# PyDAGI v0.1: User Guide and API Reference (Outline)

Sebastian Dumbrava

# Contents

CHAPTER 1

# Introduction to PyDAGI v0.1

CHAPTER 2

# Core Concepts and Classes (v0.1)

## 1. Conceptual Module: `pydagi.agent`

**1.1. BaseAgent (Abstract Base Class).**

**1.2. LLMAgent.**

**1.3. Specialized Agents: CriticAgent, CoderAgent.**

**1.4. AgentManager.**

## 2. Conceptual Module: `pydagi.communication`

**2.1. Message.**

**2.2. CommunicationChannel (Abstract Base Class).**

**2.3. LocalCommunicationChannel.**

**2.4. DistributedCommunicationChannel (Abstract Base Class).**

**2.5. TopologyManager (Placeholder).**

## 3. Conceptual Module: `pydagi.task`

**3.1. Task.**

**3.2. TaskAllocationStrategy (Abstract Base Class).**

**3.3. CapabilityBasedAllocator.**

**3.4. TaskManager.**

## 4. Conceptual Module: `pydagi.iteration`

**4.1. StoppingCondition (Abstract Base Class).**

**4.2. MaxIterationsCondition.**

**4.3. SolutionConvergenceCondition.**

**4.4. IterationController.**

## 5. Conceptual Module: `pydagi.utilities`

**5.1. Exception Classes.**

**5.2. Logging.**

**5.3. load_config(filepath).**

**5.4. Placeholder Utilities.**

## 6. Top-Level Orchestration

**6.1. DAGINetwork.**

CHAPTER 3

# Building a Simple DAGI Network (v0.1 Example)

**1. Create `config.yaml`**

**2. Initialize the DAGINetwork**

**3. Start the Network**

**4. Define and Submit the Task**

**5. Process the Results**

**6. Stop the Network**

CHAPTER 4

# API Reference (v0.1)

## 1. Exceptions

## 2. Core Classes

**2.1.** `DAGINetwork`.

**2.2.** `Task`.

## 3. Agent Classes

**3.1.** `BaseAgent` **(ABC).**

**3.2.** `LLMAgent`.

**3.3.** `CriticAgent`.

**3.4.** `CoderAgent`.

**3.5.** `AgentManager`.

## 4. Communication Classes

**4.1.** `Message`.

**4.2.** `CommunicationChannel` **(ABC).**

**4.3.** `LocalCommunicationChannel`.

**4.4.** `DistributedCommunicationChannel` **(ABC).**

**4.5.** `TopologyManager`.

## 5. Task and Iteration Control Classes

**5.1.** `TaskAllocationStrategy` **(ABC).**

**5.2.** `CapabilityBasedAllocator`.

**5.3.** `TaskManager`.

**5.4.** `StoppingCondition` **(ABC).**

**5.5.** `MaxIterationsCondition`.

**5.6.** `SolutionConvergenceCondition`.

**5.7.** `IterationController`.

## 6. Utility Functions and Classes

**6.1.** `load_config(filepath)`.

**6.2.** `PromptTemplateManager` **(Placeholder).**

**6.3.** `evaluate_task_result(task)` **(Placeholder).**

CHAPTER 5

# Advanced Usage and Customization (v0.1)

CHAPTER 6

# Examples and Use Cases (Conceptual for v0.1)

**1. Collaborative Tagline Generation**

**2. Code Generation and Critique**

**3. Multi-Perspective Summarization**

CHAPTER 7

# Future Directions and Contributing

APPENDIX A

# Full pydagi.py Code Listing (v0.1)

```
    """
PyDAGI: A Python Package for Distributed Artificial General Intelligence

This package provides the core infrastructure for building and experimenting with
DAGI (Distributed Artificial General Intelligence) Networks, as described in
the accompanying research paper: "Distributed Artificial General Intelligence:
Architecture, Implementation, and Challenges" by Sebastian Dumbrava.

Key Principles Reflected:
- Heterogeneity: Agents can have different architectures, training data, and specializations
                                              (Sec 3.2).
- Iterative Refinement: Collaboration through proposal, critique, and revision cycles (Sec 3
                                             .3).
- Dynamic Communication: Support for adaptable communication topologies (Sec 3.4).
- Specialization: Agents can take specific roles (Sec 3.5).
- Emergence: Framework designed to facilitate the emergence of intelligence from agent
                                             interactions.

Core Modules (as described in Sec 8.2):
- pydagi.agent: Defines agent interfaces, implementations, and management.
- pydagi.communication: Handles inter-agent communication channels and messaging.
- pydagi.task: Defines tasks and task allocation strategies.
- pydagi.iteration: Controls the iterative refinement process and stopping conditions.
- pydagi.utilities: Provides utility functions (logging, config, serialization, etc.).

This single file implementation uses comments to delineate the conceptual modules.
"""

import abc       # For abstract base classes
import uuid      # For generating unique IDs
import logging   # For logging
import json      # For data serialization and loading/saving agent state
import asyncio   # For asynchronous operations
import time      # For potential delays or timeouts
import random    # For basic random selections
from datetime import datetime
from typing import Dict, List, Any, Optional, Union, Callable, Tuple, Type

# --- pydagi.utilities: Logging Configuration ---
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(name)s - %(levelname)s - %(
                                             message)s')
logger = logging.getLogger("pydagi")

# --- pydagi.utilities: Error Handling ---
class DAGIError(Exception):
    """Base class for all DAGI-specific exceptions."""
    pass

class AgentError(DAGIError):
    """Base class for agent-related errors."""
    pass

class AgentNotFoundError(AgentError):
    """Raised when an agent with a given ID is not found."""
    pass

class CommunicationError(DAGIError):
    """Raised when an error occurs during inter-agent communication."""
    pass

class TaskError(DAGIError):
    """Base class for task-related errors."""
    pass

class TaskAllocationError(TaskError):
    """Raised when an error occurs during task allocation."""
    pass

class IterationError(DAGIError):
    """Raised when an error occurs during the iterative refinement process."""
    pass

class ConfigurationError(DAGIError):
    """Raised for configuration-related issues."""
    pass

# --- pydagi.communication: Message Structure ---
class Message:
    """
    Standardized data structure for messages exchanged between agents. (Sec 8.2.2)
    """
    def __init__(self,
                 sender_id: str,
                 recipient_id: Union[str, List[str]],
                 message_type: str,
                 content: Dict[str, Any],
                 task_id: Optional[str] = None,
                 metadata: Optional[Dict[str, Any]] = None):
        """
        Initializes a new Message.

        Args:
            sender_id: Unique identifier of the sending agent.
            recipient_id: Unique identifier(s) of the receiving agent(s). '*' for broadcast?
                                                          (Needs protocol definition)
            message_type: Type of message (e.g., 'TASK_PROMPT', 'PROPOSAL', '
                                                         CRITIQUE_REQUEST', 'CRITIQUE', '
                                                         REVISION_REQUEST', 'STATUS_UPDATE
                                                         ').
            content: The message content (dictionary). Structure depends on message_type.
            task_id: Optional ID of the task this message relates to.
            metadata: Optional metadata (dictionary, e.g., {'iteration': 3, '
                                                         anonymize_sender': True}).
        """
        if not isinstance(sender_id, str) or not sender_id:
            raise ValueError("sender_id must be a non-empty string.")
        if not isinstance(recipient_id, (str, list)) or not recipient_id:
             raise ValueError("recipient_id must be a non-empty string or list.")
        if isinstance(recipient_id, list) and not all(isinstance(r_id, str) and r_id for
                                                     r_id in recipient_id):
             raise ValueError("All recipient IDs in a list must be non-empty strings.")
        if not isinstance(message_type, str) or not message_type:
            raise ValueError("message_type must be a non-empty string.")
        if not isinstance(content, dict):
            raise ValueError("content must be a dictionary.")

        self.message_id: str = str(uuid.uuid4())
        self.sender_id: str = sender_id
        self.recipient_id: Union[str, List[str]] = recipient_id
        self.message_type: str = message_type
        self.content: Dict[str, Any] = content
        self.task_id: Optional[str] = task_id
        self.timestamp: datetime = datetime.now()
        self.metadata: Dict[str, Any] = metadata or {}

    def __str__(self):
        return (f"Message(id={self.message_id}, task={self.task_id}, sender={self.sender_id}
                                                                   , "
                f"recipient={self.recipient_id}, type={self.message_type}, "
                f"timestamp={self.timestamp}, metadata={self.metadata})") # Content excluded
                                                                            for brevity

    def to_dict(self) -> Dict[str, Any]:
        """Serializes the message to a dictionary."""
        return {
            "message_id": self.message_id,
            "sender_id": self.sender_id,
            "recipient_id": self.recipient_id,
            "message_type": self.message_type,
            "content": self.content,
            "task_id": self.task_id,
            "timestamp": self.timestamp.isoformat(),
            "metadata": self.metadata,
        }

    @classmethod
    def from_dict(cls, data: Dict[str, Any]) -> "Message":
        """Deserializes the message from a dictionary."""
        msg = cls(
            sender_id=data['sender_id'],
            recipient_id=data['recipient_id'],
            message_type=data['message_type'],
            content=data['content'],
            task_id=data.get('task_id'),
            metadata=data.get('metadata')
        )
        # Restore original ID and timestamp if present
        msg.message_id = data.get('message_id', msg.message_id)
        if 'timestamp' in data:
             try:
                 msg.timestamp = datetime.fromisoformat(data['timestamp'])
             except (ValueError, TypeError):
                  logger.warning(f"Could not parse timestamp '{data['timestamp']}', using
                                                                             current time.")
                  msg.timestamp = datetime.now()
        return msg


# --- pydagi.agent: Base Agent ---
class BaseAgent(abc.ABC):
    """
    Abstract base class for all DAGI agents. (Sec 8.2.1)

    Defines the core interface including processing messages, managing state,
    and declaring capabilities.
    """
    def __init__(self, agent_id: Optional[str] = None, capabilities: Optional[Dict[str, Any]
                                                  ] = None, config: Optional[Dict[str, Any]]
                                                   = None):
        """
        Initializes a new Agent.

        Args:
            agent_id: A unique identifier. If None, a UUID is generated.
            capabilities: Dictionary describing agent's skills/specializations. Structure
                                                     depends on allocation strategy.
            config: Agent-specific configuration parameters.
        """
        self.agent_id: str = agent_id or f"agent_{uuid.uuid4()}"
        self.config: Dict[str, Any] = config or {}
        # Store raw capabilities from config, then validate/process them
        self._raw_capabilities: Dict[str, Any] = capabilities or {}
        self._capabilities: Dict[str, Any] = self._validate_capabilities(self.
                                                     _raw_capabilities)
        self._state: Dict[str, Any] = {"status": "idle"} # Agent's internal state
        self._inbox: asyncio.Queue[Message] = asyncio.Queue()
        self._is_running: bool = False
        self._processing_task: Optional[asyncio.Task] = None
        self._communication_channel: Optional["CommunicationChannel"] = None # Set by
                                                     AgentManager
        self._agent_manager_ref: Optional["AgentManager"] = None # Set by AgentManager

        logger.info(f"Initialized Agent: {self.agent_id} with capabilities: {self.
                                                     _capabilities}")

    def link_infra(self, channel: "CommunicationChannel", manager: "AgentManager"):
        """Connects the agent to communication and management infrastructure."""
        self._communication_channel = channel
        self._agent_manager_ref = manager
