

# Convert results to string for LLM analysis

results\_str = str(results)

# Use LLM to identify issues

prompt = (

f"Analyze these results for inconsistencies, errors, or quality issues:\n"

f"{results\_str}\n\n"

f"List any issues found, one per line. If no issues, respond with 'No issues found.'"

)

system\_prompt = (

"You are a quality control expert. Critically analyze the results to find "

"inconsistencies, logical errors, incomplete information, or other quality issues."

)

issues\_text = self.llm\_service.generate\_text(prompt, system\_prompt)

# Parse issues

if "No issues found" not in issues\_text:

# Split by lines and filter empty lines

issues = [line.strip() for line in issues\_text.split("\n") if line.strip()]

return issues

@log\_function\_call

def generate\_correction\_plan(self, issues: List[str], results: Any) -> Dict[str, Any]:

"""

Generates a plan to correct identified issues.

"""

# Implementation

**Explanation:** This component analyzes result quality, detecting inconsistencies and generating correction plans, a key part of the feedback loop in your diagram.

## **16. Correction Plan and Additional Tasks**

**Workflow Component:** Correction Plan, Submit Additional Tasks **File:** quality\_analyzer.py **Class:** QualityControlAnalyzer **Function:** generate\_correction\_plan()

# In quality\_analyzer.py (continued)

class QualityControlAnalyzer:

# ... other methods

@log\_function\_call

def generate\_correction\_plan(self, issues: List[str], results: Any) -> Dict[str, Any]:

"""

Generates a plan to correct identified issues.

"""

# Convert results and issues to string for LLM

results\_str = str(results)

issues\_str = "\n".join(f"- {issue}" for issue in issues)

# Use LLM to generate correction plan

prompt = (

f"Generate a correction plan for these results with the following issues:\n\n"

f"Results:\n{results\_str}\n\n"

f"Issues:\n{issues\_str}\n\n"

f"Create a structured correction plan with specific tasks needed to fix these issues."

)

system\_prompt = (

"You are a quality improvement expert. Generate a detailed correction plan "

"that addresses each issue. Format your response as a JSON with 'plan\_summary' "

"and 'correction\_tasks' fields. Each correction task should have 'task\_name', "

"'agent', and 'description' fields."

)

# Get correction plan from LLM

correction\_plan = self.llm\_service.generate\_json(prompt, system\_prompt)

return {

"summary": correction\_plan.get("plan\_summary", "No summary available"),

"tasks": correction\_plan.get("correction\_tasks", [])

}

@log\_function\_call

def create\_correction\_tasks(self, correction\_plan: Dict[str, Any]) -> List[Task]:

"""

Creates executable tasks from a correction plan.

"""

correction\_tasks = []

for task\_spec in correction\_plan.get("tasks", []):

task = Task(

name=task\_spec.get("task\_name", "Correction Task"),

description=task\_spec.get("description", "Fix an issue"),

agent=task\_spec.get("agent", "Nova"),

function\_name=task\_spec.get("function\_name"),

args=task\_spec.get("args", {})

)

correction\_tasks.append(task)

return correction\_tasks

**Explanation:** This component generates specific correction tasks when results don't meet quality standards, feeding back into the task execution system.

## **17. Main Integration**

**Workflow Component:** Main Execution Loop **File:** main.py **Function:** main()

# In main.py

@log\_function\_call

def main():

"""

Main entry point for the enhanced Nova Coordinator system.

