The Nova AI Coordinator: Task Delegation System

Table of Contents

1. Overview of Nova's Delegation Architecture

Nova operates as the central coordinator in the multi-agent system, dynamically routing tasks to specialized agents based on their capabilities:

- **Nova**: Coordinates tasks, handles general knowledge, web interactions, math
- **Emil**: Specializes in energy modeling, analysis, and simulation
- **Lola**: Focuses on report generation, copywriting, and communication
- **Ivan**: Manages technical implementations, visualizations, and code generation

The delegation process follows a sophisticated pipeline that transforms natural language prompts into properly routed executable tasks while maintaining conversation context.

2. Prompt Analysis & Intent Detection

When a user submits a prompt, Nova first analyzes it to understand what's being requested. This critical first step determines how tasks will be created and routed.

python

```python
@log_function_call
async def create_task_list_from_prompt_async(self, prompt: str) -> List[Task]:
    """Creates tasks from a user prompt with multi-intent detection"""

    # Get conversation context for LLM detection
    current_conversation = self.kb.get_item("current_conversation")
    previous_prompts = []
    if current_conversation and current_conversation.get("questions"):
        previous_prompts = current_conversation["questions"][-5:]  # Last 5 questions

    # Use LLM to detect query type
    query_type = await self.detect_query_type_with_llm(prompt, previous_prompts)
    is_history_query = query_type["is_history_query"]
    is_follow_up = query_type["is_follow_up"]

    # Special handling for history queries
    if is_history_query:
        print(f"LLM DETECTED: History query: '{prompt}'")
        # Handle history query directly...
        return [history_task]

    # Determine if prompt contains multiple intents
    multiple_intents = await self.identify_multiple_intents_async(prompt)

    print(f"Identified {len(multiple_intents)} intent(s) in the prompt")
```

Nova uses advanced LLM techniques to:

1. **Detect history queries**: Questions about past conversations or session data
2. **Identify follow-up questions**: Queries that reference previous context
3. **Recognize multi-intent prompts**: Complex requests containing multiple actions
4. **Extract conversation entities**: Track countries, cities, or other context elements

The `detect_query_type_with_llm` method uses a carefully crafted prompt to analyze query characteristics:

```python
async def detect_query_type_with_llm(self, prompt, previous_prompts=None):
    """
    Use LLM to detect if a prompt is a history query or a follow-up question.
    """
    # Context for the LLM detection
    context = """
    You are a query analysis assistant tasked with categorizing user questions.
    Analyze the question and determine:

    1. If it's a HISTORY query (asking about previous interactions)
    2. If it's a FOLLOW-UP question (relies on previous context)
    3. If it's a STANDALONE question (makes sense without context)

    For a HISTORY query, respond with: "TYPE: HISTORY"
    For a FOLLOW-UP question, respond with: "TYPE: FOLLOW-UP"
    For a STANDALONE question, respond with: "TYPE: STANDALONE"
    """

    # Create message with context from previous questions if available
    if previous_prompts:
        previous_context = "\n".join([f"Previous Question {i+1}: {p}"
                                      for i, p in enumerate(previous_prompts)])
        message = f"{previous_context}\n\nNew Question: {prompt}\n\nWhat type is this?"
    else:
        message = f"Question: {prompt}\n\nWhat type is this?"

    # Get LLM response and parse result
    response = await run_open_ai_ns_async(message, context)
    response = response.strip().upper()

    return {
        "is_history_query": "TYPE: HISTORY" in response,
        "is_follow_up": "TYPE: FOLLOW-UP" in response,
        "is_standalone": "TYPE: STANDALONE" in response
    }
```

3. Intent Categorization Process

Once intents are identified, Nova categorizes each intent to determine which agent should handle it. This categorization process relies on function maps defined in CSV files and LLM-based classification.