        logger.info(f"Agent {self.agent_id} linked to communication channel and agent
                                                     manager.")

    @abc.abstractmethod
    async def process(self, message: Message) -> Optional[Message]:
        """
        Core method to process an incoming message and generate an output message (if any).
                                                     (Sec 8.2.1)
        This encapsulates the agent's primary logic (e.g., generating proposal, critique,
                                                     refining).

        Args:
            message: The incoming Message object.

        Returns:
            An optional Message object representing the agent's response/action,
            or None if no direct message response is generated.
        """
        raise NotImplementedError

    def get_capabilities(self) -> Dict[str, Any]:
        """Returns validated metadata about agent skills/specializations. (Sec 8.2.1)"""
        # Return the processed/validated capabilities
        return self._capabilities

    def _validate_capabilities(self, capabilities: Dict[str, Any]) -> Dict[str, Any]:
        """Placeholder for validating capability structure based on future standards."""
        # Example: ensure 'roles', 'skills', 'task_types' are lists if present
        for key in ['roles', 'skills', 'task_types']:
            if key in capabilities and not isinstance(capabilities[key], list):
                logger.warning(f"Agent {self.agent_id}: Capability '{key}' should be a list,
                                                     converting.")
                try:
                    capabilities[key] = [capabilities[key]]
                except Exception: # Fallback if conversion fails
                     logger.error(f"Agent {self.agent_id}: Failed to convert capability '{
                                                     key}' to list.")
                     capabilities[key] = [] # Or handle error differently
        # Add more validation based on paper or conventions
        return capabilities

    async def save_state(self, filepath: Optional[str] = None) -> Dict[str, Any]:
        """
        Saves the agent's internal state. (Sec 8.2.1 / 8.4)
        Returns the state dictionary and optionally saves to a file.
        """
        state_data = {
            "agent_id": self.agent_id,
            "capabilities": self._raw_capabilities, # Save original capabilities from config
            "internal_state": self._state, # Save agent-specific state
            "config": self.config,
            # Note: Inbox state is generally not saved/restored directly.
        }
        if filepath:
            try:
                with open(filepath, 'w') as f:
                    json.dump(state_data, f, indent=4)
                logger.info(f"Agent {self.agent_id} state saved to {filepath}")
            except IOError as e:
                logger.error(f"Failed to save agent {self.agent_id} state to {filepath}: {e}
                                                            ")
        return state_data

    async def load_state(self, state_data: Optional[Dict[str, Any]] = None, filepath:
                                                    Optional[str] = None):
        """
        Loads the agent's internal state from a dictionary or file. (Sec 8.2.1 / 8.4)
        """
        if filepath:
            try:
                with open(filepath, 'r') as f:
                    state_data = json.load(f)
                logger.info(f"Agent {self.agent_id} state loaded from {filepath}")
            except FileNotFoundError:
                logger.error(f"State file not found for agent {self.agent_id}: {filepath}")
                raise AgentError(f"State file not found: {filepath}")
            except (IOError, json.JSONDecodeError) as e:
                logger.error(f"Failed to load agent {self.agent_id} state from {filepath}: {
                                                            e}")
                raise AgentError(f"Failed to load state from {filepath}: {e}")

        if state_data:
             # Basic validation: Check if agent ID matches if needed (optional)
            # loaded_id = state_data.get('agent_id')
            # if loaded_id and loaded_id != self.agent_id:
            #    logger.warning(f"Loading state for agent {self.agent_id} from data
                                                        belonging to {loaded_id}")

            # Load raw capabilities, then re-validate
            self._raw_capabilities = state_data.get('capabilities', {})
            self._capabilities = self._validate_capabilities(self._raw_capabilities)
            self._state = state_data.get('internal_state', {"status": "idle"})
            self.config = state_data.get('config', {})
            logger.info(f"Agent {self.agent_id} state restored. Current status: {self._state
                                                        .get('status')}")
        else:
             logger.warning(f"No state data provided for agent {self.agent_id} load_state.")


    async def _message_loop(self):
        """Internal loop to process messages from the inbox."""
        logger.info(f"Agent {self.agent_id} message loop started.")
        while self._is_running:
            try:
                message = await self._inbox.get()
                logger.debug(f"Agent {self.agent_id} dequeued message {message.message_id} (
                                                             {message.message_type})")
                self._state["status"] = f"processing_{message.message_type}"
                response_message = await self.process(message)
                self._inbox.task_done()

                if response_message:
                    # Ensure sender_id is correctly set to this agent
                    response_message.sender_id = self.agent_id
                    await self.send_message_via_channel(response_message)

                self._state["status"] = "idle" # Reset status after processing

            except asyncio.CancelledError:
                logger.info(f"Agent {self.agent_id} message loop cancelled.")
                break
            except Exception as e:
                logger.exception(f"Agent {self.agent_id} encountered error in message loop:
                                                             {e}")
                self._state["status"] = "error"
                # Potentially notify the network manager or retry
                await asyncio.sleep(1) # Avoid tight loop on error
        logger.info(f"Agent {self.agent_id} message loop stopped.")

    async def start(self):
        """Starts the agent's message processing loop."""
        if not self._is_running:
            if not self._communication_channel or not self._agent_manager_ref:
                 raise AgentError(f"Agent {self.agent_id} cannot start without linked
                                                              infrastructure (channel/
                                                              manager).")
            self._is_running = True
            self._processing_task = asyncio.create_task(self._message_loop())
            logger.info(f"Agent {self.agent_id} started.")
        else:
            logger.warning(f"Agent {self.agent_id} is already running.")

    async def stop(self):
        """Stops the agent's message processing loop gracefully."""
        if self._is_running and self._processing_task:
            self._is_running = False
            self._processing_task.cancel()
            try:
                await self._processing_task
            except asyncio.CancelledError:
                logger.info(f"Agent {self.agent_id} processing task cancelled successfully."
                                                             )
            self._processing_task = None
            logger.info(f"Agent {self.agent_id} stopped.")
        else:
            # logger.warning(f"Agent {self.agent_id} is not running or already stopped.")
            pass # Avoid warning if stop is called multiple times

    async def receive_message(self, message: Message):
        """Called by the communication channel to deliver a message."""
        if self._is_running:
            await self._inbox.put(message)
            logger.debug(f"Agent {self.agent_id} enqueued message {message.message_id}")
        else:
            logger.warning(f"Agent {self.agent_id} received message while not running.
                                                         Message ignored: {message.
                                                         message_id}")
    async def send_message_via_channel(self, message: Message):
        """Sends a message using the linked communication channel."""
        if not self._communication_channel:
            raise CommunicationError(f"Agent {self.agent_id} has no communication channel
                                                         linked.")
        try:
             # Ensure sender ID is correct
             if message.sender_id != self.agent_id:
                  logger.warning(f"Agent {self.agent_id} sending message with incorrect
                                                               sender_id '{message.
                                                               sender_id}'. Correcting.")
                  message.sender_id = self.agent_id

             await self._communication_channel.send(message)
             logger.debug(f"Agent {self.agent_id} sent message {message.message_id} to {
                                                          message.recipient_id}")
        except CommunicationError as e:
            logger.error(f"Agent {self.agent_id} failed to send message {message.message_id}
                                                         : {e}")
            # Optionally re-raise or handle
            raise

# --- pydagi.agent: LLM Agent Implementation ---
class LLMAgent(BaseAgent):
    """
    Concrete implementation for agents interacting with LLM APIs. (Sec 8.2.1)
    Handles API calls, prompt formatting, basic state/context management.
    """
    def __init__(self, agent_id: Optional[str] = None, capabilities: Optional[Dict[str, Any]
                                                 ] = None, config: Optional[Dict[str, Any]]
                                                  = None):
        """
        Initializes the LLMAgent.

        Args:
            agent_id: Unique ID.
            capabilities: Agent capabilities.
            config: Configuration dictionary, expected to contain 'model' (e.g., 'gpt-4'),
                    'api_key' (optional, loaded from env otherwise), 'temperature', etc.,
                    and potentially 'provider' (e.g., 'openai', 'anthropic', 'google').
        """
        super().__init__(agent_id, capabilities, config)

        self.model = self.config.get('model')
        if not self.model:
            raise ConfigurationError(f"Agent {self.agent_id}: LLMAgent config must include '
                                                         model'.")

        # Add 'model_family' to capabilities if not present, using config['provider'] if
                                                     available
        if 'model_family' not in self._capabilities:
             provider_hint = self.config.get('provider')
             if provider_hint:
                 self._capabilities['model_family'] = provider_hint.lower()
             elif 'gpt' in self.model.lower(): self._capabilities['model_family'] = 'openai'
             elif 'claude' in self.model.lower(): self._capabilities['model_family'] = '
                                                          anthropic'
             elif 'gemini' in self.model.lower(): self._capabilities['model_family'] = '
                                                          google'
             else: self._capabilities['model_family'] = 'unknown'

        # Initialize internal state for LLM context
        if "conversation_history" not in self._state:
            self._state["conversation_history"] = []
        if "current_task_data" not in self._state:
            self._state["current_task_data"] = None

        # Placeholder for actual LLM client initialization (would happen here)
        self._llm_client = self._initialize_llm_client()
        logger.info(f"LLMAgent {self.agent_id} initialized for model {self.model}")

    def _initialize_llm_client(self) -> Any:
        """Placeholder for initializing the actual LLM client based on config."""
        provider = self.config.get('provider', self._capabilities.get('model_family', '
                                                    unknown'))
        api_key = self.config.get('api_key') # Load from env if needed
        logger.info(f"Initializing mock LLM client for provider '{provider}' and model '{
                                                    self.model}'")
        # Example (replace with actual client libraries like openai, anthropic, etc.):
        # if provider == 'openai':
        #     import openai
        #     # Ensure API key is loaded securely, e.g., from environment variables
        #     # client = openai.OpenAI(api_key=os.getenv("OPENAI_API_KEY"))
        #     # return client
        # elif provider == 'anthropic':
        #     import anthropic
        #     # client = anthropic.Anthropic(api_key=os.getenv("ANTHROPIC_API_KEY"))
        #     # return client
        # ...
        return {"provider": provider, "model": self.model} # Mock client

    async def _call_llm_api(self, prompt_messages: List[Dict[str, str]], **kwargs) -> str:
        """Placeholder for making the actual LLM API call."""
        logger.info(f"Agent {self.agent_id} calling mock LLM API for model {self.model} with
                                                     {len(prompt_messages)} messages.")
        # Simulate API call delay
        await asyncio.sleep(random.uniform(0.2, 0.8)) # Shorter delay for testing

        # --- Replace with actual API call using self._llm_client ---
        # Example using OpenAI format:
        # try:
        #     response = self._llm_client.chat.completions.create(
        #         model=self.model,
        #         messages=prompt_messages,
        #         temperature=self.config.get('temperature', 0.7),
        #         **kwargs
        #     )
        #     return response.choices[0].message.content
        # except Exception as api_error:
        #     logger.error(f"Agent {self.agent_id} LLM API call failed: {api_error}")
        #     raise CommunicationError(f"LLM API call failed: {api_error}") from api_error
        # -----------------------------------------------------------

        # Mock response based on prompt content
        last_user_message = next((m['content'] for m in reversed(prompt_messages) if m['role
                                                    '] == 'user'), "No user message found"
                                                    )
        mock_response_text = f"Mock response from {self.agent_id} ({self.model}) regarding:
                                                    '{last_user_message[:50]}...'"
        if isinstance(last_user_message, str): # Basic check
            if "critique" in last_user_message.lower():
                 mock_response_text = f"Mock critique from {self.agent_id}: The proposal
                                                             could improve on clarity
                                                             regarding point A and
                                                             feasibility of point B."
            elif "refine" in last_user_message.lower() or "revise" in last_user_message.
                                                        lower():
```

```
                mock_response_text = f"Mock refined proposal from {self.agent_id}: Adjusted
                                                              point A for clarity and added
                                                              implementation details for
                                                              point B."
            elif "names" in last_user_message.lower():
                 mock_response_text = f"Mock names from {self.agent_id}: CollaborAI,
                                                              SynthoMind, AgentWeave"

        logger.info(f"Agent {self.agent_id} received mock LLM response.")
        return mock_response_text

    async def _prepare_llm_prompt(self, task_prompt: str, history_context: List[Dict[str,
                                                  Any]], role_directive: Optional[str] =
                                                  None) -> List[Dict[str, str]]:
        """Constructs the message list for the LLM API call."""
        messages = []
        # Default system prompt emphasizing role and capabilities
        base_directive = f"You are AI Agent {self.agent_id}."
        caps_string = ", ".join([f"{k}: {v}" for k, v in self.get_capabilities().items()])
        if caps_string:
            base_directive += f" Your capabilities/roles include: {caps_string}."
        base_directive += " Respond according to your role and the task."

        if role_directive:
            # Combine base directive with specific role directive
             messages.append({"role": "system", "content": f"{base_directive}\nSpecifically
                                                          for this request: {role_directive
                                                          }"})
        else:
             messages.append({"role": "system", "content": base_directive})

        # Add history (needs sophisticated context management - Sec 3.6)
        # Simple approach: add last N interactions from state
        history_to_add = self._state.get("conversation_history", [])
        max_history = self.config.get("context_history_limit", 5) # Limit history length
        messages.extend(history_to_add[-max_history:])

        # Add the current task prompt as a user message
        messages.append({"role": "user", "content": task_prompt})
        return messages

    async def process(self, message: Message) -> Optional[Message]:
        """Processes incoming messages using the LLM. (Sec 8.2.1)"""
        logger.debug(f"LLMAgent {self.agent_id} processing message type: {message.
                                                    message_type}")
        response_content = None
        response_type = None
        # Default: respond to controller specified in metadata, fallback to sender
        recipient_id = message.metadata.get("reply_to", message.sender_id)

        try:
            current_history = self._state.get("conversation_history", [])

            # Store or update task data based on incoming message
            if message.task_id: # Associate state with task ID if available
                 if "task_context" not in self._state: self._state["task_context"] = {}
                 if message.task_id not in self._state["task_context"]:
                      self._state["task_context"][message.task_id] = {"description": None, "
                                                                   history": [], "
                                                                   received_critiques": []}

                 task_ctx = self._state["task_context"][message.task_id]
                 if message.message_type == 'TASK_PROMPT':
                      task_ctx["description"] = message.content.get("description")
                task_ctx["history"].append({"role": "user", "content": task_ctx["
                                                        description"]})
                current_history = task_ctx["history"] # Use task-specific history

            elif message.message_type == 'CRITIQUE':
                critique_content = message.content.get('critique', 'No critique content
                                                       .')
                logger.info(f"Agent {self.agent_id} received critique for task {message
                                                       .task_id}: {
                                                       critique_content[:100]}
                                                       ...")
                task_ctx["received_critiques"].append(critique_content)
                # Add to task history as system message for context in revision
                task_ctx["history"].append({"role": "system", "content": f"Critique
                                                       Received: {
                                                       critique_content}"})
                current_history = task_ctx["history"]
                return None # No direct response needed

        else: # Use global history if no task ID
            current_history = self._state.get("conversation_history", [])


        # --- Message Type Handling ---
        if message.message_type == 'TASK_PROMPT':
            task_desc = message.content.get("description", "")
            role_directive = f"Generate a proposal based on the following task: {
                                                   task_desc}"
            llm_prompt_messages = await self._prepare_llm_prompt(task_desc,
                                                   current_history[:-1],
                                                   role_directive) # Exclude
                                                   current prompt
            llm_response = await self._call_llm_api(llm_prompt_messages)
            response_content = {"proposal": llm_response, "original_task": message.
                                                   content}
            response_type = "PROPOSAL"
            # Update history
            if message.task_id and task_ctx: task_ctx["history"].append({"role": "
                                                   assistant", "content":
                                                   llm_response})
            else: self._state["conversation_history"].append({"role": "assistant", "
                                                   content": llm_response})


        elif message.message_type == 'CRITIQUE_REQUEST':
            proposals_to_critique = message.content.get("proposals", [])
            task_desc = message.content.get('original_task_description', 'N/A') # Get
                                                   task desc from message

            critique_prompt = f"Critique the following proposals based on the original
                                                   task requirements: {task_desc
                                                   }\n\n"
            valid_proposals_found = False
            display_ids_critiqued = []
            for i, prop_data in enumerate(proposals_to_critique):
                display_id = prop_data.get('display_id', f'proposal_{i+1}')
                proposal_text = prop_data.get('proposal')
                if proposal_text:
                    critique_prompt += f"--- Proposal {display_id} ---\n{proposal_text}
                                                           \n\n"
                    valid_proposals_found = True
                    display_ids_critiqued.append(display_id)

            if not valid_proposals_found:
                logger.warning(f"Agent {self.agent_id} received CRITIQUE_REQUEST for
                                                    task {message.task_id}
                                                    but no valid proposals.")
                return None

            role_directive = f"You are acting as a Critic. Provide constructive
                                                critique focused on flaws,
                                                improvements, and alignment
                                                with requirements."
            # Use current history for context, LLM should understand it's critiquing
                                                others
            llm_prompt_messages = await self._prepare_llm_prompt(critique_prompt,
                                                current_history,
                                                role_directive)
            llm_response = await self._call_llm_api(llm_prompt_messages)

            response_content = {"critique": llm_response, "
                                                critiqued_proposal_display_ids
                                                ": display_ids_critiqued}
            response_type = "CRITIQUE"
            # Do not add critique of others to agent's primary task history


        elif message.message_type == 'REVISION_REQUEST':
            critiques = message.content.get("critiques", [])
            last_proposal = message.content.get("last_proposal")
            task_desc = message.content.get("original_task_description", 'N/A')

            if not last_proposal:
                 logger.error(f"Agent {self.agent_id} received REVISION_REQUEST for