"""

# Initialize components

kb = KnowledgeBase()

llm\_service = LLMService()

# Initialize parameter handling

parameter\_analyzer = ParameterAnalyzer()

user\_interaction = UserInteractionManager(llm\_service)

# Initialize task management

task\_creator = TaskListCreator(llm\_service, parameter\_analyzer)

execution\_optimizer = ExecutionOptimizer()

task\_scheduler = TaskPriorityQueue()

# Initialize error handling

error\_handler = ErrorHandler(None) # Will update with agent\_registry

function\_monitor = FunctionMonitor(error\_handler)

# Initialize results handling

results\_manager = ResultsManager(kb)

quality\_analyzer = QualityControlAnalyzer(llm\_service)

# Load function maps from CSV

# ... existing code

# Initialize agents

nova = Nova("Nova", kb, nova\_functions, parameter\_analyzer, user\_interaction)

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

ivan = Ivan("Ivan", kb, ivan\_functions, llm\_service)

lola = Lola("Lola", kb, lola\_functions)

# Create agent registry

agent\_registry = AgentRegistry()

agent\_registry.register\_agent("Nova", nova)

agent\_registry.register\_agent("Emil", emil)

agent\_registry.register\_agent("Ivan", ivan)

agent\_registry.register\_agent("Lola", lola)

# Update error handler with agent registry

error\_handler.agent\_registry = agent\_registry

# Create task executor

task\_executor = TaskExecutor(agent\_registry, kb)

# Capture user prompt

user\_prompt = input("\n\nUser prompt: ")

# Create initial task list

tasks = task\_creator.create\_task\_list\_from\_prompt(user\_prompt)

# Optimize execution

execution\_plan = execution\_optimizer.optimize\_task\_execution(tasks)

# Execute tasks

results = task\_executor.execute\_tasks(execution\_plan)

# Aggregate results

aggregated\_results = results\_manager.aggregate\_results()

# Quality control

quality\_result = quality\_analyzer.analyze\_quality(aggregated\_results)

# Handle correction if needed

if not quality\_result.get("satisfied", True):

print("Quality issues detected. Implementing corrections...")

correction\_plan = quality\_result.get("correction\_plan", {})

correction\_tasks = quality\_analyzer.create\_correction\_tasks(correction\_plan)

# Submit correction tasks

for task in correction\_tasks:

task\_scheduler.add\_task(task)

# Execute correction tasks

while True:

task = task\_scheduler.get\_next\_task()

if not task:

break

task\_executor.execute\_single\_task(task)

# Re-aggregate results

aggregated\_results = results\_manager.aggregate\_results()

# Set final report

kb.set\_item("final\_report", aggregated\_results)

# Display final output

final\_report = kb.get\_item("final\_report")

print("\n===== FINAL OUTPUT =====\n\n")

print(final\_report if final\_report else "No final report created.")

**Explanation:** The main function integrates all components of the enhanced workflow, from initial prompt to quality control and corrections.

## **Key Implementation Highlights**

### **1. Enhanced Agent Specialization**

Each agent is now more specialized:

* **Nova** (nova.py): Central coordinator with improved parameter collection
* **Emil** (emil.py): Energy modeling specialist
* **Ivan** (ivan.py): Code generation and error analysis specialist
* **Lola** (lola.py): Content generation and reporting specialist

### **2. Hierarchical Task Structure**

The Task class now supports:

* Parent-child relationships between tasks
* Dependency tracking
* Status monitoring
* Result storage

### **3. Advanced Parameter Analysis**

Parameter handling is now more sophisticated:

* Extraction from natural language
* Function signature introspection
* Front-loaded collection of all parameters
* LLM-generated conversational questions

### **4. Intelligent Execution Optimization**

The system now includes:

* Task dependency analysis
* Parallel vs. sequential optimization
* Dynamic prioritization
* Adaptive scheduling

### **5. Robust Error Handling**

Error handling is significantly improved:

* Error classification
* Multiple resolution strategies
* Integration with Ivan for complex errors
* Performance monitoring

### **6. Quality Control Feedback Loop**

The system now features:

* Result quality analysis
* Issue detection
* Correction planning
* Automatic submission of correction tasks

This enhanced workflow brings your system much closer to the comprehensive Nova Coordinator diagram, with sophisticated task hierarchy, intelligent optimization, and robust error handling.