python

```python
@log_function_call
async def open_ai_categorisation_async(question, function_map, level=None):
    """
    Categorize user questions into specific functions using OpenAI.

    Parameters:
        question (str): The user's question or intent
        function_map (str): Path to the function map CSV file
        level (str, optional): Task level ('task list' or None)

    Returns:
        str: The categorized function key
    """
    # Load categories from CSV
    import pandas as pd
    try:
        df = pd.read_csv(function_map)
        categories = dict(zip(df['Key'], df['Description']))
    except Exception as e:
        print(f"Error loading categories: {str(e)}")
        categories = {}

    # Category information for LLM context
    category_info = ", ".join([f"'{key}': {desc}" for key, desc in categories.items()])

    # System message for categorization
    system_msg = (
        f"You are an assistant trained to categorize questions into specific functions. "
        f"Here are the available categories with descriptions: {category_info}. "
        f"If none of the categories are appropriate, categorize as 'general_question'. "
        f"Please respond with only the category from the list given, with no additional text."
    )

    # Call OpenAI API for categorization
    response = await run_open_ai_ns_async(question, system_msg, model="gpt-4.1-nano")
    category = response.strip()

    # Pattern-based overrides for improved accuracy
    if any(keyword in question.lower() for keyword in ["energy model", "build model"]):
        category = "Energy Model"

    return category
```

The categorization system incorporates both LLM-based classification and pattern matching for improved accuracy, with special logic for:

1. **Energy modeling requests**: Keywords like "model", "solar", "energy"
2. **Math expressions**: Pattern matching for arithmetic operations
3. **Report writing**: Keywords like "report", "write up", "summary"
4. **Image generation**: Combining visual and creation terms
5. **Python scripting**: Code-related terminology

4. Agent Selection & Function Mapping

The heart of Nova's delegation process occurs in the `create_task_for_category` method, which maps categorized intents to specific agents and functions:

python

```python
@log_function_call
async def create_task_for_category(self, intent_text: str, category: str) -> Task:
    """
    Creates specialized tasks based on intent category with proper agent routing.

    Parameters:
        intent_text (str): The text of the intent
        category (str): The category of the intent

    Returns:
        Task: The created task with proper agent and function routing
    """
    print(f"Creating task for intent: '{intent_text}' (category: {category})")

    # Text pattern matching - to catch cases where categorization might miss
    intent_lower = intent_text.lower()

    # Default parameters that will be updated based on category
    task_args = {"prompt": intent_text}
    target_agent = "Nova"  # Default agent
    function_name = "answer_general_question"  # Default function

    # ENERGY MODELING - route to Emil
    if category.lower() in ["energy model", "energy modeling", "build model"]:
        target_agent = "Emil"
        function_name = "process_emil_request"

        # Extract model parameters from intent
        from core.functions_registery import extract_model_parameters
        params = extract_model_parameters(intent_text)

        # Convert location parameter if found
        if params.get("locations"):
            task_args["location"] = ",".join(params["locations"])

        # Convert generation parameter if found
        if params.get("generation_types"):
            task_args["generation"] = ",".join(params["generation_types"])

        # Convert energy carrier parameter if found
        if params.get("energy_carriers"):
            task_args["energy_carrier"] = ",".join(params["energy_carriers"])

    # DATA ANALYSIS - route to Emil
    elif category.lower() in ["data analysis", "analyze results"] or is_analysis_task:
        target_agent = "Emil"
        function_name = "analyze_results"

        # Extract analysis type from intent
        analysis_types = {
            "basic": "basic", "detailed": "detailed", "comprehensive": "comprehensive",
            "technical": "technical", "financial": "financial"
        }

        # Default analysis type
        analysis_type = "basic"

        # Try to extract a more specific analysis type
        for keyword, a_type in analysis_types.items():
            if keyword in intent_lower:
                analysis_type = a_type
                break

        # Get model information from knowledge base
        latest_model_file = self.kb.get_item("latest_model_file")
        latest_model_details = self.kb.get_item("latest_model_details")

        # Configure task arguments
        task_args = {
            "prompt": intent_text,
            "analysis_type": analysis_type,
            "model_file": latest_model_file,
            "model_details": latest_model_details
        }

    # REPORT WRITING - route to Lola
    elif category.lower() in ["write report", "report writing"] or is_report_task:
        target_agent = "Lola"
        function_name = "write_report"

        # Extract report style from intent
        report_styles = {
            "executive": "executive_summary", "technical": "technical_report",
            "detailed": "detailed_report", "summary": "executive_summary",
            "presentation": "presentation_report"
        }

        # Default style
        style = "executive_summary"

        # Try to extract a more specific style
        for keyword, report_style in report_styles.items():
            if keyword in intent_lower:
                style = report_style
                break

        # Retrieve model and analysis information for the report
        latest_model_file = self.kb.get_item("latest_model_file")
        latest_model_details = self.kb.get_item("latest_model_details")
        latest_analysis_results = self.kb.get_item("latest_analysis_results")

        # Configure task arguments
        task_args = {
            "prompt": intent_text,
            "style": style,
            "model_file": latest_model_file,
            "model_details": latest_model_details,
            "analysis_results": latest_analysis_results
        }

    # COPYWRITING & PROOFREADING - route to Lola
    elif category.lower() in ["copywriting", "proofreading"]:
        target_agent = "Lola"

        # Determine if this is copywriting or proofreading
        if "proof" in intent_lower or "review" in intent_lower:
            function_name = "proofread"
        else:
            function_name = "copywrite"

    # MATH CALCULATIONS - route to Nova
    elif category.lower() == "do_maths":
        target_agent = "Nova"
        function_name = "do_maths"

    # IMAGE GENERATION - route to Ivan
    elif is_image_request:
        target_agent = "Ivan"
        function_name = "generate_image"

    # WEBSITE OPENING - route to Nova
    elif category.lower() == "open a website":
        target_agent = "Nova"
        function_name = "open_website"

    # PYTHON SCRIPTING - route to Ivan
    elif "python" in intent_lower or "script" in intent_lower:
        target_agent = "Ivan"
        function_name = "generate_python_script"
        task_args["script_name"] = f"script_{int(time.time())}"
```

```python
    # GENERAL QUESTIONS - route to Nova
    else:
        target_agent = "Nova"
        function_name = "answer_general_question"

    # Create and return the task with proper routing
    return Task(
        name=f"Handle Intent: {intent_text[:30]}...",
        description=f"Process intent categorized as {category}",
        agent=target_agent,
        function_name=function_name,
        args=task_args
    )
```

The function mapping logic ensures that:

1. **Each intent is matched to the optimal agent** based on specialized capabilities
2. **Specific functions within each agent** are selected based on the exact requirements
3. **Required parameters are extracted** and formatted correctly for each function
4. **Task objects contain all necessary information** for execution

5. Parameter Extraction & Validation

Task delegation often requires parameter extraction from natural language prompts. Nova implements multiple parameter extraction techniques:

5.1 Energy Model Parameter Extraction

python

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

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

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

    # Extract locations using pattern matching
    found_locations = []
    for loc in LOCATIONS:
        # Look for "in [location]" or "for [location]" patterns
        if f" in {loc.lower()}" in prompt_lower or f" for {loc.lower()}" in prompt_lower:
            found_locations.append(loc)
        # Also check for exact match
        elif loc.lower() in prompt_lower:
            found_locations.append(loc)

    params["locations"] = list(set(found_locations))