                                                    task {message.task_id}
                                                    without 'last_proposal'
                                                    in content.")
                 return None # Cannot revise without knowing what to revise

            revision_prompt = f"Revise your previous proposal based on the following
                                                critiques. \nOriginal Task: {
                                                task_desc}\nYour Last
                                                Proposal:\n{last_proposal}\n\
                                                nCritiques Received:\n"
            if critiques:
                for i, crit in enumerate(critiques):
                    revision_prompt += f"- {crit}\n"
            else:
                 revision_prompt += "- No specific critiques provided. Review and
                                                    improve your proposal
                                                    based on the original
                                                    task.\n"

            role_directive = f"Revise your proposal based on the provided critiques to
                                                better meet the original task
                                                 requirements."
            # Prepare prompt using relevant history and the revision request
            llm_prompt_messages = await self._prepare_llm_prompt(revision_prompt,
                                                current_history,
                                                role_directive)
            llm_response = await self._call_llm_api(llm_prompt_messages)

            response_content = {"revised_proposal": llm_response, "based_on_critiques":
                                                 critiques}
            response_type = "PROPOSAL" # Revised proposal is still a proposal
            # Update history
            if message.task_id and task_ctx: task_ctx["history"].append({"role": "
                                                assistant", "content":
                                                llm_response})
                else: self._state["conversation_history"].append({"role": "assistant", "
                                                               content": llm_response})

            # CRITIQUE message handling moved to the start of the process method

            else:
                logger.warning(f"LLMAgent {self.agent_id} received unhandled message type: {
                                                              message.message_type}")
                return None

            # --- Construct Response Message ---
            if response_content and response_type and recipient_id:
                response_message = Message(
                    sender_id=self.agent_id,
                    recipient_id=recipient_id,
                    message_type=response_type,
                    content=response_content,
                    task_id=message.task_id,
                    metadata={"responding_to_message_id": message.message_id, "iteration":
                                                                   message.metadata.get("
                                                                   iteration")}
                )
                return response_message
            else:
                return None # No message to send back

        except Exception as e:
             logger.exception(f"LLMAgent {self.agent_id} failed processing message {message.
                                                           message_id} ({message.
                                                           message_type}): {e}")
             error_content = {"error": str(e), "original_message_id": message.message_id}
             return Message(
                 sender_id=self.agent_id,
                 recipient_id=recipient_id, # Send error back to controller/original sender
                 message_type="PROCESSING_ERROR",
                 content=error_content,
                 task_id=message.task_id,
                 metadata=message.metadata
             )

    def _get_last_proposal(self) -> Optional[str]:
        """Helper to retrieve the last proposal/revision made by this agent from history."""
        # This needs to be task-specific if tasks are handled concurrently
        # Using global history for simplicity now
        history = self._state.get("conversation_history", [])
        for msg in reversed(history):
            if msg.get("role") == "assistant":
                 # Assuming assistant role implies proposal/revision
                 return msg.get("content")
        return None


# --- pydagi.agent: Specialized Agent Examples ---
# FIX APPLIED: Added 'capabilities' argument to __init__ signatures
class CriticAgent(LLMAgent):
    """Specialized agent focused on providing critiques. (Sec 8.2.1 / 3.5)"""
    def __init__(self, agent_id: Optional[str] = None, capabilities: Optional[Dict[str, Any]
                                                ] = None, config: Optional[Dict[str, Any]]
                                                 = None):
        # Define the core capabilities for this specialized agent
        core_capabilities = {"task_types": ["critique"], "roles": ["critic"]}
        # Merge with capabilities from config (config overrides/augments core)
        merged_capabilities = core_capabilities.copy()
        if capabilities:
            # Ensure lists are extended, not overwritten, if applicable
            for key, value in capabilities.items():
                if isinstance(value, list) and isinstance(merged_capabilities.get(key),
                                                           list):
                    merged_capabilities[key].extend(item for item in value if item not in
                                                                   merged_capabilities[key]
                                                                   )
                else:
                    merged_capabilities[key] = value

        base_config = config or {}
        # Could use a model known for reasoning or specific critique fine-tuning
        if 'model' not in base_config:
             base_config['model'] = 'claude-3-haiku-20240307' # Example default for critique

        # Pass the MERGED capabilities and config to the superclass (LLMAgent)
        super().__init__(agent_id=agent_id, capabilities=merged_capabilities, config=
                                                  base_config)
        logger.info(f"Initialized CriticAgent: {self.agent_id} with capabilities {self.
                                                  _capabilities}")


class CoderAgent(LLMAgent):
    """Specialized agent focused on code generation. (Sec 8.2.1 / 3.5)"""
    def __init__(self, agent_id: Optional[str] = None, capabilities: Optional[Dict[str, Any]
                                                  ] = None, config: Optional[Dict[str, Any]]
                                                   = None):
        # Define core capabilities
        core_capabilities = {"task_types": ["code_generation", "debugging"], "skills": ["
                                                  python", "javascript"]}
        # Merge with capabilities from config
        merged_capabilities = core_capabilities.copy()
        if capabilities:
             # Ensure lists are extended, not overwritten, if applicable
            for key, value in capabilities.items():
                if isinstance(value, list) and isinstance(merged_capabilities.get(key),
                                                           list):
                    merged_capabilities[key].extend(item for item in value if item not in
                                                                   merged_capabilities[key]
                                                                   )
                else:
                    merged_capabilities[key] = value

        base_config = config or {}
        # Use a model known for coding
        if 'model' not in base_config:
             base_config['model'] = 'gpt-4-turbo-preview' # Example default for coding

        # Pass MERGED capabilities and config to superclass
        super().__init__(agent_id=agent_id, capabilities=merged_capabilities, config=
                                                  base_config)
        logger.info(f"Initialized CoderAgent: {self.agent_id} with capabilities {self.
                                                  _capabilities}")


# --- pydagi.agent: Agent Manager ---
class AgentManager:
    """
    Manages the lifecycle and discovery of agents in the network. (Sec 8.2.1)
    """
    def __init__(self, communication_channel: "CommunicationChannel"):
        self._agents: Dict[str, BaseAgent] = {}
        self._communication_channel = communication_channel
        self.task_manager: Optional["TaskManager"] = None
        logger.info("AgentManager initialized.")
    def link_task_manager(self, task_manager: "TaskManager"):
        """Links the Task Manager."""
        self.task_manager = task_manager

    def add_agent(self, agent: BaseAgent):
        """Registers an agent with the manager and links infrastructure."""
        if agent.agent_id in self._agents:
            raise AgentError(f"Agent with ID {agent.agent_id} already registered.")
        self._agents[agent.agent_id] = agent
        agent.link_infra(self._communication_channel, self)
        if hasattr(self._communication_channel, 'register_agent'):
             self._communication_channel.register_agent(agent.agent_id)
        logger.info(f"Agent {agent.agent_id} added to AgentManager.")

    def remove_agent(self, agent_id: str):
        """Removes an agent from the manager."""
        if agent_id in self._agents:
            agent = self._agents[agent_id]
            # Consider agent.stop() - network stop should handle this
            del self._agents[agent_id]
            if hasattr(self._communication_channel, 'unregister_agent'):
                 self._communication_channel.unregister_agent(agent_id)
            logger.info(f"Agent {agent_id} removed from AgentManager.")
        else:
            raise AgentNotFoundError(f"Cannot remove: Agent with ID {agent_id} not found.")

    def get_agent_by_id(self, agent_id: str) -> Optional[BaseAgent]:
        """Retrieves an agent by its ID."""
        agent = self._agents.get(agent_id)
        # Reduced log noise: only log if not found and actually used somewhere problematic
        # if not agent:
        #      logger.warning(f"Agent with ID {agent_id} not found in AgentManager.")
        return agent

    def list_agent_ids(self) -> List[str]:
        """Returns a list of all registered agent IDs."""
        return list(self._agents.keys())

    def get_all_agents(self) -> List[BaseAgent]:
        """Returns a list of all registered agent instances."""
        return list(self._agents.values())

    def get_agents_with_capability(self, capability_key: str, capability_value: Any) -> List
                                                  [BaseAgent]:
        """Finds agents possessing a specific capability, handling list checks."""
        matching_agents = []
        for agent in self._agents.values():
            agent_caps = agent.get_capabilities()
            if capability_key in agent_caps:
                agent_value = agent_caps[capability_key]
                # Check if required value is met by agent's value
                if isinstance(req_value := capability_value, list):
                    # If requirement is a list (e.g., roles: [critic, reviewer])
                    agent_list = agent_value if isinstance(agent_value, list) else [
                                                             agent_value]
                    if any(item in agent_list for item in req_value):
                        matching_agents.append(agent)
                elif isinstance(agent_value, list):
                    # If agent capability is a list (e.g., skills: [python, java])
                    if req_value in agent_value:
                        matching_agents.append(agent)
                elif agent_value == req_value:
                    # Simple equality check
                    matching_agents.append(agent)
        logger.debug(f"Found {len(matching_agents)} agents for capability query: {
                                                            capability_key}={capability_value}")
        return matching_agents

    async def start_all_agents(self):
        """Starts all registered agents."""
        logger.info("Starting all agents managed by AgentManager...")
        start_tasks = [asyncio.create_task(agent.start()) for agent in self._agents.values()
                                                            ]
        await asyncio.gather(*start_tasks)
        logger.info("All agents started.")

    async def stop_all_agents(self):
        """Stops all registered agents."""
        logger.info("Stopping all agents managed by AgentManager...")
        stop_tasks = [asyncio.create_task(agent.stop()) for agent in self._agents.values()]
        if stop_tasks:
             await asyncio.gather(*stop_tasks)
        logger.info("All agents stopped.")

# --- pydagi.communication: Communication Channels ---
class CommunicationChannel(abc.ABC):
    """Abstract base class for communication backends. (Sec 8.2.2)"""

    @abc.abstractmethod
    async def setup(self):
        """Initialize communication resources."""
        raise NotImplementedError

    @abc.abstractmethod
    async def teardown(self):
        """Close communication resources."""
        raise NotImplementedError

    @abc.abstractmethod
    async def send(self, message: Message):
        """Sends a message."""
        raise NotImplementedError

    @abc.abstractmethod
    async def receive(self, agent_id: str, timeout: Optional[float] = None) -> Optional[
                                                        Message]:
        """
        Receives a message for a specific agent. (Sec 8.2.2)
        NOTE: Primarily for external interaction. Agents use internal _message_loop.
        """
        raise NotImplementedError

    @abc.abstractmethod
    def register_agent(self, agent_id: str):
        """Informs the channel about an agent that will receive messages."""
        raise NotImplementedError

    @abc.abstractmethod
    def unregister_agent(self, agent_id: str):
        """Informs the channel an agent is leaving."""
        raise NotImplementedError


class LocalCommunicationChannel(CommunicationChannel):
    """
    In-memory communication for single-process testing. (Sec 8.2.2)
    Relies on AgentManager to find agents and deliver messages to their inboxes.
    """
    def __init__(self):
        self._agent_manager_ref: Optional[AgentManager] = None
        logger.info("LocalCommunicationChannel initialized.")

    def link_manager(self, agent_manager: AgentManager):
         """Links the Agent Manager needed to find recipient agents."""
         self._agent_manager_ref = agent_manager

    async def setup(self):
        logger.info("LocalCommunicationChannel setup complete.")

    async def teardown(self):
        logger.info("LocalCommunicationChannel teardown complete.")

    def register_agent(self, agent_id: str):
        """Ensures the agent exists in the linked manager."""
        if not self._agent_manager_ref or not self._agent_manager_ref.get_agent_by_id(
                                                    agent_id):
             logger.warning(f"LocalChannel registered agent {agent_id} not (yet) in manager.
                                                         ")
        else:
             logger.debug(f"LocalChannel notified of registered agent {agent_id}.")


    def unregister_agent(self, agent_id: str):
        """No queue cleanup needed for local channel."""
        logger.debug(f"LocalCommunicationChannel notified of agent {agent_id} removal.")

    async def send(self, message: Message):
        """Delivers the message directly to the recipient agent's inbox via AgentManager."""
        if not self._agent_manager_ref:
             raise CommunicationError("LocalCommunicationChannel requires a linked
                                                         AgentManager to send messages.")

        recipient_ids = message.recipient_id
        if isinstance(recipient_ids, str):
            recipient_ids = [recipient_ids] # Handle single recipient ID

        delivered_to = []
        not_found_ids = []
        for r_id in recipient_ids:
            # Basic broadcast placeholder - send to all known agents except sender
            if r_id == '*':
                 all_other_agents = [aid for aid in self._agent_manager_ref.list_agent_ids()
                                                               if aid != message.sender_id]
                 # Recursively call send for each agent (or handle broadcast more
                                                               efficiently)
                 if all_other_agents:
                     broadcast_msg = Message(**message.to_dict()) # Create copy
                     broadcast_msg.recipient_id = all_other_agents
                     await self.send(broadcast_msg) # Send to the list
                 return # Exit after handling broadcast

            recipient_agent = self._agent_manager_ref.get_agent_by_id(r_id)
            if recipient_agent:
                try:
                    await recipient_agent.receive_message(message)
                    delivered_to.append(r_id)
                except Exception as e:
                     logger.error(f"Error delivering message {message.message_id} to agent {
                                                                r_id}'s inbox: {e}")
                     # Log and continue trying other recipients
            else:
                not_found_ids.append(r_id)

        if not_found_ids:
            logger.warning(f"Recipient agent(s) not found for message {message.message_id}:
                                                            {', '.join(not_found_ids)}")

        if not delivered_to and recipient_ids and recipient_ids != ['*']:
             raise AgentNotFoundError(f"Failed to deliver message {message.message_id}.
                                                             Recipient(s) not found: {', '.
                                                             join(not_found_ids)}")
        elif delivered_to:
             logger.debug(f"Message {message.message_id} delivered via LocalChannel to: {',
                                                             '.join(delivered_to)}")


    async def receive(self, agent_id: str, timeout: Optional[float] = None) -> Optional[
                                                   Message]:
        """External polling - Use with caution, agent's internal loop is preferred."""
        if not self._agent_manager_ref:
             raise CommunicationError("LocalCommunicationChannel needs AgentManager for
                                                             external receive.")

        agent = self._agent_manager_ref.get_agent_by_id(agent_id)
        if not agent or not hasattr(agent, '_inbox'):
            logger.error(f"Cannot externally receive for agent {agent_id}: Not found or no
                                                            inbox.")
            return None

        try:
            message = await asyncio.wait_for(agent._inbox.get(), timeout=timeout)
            # DO NOT call task_done here.
            return message
        except asyncio.TimeoutError:
            return None
        except Exception as e:
            logger.exception(f"Error externally receiving message for agent {agent_id}: {e}"
                                                            )
            return None


class DistributedCommunicationChannel(CommunicationChannel, abc.ABC):
    """ABC for channels using external message brokers (RabbitMQ, Redis, etc.). (Sec 8.2.2)
                                                   """
    def __init__(self, config: Dict[str, Any]):
        self.config = config
        logger.info(f"Initializing DistributedCommunicationChannel with config: {config}")

    @abc.abstractmethod
    async def setup(self): raise NotImplementedError
    @abc.abstractmethod
    async def teardown(self): raise NotImplementedError
    @abc.abstractmethod
    async def send(self, message: Message): raise NotImplementedError # Needs serialization
    @abc.abstractmethod
    async def receive(self, agent_id: str, timeout: Optional[float] = None) -> Optional[
                                                   Message]: raise NotImplementedError #
                                                   Needs deserialization
    @abc.abstractmethod
    def register_agent(self, agent_id: str): raise NotImplementedError
    @abc.abstractmethod
    def unregister_agent(self, agent_id: str): raise NotImplementedError


# --- pydagi.communication: Topology Manager (Placeholder) ---
class TopologyManager:
    """Manages connections and routing rules (placeholder). (Sec 8.2.2)"""
    def __init__(self, config: Optional[Dict[str, Any]] = None):
        self.config = config or {}
        self.topology_type = self.config.get("type", "fully_connected") # Default to fully
                                                                 connected
        self.connections = {}
        logger.info(f"TopologyManager initialized (type: {self.topology_type}).")
        if self.topology_type == 'static':
             self.connections = self.config.get('connections', {})

    def update_topology(self, new_connections: Dict):
        if self.topology_type == 'dynamic':
             self.connections = new_connections
             logger.info(f"Dynamic topology updated.")
        else:
             logger.warning("Attempted to update topology for non-dynamic manager.")

    def can_communicate(self, sender_id: str, recipient_id: str) -> bool:
        """Checks if direct communication is allowed based on the topology."""
        if self.topology_type == 'fully_connected':
             return True
        elif self.topology_type == 'static':
             allowed_recipients = self.connections.get(sender_id, [])
             return recipient_id in allowed_recipients
        elif self.topology_type == 'dynamic':
             logger.warning("Dynamic topology communication check not fully implemented.")
             return True # Assume allowed for now
        else:
            return True # Default allow

# --- pydagi.task: Task Definition ---
class Task:
    """Represents a task processed by the DAGI network. (Sec 8.2.3)"""
    def __init__(self,
                 description: str,
                 task_id: Optional[str] = None,
                 input_data: Optional[Dict[str, Any]] = None,
                 requirements: Optional[Dict[str, Any]] = None,
                 evaluation_criteria: Optional[Dict[str, Any]] = None):
        self.task_id: str = task_id or f"task_{uuid.uuid4()}"
        self.description: str = description
        self.input_data: Dict[str, Any] = input_data or {}
        self.requirements: Dict[str, Any] = requirements or {}
        self.evaluation_criteria: Dict[str, Any] = evaluation_criteria or {}
        self.status: str = "pending"
        self.results: Optional[Any] = None
        self.history: List[Dict[str, Any]] = []
        self.created_at: datetime = datetime.now()
        self.completed_at: Optional[datetime] = None
        self.error_info: Optional[str] = None

    def update_status(self, status: str, error: Optional[str] = None):
        self.status = status
        if error:
            self.error_info = error
        logger.info(f"Task {self.task_id} status updated to: {status}{' (' + error + ')' if
                                                                 error else ''}")
        if status in ["completed", "failed", "converged", "failed_timeout"]:
             self.completed_at = datetime.now()

    def set_results(self, results: Any, final_history: List[Dict[str, Any]]):
         self.results = results
         self.history = final_history # Store final history snapshot
         logger.info(f"Task {self.task_id} results recorded.")


# --- pydagi.task: Task Allocation ---
class TaskAllocationStrategy(abc.ABC):
    """ABC for task allocation strategies. (Sec 8.2.3)"""
    @abc.abstractmethod
    async def allocate(self, task: Task, agents: List[BaseAgent]) -> Tuple[List[BaseAgent],
                                                        List[BaseAgent]]:
        """Selects proposing and critiquing agents."""
        raise NotImplementedError


class CapabilityBasedAllocator(TaskAllocationStrategy):