    # Extract generation types using keyword patterns
    found_gen_types = []
    for gen in GENERATION_TYPES.keys():
        # Look for patterns like "solar power" or "wind energy"
        if f"{gen} power" in prompt_lower or f"{gen} generation" in prompt_lower:
            found_gen_types.append(gen)
        # Also check for exact match
        elif gen in prompt_lower:
            found_gen_types.append(gen)

    params["generation_types"] = found_gen_types

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

    # Set defaults if nothing found
    if not params["locations"]:
        params["locations"] = ["Unknown"]
    if not params["generation_types"]:
        params["generation_types"] = ["solar"]
    if not params["energy_carriers"]:
        params["energy_carriers"] = ["electricity"]

    return params
```

5.2 Interactive Parameter Collection

When parameters cannot be automatically extracted, Nova can interactively collect them from the user:

python

```python
@log_function_call
async def get_energy_parameters_from_user_async(self, missing_params: List[str]) -> Dict[str, A
    """
    Interactively collects energy modeling parameters from the user.

    Parameters:
        missing_params (List[str]): List of missing parameter names

    Returns:
        Dict[str, Any]: Collected parameters
    """
    collected_args = {}

    # Parameter descriptions for user guidance
    param_descriptions = {
        "location": "The geographic location for the energy model (e.g., UK, France, Spain)",
        "generation": "The generation type for the model (e.g., solar, wind, hydro)",
        "energy_carrier": "The energy carrier to model (e.g., electricity, hydrogen)"
    }

    # Parameter examples for better user experience
    param_examples = {
        "location": "UK, France, Germany, or 'all' for all available locations",
        "generation": "solar, wind, hydro, thermal, bio, or 'all' for all types",
        "energy_carrier": "electricity (default), hydrogen, methane"
    }

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

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

        # Get input asynchronously
        user_response = await asyncio.to_thread(input, "> ")
        collected_args[param] = user_response.strip()

    return collected_args
```

5.3 Parameter Validation

Once tasks are delegated, specialized agents validate parameters before execution. Emil, for example, implements comprehensive parameter verification:

```
python
```

```python
    @log_function_call
    async def verify_parameters_async(self, function_name: str, task_args: dict) -> dict:
        """
        Asynchronous parameter verification with special case handling.

        Parameters:
            function_name (str): The function to verify parameters for
            task_args (dict): The provided task arguments

        Returns:
            dict: Verification results with 'success' flag and 'missing' parameters
        """
        # Special handling for process_emil_request
        if function_name == 'process_emil_request':
            # If a prompt is provided, consider it a valid call
            if task_args.get('prompt'):
                return {
                    "success": True,
                    "missing": [],
                    "message": "Prompt provided for Emil request"
                }

        # Special handling for analyze_results
        if function_name == 'analyze_results':
            # Analysis tasks don't need explicit parameters
            # All data should be retrieved from the knowledge base
            return {
                "success": True,
                "missing": [],
                "message": "Analysis tasks don't require explicit parameters"
            }

        # If function not found in map, return error
        if function_name not in self.function_map:
            return {
                "success": False,
                "missing": [],
                "message": f"Function '{function_name}' not found in Emil's function map"
            }

        # Get the function from the map
        func = self.function_map[function_name]
        sig = inspect.signature(func)

        # Get required parameters (excluding 'self' and 'kb')
        required_params = {
```

```
        param.name for param in sig.parameters.values()
        if param.default == inspect.Parameter.empty
        and param.name != 'self'
        and param.name != 'kb'
    }

    # Check for missing parameters
    missing_params = [arg for arg in required_params if arg not in task_args]

    if missing_params:
        return {
            "success": False,
            "missing": missing_params,
            "message": f"Missing required parameters: {', '.join(missing_params)}"
        }

    return {
        "success": True,
        "missing": [],
        "message": "All required parameters are present"
    }
```

6. Hierarchical Task Construction

Nova can organize tasks into hierarchical structures, particularly for sequential workflows like energy modeling followed by analysis and reporting:

python

```python
# Check if we have a model-analyze-report workflow pattern
has_model_task = any("model" in intent.lower() for intent in intents)
has_analyze_task = any("analysis" in intent.lower() for intent in intents)
has_report_task = any("report" in intent.lower() for intent in intents)

if has_model_task and (has_analyze_task or has_report_task):
    print("DETECTED: Sequential task chain for model → analyze → report")