    """Allocates based on agent capabilities matching task requirements. (Sec 8.2.3)"""
    async def allocate(self, task: Task, agents: List[BaseAgent]) -> Tuple[List[BaseAgent],
                                                        List[BaseAgent]]:
        proposing_agents = []
        critiquing_agents = []
        proposer_reqs = task.requirements.get('proposer_capabilities', {})
        critic_reqs = task.requirements.get('critic_capabilities', {"roles": ["critic"]})

        logger.debug(f"Allocating task {task.task_id} based on proposer_reqs: {proposer_reqs
                                                            }, critic_reqs: {critic_reqs}")

        for agent in agents:
            agent_caps = agent.get_capabilities()
            if self._matches_requirements(agent_caps, proposer_reqs):
                proposing_agents.append(agent)
            if self._matches_requirements(agent_caps, critic_reqs):
                critiquing_agents.append(agent)

        if not proposing_agents:
            raise TaskAllocationError(f"No suitable proposing agents found for task {task.
                                                                task_id} requirements: {
                                                                proposer_reqs}")

        selected_proposers = proposing_agents
        proposer_ids = {p.agent_id for p in selected_proposers}
        selected_critics = [c for c in critiquing_agents if c.agent_id not in proposer_ids]

        if not selected_critics and critiquing_agents:
             logger.warning(f"Task {task.task_id}: Critics overlap completely with proposers
                                                                 . Allowing overlap.")
             selected_critics = critiquing_agents
        elif not selected_critics:
             logger.info(f"Task {task.task_id}: No suitable non-overlapping critics found/
                                                                 required.")

        logger.info(f"Task {task.task_id} allocated proposers: {[a.agent_id for a in
                                                            selected_proposers]}, critics: {[a.
                                                            agent_id for a in selected_critics]}")
        return selected_proposers, selected_critics

    def _matches_requirements(self, agent_capabilities: Dict[str, Any],
                                                        required_capabilities: Dict[str, Any]) ->
                                                        bool:
        """Checks if agent capabilities meet task requirements."""
        if not required_capabilities: return True
        for key, req_value in required_capabilities.items():
            agent_value = agent_capabilities.get(key)
            if agent_value is None: return False
            if isinstance(req_value, list):
                agent_list = agent_value if isinstance(agent_value, list) else [agent_value]
                if not any(item in agent_list for item in req_value): return False
            elif isinstance(agent_value, list):
                if req_value not in agent_value: return False
            elif agent_value != req_value: return False
        return True
```

```
# --- pydagi.task: Task Manager ---
class TaskManager:
    """Manages the lifecycle of tasks. (Sec 8.2.3)"""
    def __init__(self, agent_manager: AgentManager, allocation_strategy:
                                                  TaskAllocationStrategy):
        self._tasks: Dict[str, Task] = {}
        self._agent_manager = agent_manager
        self._allocation_strategy = allocation_strategy
        self._iteration_controllers: Dict[str, "IterationController"] = {}
        self._network_ref: Optional["DAGINetwork"] = None
        # Define a stable ID for the controller/manager entity
        self._controller_agent_id = "dagi_controller"
        logger.info(f"TaskManager initialized. Controller Agent ID: {self.
                                                      _controller_agent_id}")

    def link_network(self, network: "DAGINetwork"):
         self._network_ref = network
         # Register controller ID with comm channel if needed (e.g., for replies in local
                                                       mode)
         if hasattr(network.comm_channel, 'register_agent'):
              network.comm_channel.register_agent(self._controller_agent_id)

    async def submit_task(self,
                          description: str,
                          input_data: Optional[Dict[str, Any]] = None,
                          requirements: Optional[Dict[str, Any]] = None,
                          evaluation_criteria: Optional[Dict[str, Any]] = None,
                          iteration_config: Optional[Dict[str, Any]] = None) -> str:
        """Creates, allocates, and starts the iteration process for a new task."""
        if not self._network_ref:
             raise DAGIError("TaskManager needs a linked DAGINetwork instance.")

        task = Task(description, input_data=input_data, requirements=requirements,
                                                      evaluation_criteria=
                                                      evaluation_criteria)
        self._tasks[task.task_id] = task
        logger.info(f"Task {task.task_id} submitted: '{description}'")

        try:
            available_agents = self._agent_manager.get_all_agents()
            proposer_agents, critic_agents = await self._allocation_strategy.allocate(task,
                                                          available_agents)

            if not proposer_agents:
                 task.update_status("failed", "Allocation failed: No proposing agents.")
                 raise TaskAllocationError(f"Task {task.task_id}: No proposing agents found.
                                                               ")

            iter_conf = self._network_ref.config.get('iteration_defaults', {}).copy()
            if iteration_config: iter_conf.update(iteration_config)

            # Resolve stopping condition class
            stopping_condition_cls = iter_conf.get("stopping_condition_class",
                                                          MaxIterationsCondition)
            if isinstance(stopping_condition_cls, str):
                 try: stopping_condition_cls = globals()[stopping_condition_cls]
                 except KeyError: raise ConfigurationError(f"Stopping condition class '{
                                                               stopping_condition_cls}' not
                                                               found.")
            if not isinstance(stopping_condition_cls, type) or not issubclass(
                                                          stopping_condition_cls,
                                                          StoppingCondition):
                 raise ConfigurationError(f"Invalid stopping condition class: {
                                                               stopping_condition_cls}")
            stopping_config = iter_conf.get("stopping_condition_config", {})
            stopping_condition = stopping_condition_cls(**stopping_config)

            # Resolve controller class
            controller_cls = iter_conf.get("controller_class", IterationController)
            if isinstance(controller_cls, str):
                 try: controller_cls = globals()[controller_cls]
                 except KeyError: raise ConfigurationError(f"Controller class '{
                                                             controller_cls}' not found.")
            if not isinstance(controller_cls, type) or not issubclass(controller_cls,
                                                         IterationController):
                 raise ConfigurationError(f"Invalid controller class: {controller_cls}")


            controller = controller_cls(
                task=task,
                agent_manager=self._agent_manager,
                communication_channel=self._network_ref.get_communication_channel(),
                proposer_agents=proposer_agents,
                critic_agents=critic_agents,
                stopping_condition=stopping_condition,
                controller_agent_id=self._controller_agent_id,
                config=iter_conf
            )
            self._iteration_controllers[task.task_id] = controller
            task.update_status("running")

            # Start the controller's loop in the background
            asyncio.create_task(controller.run_iteration_loop())
            return task.task_id

        except (TaskAllocationError, ConfigurationError, Exception) as e:
            logger.exception(f"Error submitting task {task.task_id}: {e}")
            task.update_status("failed", f"Submission error: {e}")
            if task.task_id in self._iteration_controllers: del self._iteration_controllers[
                                                         task.task_id]
            # Re-raise specific config/allocation errors, wrap others
            if isinstance(e, (TaskAllocationError, ConfigurationError)): raise
            else: raise DAGIError(f"Failed to submit task {task.task_id}: {e}")


    def get_task(self, task_id: str) -> Optional[Task]:
        """Retrieves a task by its ID."""
        return self._tasks.get(task_id)

    def get_task_status(self, task_id: str) -> Optional[str]:
        task = self.get_task(task_id)
        return task.status if task else None

    def get_result(self, task_id: str) -> Optional[Any]:
        task = self.get_task(task_id)
        if task and task.status in ["completed", "converged"]:
            return task.results
        return None

    def get_task_history(self, task_id: str) -> Optional[List[Dict[str, Any]]]:
         task = self.get_task(task_id)
         return task.history if task else None

    def task_finished_callback(self, task_id: str):
         """Callback for IterationController to signal task completion."""
         if task_id in self._iteration_controllers:
              del self._iteration_controllers[task_id]
              logger.info(f"Iteration controller removed for finished task {task_id}.")
         else:
              logger.warning(f"Task finished callback received for unknown/already removed
                                                              controller: {task_id}")


# --- pydagi.iteration: Stopping Conditions ---
class StoppingCondition(abc.ABC):
    """ABC for iteration termination criteria. (Sec 8.2.4)"""
    @abc.abstractmethod
    def check(self, iteration_state: Dict[str, Any]) -> bool:
        """Returns True if the iteration process should stop."""
        raise NotImplementedError

class MaxIterationsCondition(StoppingCondition):
    """Stops after a fixed number of iterations."""
    def __init__(self, max_iterations: int = 10):
        if not isinstance(max_iterations, int) or max_iterations <= 0:
             raise ValueError("max_iterations must be a positive integer.")
        self.max_iterations = max_iterations
        logger.info(f"Initialized MaxIterationsCondition with max_iterations={max_iterations
                                                      }")

    def check(self, iteration_state: Dict[str, Any]) -> bool:
        current_iteration = iteration_state.get("current_iteration", 0)
        # Stop *before* exceeding max_iterations (check is done at start of loop)
        stop = current_iteration >= self.max_iterations
        if stop:
             logger.info(f"Stopping condition check: Max iterations ({self.max_iterations})
                                                           will be reached or exceeded by
                                                           next iteration ({
                                                           current_iteration+1}). Stopping."
                                                           )
        return stop

class SolutionConvergenceCondition(StoppingCondition):
    """Stops when changes between iterations fall below a threshold (placeholder). (Sec 8.2.
                                               4)"""
    def __init__(self, threshold: float = 0.95, metric: str = "semantic_similarity"):
        if not isinstance(threshold, (float, int)) or not 0.0 <= threshold <= 1.0:
             raise ValueError("threshold must be a float between 0.0 and 1.0")
        self.threshold = float(threshold)
        self.metric = metric
        logger.info(f"Initialized SolutionConvergenceCondition with threshold={self.