    # Find the model creation task
    model_intent = next(intent for intent in intents if "model" in intent.lower())
    model_category = await open_ai_categorisation_async(model_intent, csv_path)
    model_task = await self.create_task_for_category(model_intent, model_category)

    # If we have analysis, add it as a subtask
    if has_analyze_task:
        analysis_intent = next(intent for intent in intents if "analysis" in intent.lower())
        analysis_category = await open_ai_categorisation_async(analysis_intent, csv_path)
        analysis_task = await self.create_task_for_category(analysis_intent, analysis_category)

        # Force the analysis task to be assigned to Emil
        analysis_task.agent = "Emil"
        analysis_task.function_name = "analyze_results"

        # Add analysis task as subtask to model task
        model_task.sub_tasks.append(analysis_task)

        # If we have reporting, add it as a subtask of analysis
        if has_report_task:
            report_intent = next(intent for intent in intents if "report" in intent.lower())
            report_category = await open_ai_categorisation_async(report_intent, csv_path)
            report_task = await self.create_task_for_category(report_intent, report_category)

            # Force the report task to be assigned to Lola
            report_task.agent = "Lola"
            report_task.function_name = "write_report"

            # Add report task as subtask to analysis task
            analysis_task.sub_tasks.append(report_task)
```

This hierarchical structuring ensures that:

1. **Dependent tasks are executed in the correct order**
2. **Results from parent tasks are available to subtasks**

3. **Complex workflows are properly coordinated**
4. **Agent communication happens through the Knowledge Base**

7. Execution Flow & Task Sequencing

Once tasks are properly delegated, Nova coordinates their execution with the following process:

```python
async def process_prompt_tasks(prompt_idx, prompt, task_list, agents, kb):
    """
    Process all tasks for a single prompt sequentially.

    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
    """
    print(f"Processing: '{prompt}'")

    # Process each task in sequence
    for i, task in enumerate(task_list, 1):
        print(f"Task {i}/{len(task_list)}: {task.name}")
        print(f"Delegating to {task.agent}")

        # Get the appropriate agent
        agent = agents.get(task.agent)
        if agent:
            try:
                # Execute the task with the correct agent
                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)

                # Check if a task has subtasks
                if task.sub_tasks:
                    print(f"Processing {len(task.sub_tasks)} subtasks")
                    await process_subtasks(task, agents)
            except Exception as e:
                error_msg = f"Error processing task: {str(e)}"
                print(error_msg)
        else:
            print(f"No agent found for name {task.agent}")
```

When tasks have subtasks, they are processed recursively:

python

```python
async def process_subtasks(task, agents):
    """
    Process all subtasks of a task recursively.

    Parameters:
        task (Task): The parent task with subtasks
        agents (dict): Dictionary of agent instances
    """
    for idx, subtask in enumerate(task.sub_tasks, 1):
        print(f"Subtask {idx}/{len(task.sub_tasks)}: {subtask.name}")
        print(f"Delegating to {subtask.agent}")

        agent = agents.get(subtask.agent)
        if agent:
            # Execute the subtask
            result = await agent.handle_task_async(subtask)

            # Process any sub-subtasks recursively
            if subtask.sub_tasks:
                await process_subtasks(subtask, agents)
```

8. Context Management & Feedback Loop

Nova maintains conversation context throughout the delegation process, enabling follow-up questions and contextual awareness:

```python
# Update conversation context with task result
conversation = kb.get_item("current_conversation")
if conversation is None:
    conversation = {"questions": [], "answers": []}

# Store question in conversation history
if "questions" not in conversation:
    conversation["questions"] = []
conversation["questions"].append(prompt)

# Store answer in conversation history
if "answers" not in conversation:
    conversation["answers"] = []
conversation["answers"].append(result)