                                                      threshold}, metric={self.metric}")

    def check(self, iteration_state: Dict[str, Any]) -> bool:
        history = iteration_state.get("history", [])
        current_iteration = iteration_state.get("current_iteration", 0)
        if current_iteration < 1 or len(history) < 2: # Need at least 1 full iteration
                                                      completed
            return False

        last_iter_data = history[-1]
        prev_iter_data = history[-2]
        current_proposals = last_iter_data.get("proposals") # Proposals key holds proposals/
                                                      revisions
        prev_proposals = prev_iter_data.get("proposals")

        similarity = self._calculate_similarity(prev_proposals, current_proposals)
        stop = similarity >= self.threshold
        if stop:
             logger.info(f"Stopping condition met: Solution convergence threshold ({self.
                                                           threshold}) reached with
                                                           similarity {similarity:.4f} at
                                                           iteration {current_iteration}.")
        else:
             logger.debug(f"Convergence check: Similarity {similarity:.4f} < threshold {self
                                                              .threshold} at iteration {
                                                              current_iteration}.")
        return stop

    def _calculate_similarity(self, prev_proposals: Optional[List[Dict]], current_proposals:
                                                      Optional[List[Dict]]) -> float:
        # --- Placeholder --- Requires actual NLP implementation ---
        if not prev_proposals or not current_proposals: return 0.0
        try:
            # Compare first proposal texts (very basic)
            prev_text = prev_proposals[0].get("response_data", {}).get("proposal") or
                                                             prev_proposals[0].get("
                                                             response_data", {}).get("
                                                             revised_proposal", "")
            curr_text = current_proposals[0].get("response_data", {}).get("proposal") or
                                                             current_proposals[0].get("
                                                             response_data", {}).get("
                                                             revised_proposal", "")
            if prev_text and curr_text:
                 # Mock similarity based on content hash or length (replace with actual
                                                                  metric)
                 sim = 1.0 if hash(prev_text) == hash(curr_text) else max(0.0, 1.0 - abs(len
                                                                  (prev_text) - len(curr_text))
                                                                   / max(len(prev_text), len(
                                                                  curr_text), 1))
                 logger.debug(f"Placeholder similarity: {sim:.4f}")
                 return sim
            else: return 0.0
        except Exception as e:
             logger.error(f"Error calculating similarity: {e}")
             return 0.0


# --- pydagi.iteration: Iteration Controller ---
class IterationController:
    """Orchestrates the iterative refinement loop for a task. (Sec 8.2.4)"""
    def __init__(self,
                 task: Task,
                 agent_manager: AgentManager,
                 communication_channel: CommunicationChannel,
                 proposer_agents: List[BaseAgent],
                 critic_agents: List[BaseAgent],
                 stopping_condition: StoppingCondition,
                 controller_agent_id: str,
                 config: Optional[Dict[str, Any]] = None):
        self.task = task
        self.agent_manager = agent_manager
        self.comm_channel = communication_channel
        self.proposer_agents = proposer_agents
        self.critic_agents = critic_agents
        self.stopping_condition = stopping_condition
        self.controller_agent_id = controller_agent_id
        self.config = config or {}
        self.iteration_num = 0
        self.history: List[Dict[str, Any]] = []
        self.anonymity_enabled: bool = self.config.get("anonymity", True)
        self._task_running = False

        logger.info(f"IterationController initialized for task {self.task.task_id}.
                                                     Controller ID: {self.
                                                     controller_agent_id}, Anonymity: {self
                                                     .anonymity_enabled}")
        if not self.proposer_agents: logger.error(f"Task {self.task.task_id}: No proposer
                                                agents assigned!")

    def _get_iteration_state(self) -> Dict[str, Any]:
        """Packages current state for stopping condition check."""
        return {"current_iteration": self.iteration_num, "history": self.history, "
                                                task_status": self.task.status}

    async def _collect_responses(self,
                                recipient_agents: List[BaseAgent],
                                initiating_message_type: str,
                                expected_response_type: str,
                                prompt_content: Dict[str, Any],
                                agent_specific_content: Optional[Dict[str, Dict[str, Any]]]
                                                                = None, #
                                                                Optional map
                                                                agent_id ->
                                                                specific
                                                                content
                                timeout_per_agent: float = 300.0
                                ) -> List[Dict[str, Any]]:
        """Sends messages and collects responses asynchronously."""
        responses = []
        tasks = []
        agent_specific_content = agent_specific_content or {}

        for agent in recipient_agents:
             # Combine general prompt content with agent-specific content
             final_content = prompt_content.copy()
             final_content.update(agent_specific_content.get(agent.agent_id, {}))

             msg = Message(
                 sender_id=self.controller_agent_id,
                 recipient_id=agent.agent_id,
                 message_type=initiating_message_type,
                 content=final_content,
                 task_id=self.task.task_id,
                 metadata={"reply_to": self.controller_agent_id, "iteration": self.
                                                      iteration_num}
             )

             async def get_response(agent: BaseAgent, msg: Message):
                  try:
                      response_msg = await asyncio.wait_for(agent.process(msg), timeout=
                                                           timeout_per_agent)
                      if response_msg and response_msg.message_type ==
                                                           expected_response_type:
                          return {"agent_id": agent.agent_id, "response_data": response_msg.
                                                               content, "message_id
                                                               ": response_msg.
                                                               message_id}
                      elif response_msg and response_msg.message_type == "PROCESSING_ERROR":
                           logger.error(f"Agent {agent.agent_id} processing error: {
                                                                response_msg.
                                                                content.get('error
                                                                ')}")
                           return {"agent_id": agent.agent_id, "error": response_msg.content
                                                                .get('error', '
                                                                Unknown error')}
                      else:
                        logger.warning(f"Agent {agent.agent_id} unexpected response: type=
                                                                    {response_msg.
                                                                    message_type if
                                                                    response_msg else '
                                                                    None'} (expected {
                                                                    expected_response_type
                                                                    })")
                        return {"agent_id": agent.agent_id, "error": f"Incorrect/No
                                                                    response (expected {
                                                                    expected_response_type
                                                                    })"}
                except asyncio.TimeoutError:
                     logger.error(f"Timeout waiting for {agent.agent_id} response.")
                     return {"agent_id": agent.agent_id, "error": "Timeout"}
                except Exception as e:
                     logger.exception(f"Error getting response from {agent.agent_id}: {e}"
                                                                    )
                     return {"agent_id": agent.agent_id, "error": f"Unexpected error: {e}"
                                                                    }

            tasks.append(asyncio.create_task(get_response(agent, msg)))

        completed_tasks_results = await asyncio.gather(*tasks)
        valid_responses = [res for res in completed_tasks_results if res and not res.get("
                                                   error")]
        errors = [res for res in completed_tasks_results if res and res.get("error")]
        if errors: logger.error(f"Task {self.task.task_id} Iteration {self.iteration_num}: {
                                                   len(errors)} agent(s) failed.")

        logger.info(f"Collected {len(valid_responses)} valid responses (type {
                                                   expected_response_type}) for task {
                                                   self.task.task_id} iteration {self.
                                                   iteration_num}.")
        return valid_responses


    async def run_iteration_loop(self):
        """Executes the generate-critique-refine loop."""
        if self._task_running: return # Prevent concurrent runs for same controller
        self._task_running = True
        logger.info(f"Starting iteration loop for task {self.task.task_id}...")
        final_result = None
        loop_error = None

        try:
            # Check initial stopping condition
            if self.stopping_condition.check(self._get_iteration_state()):
                logger.info(f"Task {self.task.task_id} stopping condition met before first
                                                           iteration.")
                self.task.update_status("completed")
                return

            last_proposals_map: Dict[str, Any] = {} # agent_id -> last proposal content

            while True: # Loop continues until stopping condition met or error
                self.iteration_num += 1
                logger.info(f"--- Task {self.task.task_id} - Iteration {self.iteration_num}
                                                           Start ---")
                iteration_log = {"iteration": self.iteration_num, "proposals": [], "
                                                           critiques": []}

                # --- 1. Proposal/Revision Phase ---
                current_proposals: List[Dict[str, Any]]
                if self.iteration_num == 1:
                    logger.info(f"Task {self.task.task_id} Iteration {self.iteration_num}:
                                                            Generating initial