# Update entity tracking
if task.function_name == "answer_general_question":
    # Extract entities from result (countries, cities, etc.)
    for location in LOCATIONS:
        if location.lower() in result.lower():
            conversation["current_country"] = location
            if "entities" not in conversation:
                conversation["entities"] = {}
            conversation["entities"]["country"] = location

# Save updated conversation context
kb.set_item("current_conversation", conversation)
```

This context management enables:

1. **Entity tracking**: Countries, cities, and other mentioned entities
2. **Pronoun resolution**: Understanding references to previous topics
3. **History access**: Retrieving information from past interactions
4. **Follow-up handling**: Continuing conversations across multiple turns

9. Recovery Strategies & Error Handling

The delegation system implements robust error handling and recovery strategies:

python

```python
try:
    # Try executing the task with the appropriate agent
    result = await agent.handle_task_async(task)
except Exception as e:
    # Log the error
    error_msg = f"Error executing {task.function_name}: {str(e)}"
    print(error_msg)

    # Mark the task as failed
    task.mark_failed(error_msg)

    # Try to recover by accessing knowledge base for partial results
    if task.function_name == "process_emil_request":
        emil_result = await kb.get_item_async("emil_result")
        if emil_result:
            # Use partial result
            result = f"Partial result available: {emil_result.get('message', 'Unknown')}"
        else:
            # Create placeholder response
            result = f"Unable to complete energy modeling task due to error: {str(e)}"
    elif task.function_name == "answer_general_question":
        # Fall back to simpler model for general questions
        result = "I apologize, but I encountered an error answering your question. " + \
                 "Let me try a simplified approach."
        # Attempt simplified approach
        try:
            result = run_open_ai_ns(prompt, "You are a helpful assistant.")
        except:
            result = "I'm sorry, I'm having trouble answering that question right now."
```

Additionally, Nova implements quality evaluation with fallback strategies:

python

```python
# Get evaluation system configuration
eval_config = kb.get_item("evaluation_config") or {}

# Evaluate the quality of the response
if eval_config.get("evaluation_enabled", True):
    evaluation = await evaluate_answer_quality(kb, question, result)

    # Check if the evaluation score passes the threshold
    if not evaluation["passed"]:
        print(f"Answer quality below threshold ({evaluation['score']})")

        # Determine best fallback strategy
        available_alternatives = ["internet_search", "more_detailed_llm"]
        strategy = await determine_fallback_strategy(kb, question, evaluation, available_alterr

        # Execute recommended fallback strategy
        if strategy["recommended_strategy"] == "internet_search":
            print("Using internet search fallback")
            search_results = await internet_search(kb, question)
            improved_result = await generate_improved_answer(kb, question, result, evaluation,
            result = improved_result
        elif strategy["recommended_strategy"] == "more_detailed_llm":
            print("Using more detailed LLM fallback")
            improved_result = await generate_improved_answer(kb, question, result, evaluation)
            result = improved_result
```

These recovery mechanisms ensure that:

1. **Task failures are gracefully handled**
2. **Partial results are utilized when available**
3. **Alternative approaches are attempted when primary methods fail**
4. **Users receive helpful responses even in error conditions**
5. **Quality evaluation identifies and improves poor responses**

Summary

Nova's task delegation system transforms natural language requests into properly routed executable tasks through a sophisticated pipeline:

1. **Prompt Analysis**: Break down complex requests into intents
2. **Intent Categorization**: Map intents to system capabilities

3. **Agent Selection**: Route tasks to specialized agents
4. **Parameter Extraction**: Extract and validate required parameters
5. **Hierarchical Construction**: Build task trees with proper dependencies
6. **Execution Coordination**: Manage task execution sequence
7. **Context Management**: Maintain conversation state and entity tracking
8. **Error Recovery**: Implement fallback strategies when issues occur

This delegation architecture enables Nova to function as an intelligent coordinator that leverages specialized agents to handle complex user requests while maintaining conversation context and recovering from failures.