                                                            proposals...")
                    prompt_content = {"description": self.task.description, **self.task.
                                                            input_data}
                    current_proposals = await self._collect_responses(
                        recipient_agents=self.proposer_agents,
                        initiating_message_type="TASK_PROMPT",
                        expected_response_type="PROPOSAL",
                        prompt_content=prompt_content
                    )
                else: # Revision phase
                    logger.info(f"Task {self.task.task_id} Iteration {self.iteration_num}:
                                                            Generating revisions...")
                    last_critiques_data = self.history[-1].get("critiques", [])
                    if not last_critiques_data:
                        logger.warning(f"Task {self.task.task_id}: No critiques from
                                                                previous iteration.
                                                                Stopping.")
                        self.task.update_status("completed", "Stopped: No critiques for
                                                                revision.")
                        break

                    agent_specific_prompts = {}
                    agents_to_prompt = []
                    for proposer in self.proposer_agents:
                        last_prop = last_proposals_map.get(proposer.agent_id)
                        if last_prop is None:
                            logger.warning(f"Task {self.task.task_id}: Agent {proposer.
                                                                    agent_id} has no
                                                                    previous proposal
                                                                    to revise.")
                            continue
                        # Provide all critiques to each proposer for now
                        critique_texts = [c.get("response_data", {}).get("critique", "") for
                                                                 c in
                                                                last_critiques_data]
                        agent_specific_prompts[proposer.agent_id] = {
                            "last_proposal": last_prop,
                            "critiques": critique_texts,
                            "original_task_description": self.task.description
                        }
                        agents_to_prompt.append(proposer)

                    if not agents_to_prompt:
                        logger.error(f"Task {self.task.task_id}: No agents available for
                                                                revision.")
                        self.task.update_status("failed", "No agents for revision.")
                        break

                    current_proposals = await self._collect_responses(
                        recipient_agents=agents_to_prompt,
                        initiating_message_type="REVISION_REQUEST",
                        expected_response_type="PROPOSAL",
                        prompt_content={}, # Base content is empty, specific content passed
                                                                below
                        agent_specific_content=agent_specific_prompts
                    )

                iteration_log["proposals"] = current_proposals # Store proposals/revisions

                # Update map of last known proposals
                for prop_data in current_proposals:
                     agent_id = prop_data.get("agent_id")
                     resp_data = prop_data.get("response_data", {})
```

```
                        prop_content = resp_data.get("proposal") or resp_data.get("
                                                                        revised_proposal")
                        if agent_id and prop_content:
                             last_proposals_map[agent_id] = prop_content

                   if not current_proposals: # Check if any agent succeeded in proposing/
                                                                   revising
                       logger.error(f"Task {self.task.task_id} Iteration {self.iteration_num}:
                                                                       No valid proposals/
                                                                       revisions generated.")
                       self.task.update_status("failed", f"No valid output in iteration {self.
                                                                       iteration_num}.")
                       break

                   # --- 2. Critique Phase ---
                   if self.critic_agents:
                       logger.info(f"Task {self.task.task_id} Iteration {self.iteration_num}:
                                                                       Generating critiques...")
                       critique_request_content = {"proposals": [], "original_task_description"
                                                                       : self.task.description}
                       anonymized_proposals_map = {} # display_id -> original_agent_id

                       for prop_data in current_proposals:
                            agent_id = prop_data.get("agent_id")
                            resp_data = prop_data.get("response_data", {})
                            proposal_text = resp_data.get("proposal") or resp_data.get("
                                                                            revised_proposal")
                            if proposal_text:
                                 prop_info = {"proposal": proposal_text}
                                 display_id = f"proposal_{self.iteration_num}_{uuid.uuid4().hex
                                                                                 [:6]}"
                                 if self.anonymity_enabled:
                                      prop_info["display_id"] = display_id
                                      anonymized_proposals_map[display_id] = agent_id
                                 else:
                                      prop_info["agent_id"] = agent_id # Include original
                                                                                      author if
                                                                                      not
                                                                                      anonymous
                                 critique_request_content["proposals"].append(prop_info)

                       if critique_request_content["proposals"]:
                            original_proposer_ids = set(anonymized_proposals_map.values()) if
                                                                            self.
                                                                            anonymity_enabled
                                                                            else {p.get("agent_id
                                                                            ") for p in
                                                                            critique_request_content
                                                                            ["proposals"] if p.
                                                                            get("agent_id")}
                            eligible_critics = [a for a in self.critic_agents if a.agent_id not
                                                                             in
                                                                            original_proposer_ids
                                                                            ]

                            if not eligible_critics and self.critic_agents:
                                 logger.warning(f"Task {self.task.task_id} Iteration {self.
                                                                                 iteration_num}:
                                                                                 Critics overlap
                                                                                 with proposers.
                                                                                 Allowing overlap
                                                                                 .")
                                 eligible_critics = self.critic_agents # Allow overlap

                            if eligible_critics:
                              critiques = await self._collect_responses(
                                  recipient_agents=eligible_critics,
                                  initiating_message_type="CRITIQUE_REQUEST",
                                  expected_response_type="CRITIQUE",
                                  prompt_content=critique_request_content
                              )
                              iteration_log["critiques"] = critiques
                         else: logger.info(f"Task {self.task.task_id} Iteration {self.
                                                                       iteration_num}: No
                                                                       eligible critics.")
                    else: logger.warning(f"Task {self.task.task_id} Iteration {self.
                                                                  iteration_num}: No valid
                                                                  proposals found for
                                                                  critique.")
                else: logger.info(f"Task {self.task.task_id} Iteration {self.iteration_num}:
                                                               Skipping critique phase (no
                                                              critics).")

                # --- 3. Record History & Check Stopping Condition ---
                self.history.append(iteration_log)
                self.task.history = self.history # Update task with latest history

                if self.stopping_condition.check(self._get_iteration_state()):
                     logger.info(f"Task {self.task.task_id} stopping condition met AFTER
                                                                   iteration {self.
                                                                   iteration_num}.")
                     self.task.update_status("converged")
                     break

                # Safety break
                if self.iteration_num >= self.config.get("absolute_max_iterations", 20):
                    logger.warning(f"Task {self.task.task_id} reached absolute max
                                                                  iterations ({self.
                                                                  iteration_num}). Stopping.
                                                                  ")
                    self.task.update_status("completed", "Reached max iterations.")
                    break

                logger.info(f"--- Task {self.task.task_id} - Iteration {self.iteration_num}
                                                              End ---")
                await asyncio.sleep(0.05) # Small yield


            # --- Loop Finished ---
            if self.task.status not in ["failed", "failed_timeout"]:
                 final_result = self._select_final_result(self.history)
                 self.task.set_results(final_result, self.history)
                 # Update status if it was still running (e.g. hit safety break)
                 if self.task.status == "running":
                     self.task.update_status("completed", "Finished loop.")

        except Exception as e:
            loop_error = str(e)
            logger.exception(f"Error during iteration loop for task {self.task.task_id}: {e}
                                                          ")
            self.task.update_status("failed", f"Runtime error: {e}")
            self.task.set_results({"error": loop_error}, self.history)
        finally:
            logger.info(f"Iteration loop finished for task {self.task.task_id}. Final status
                                                          : {self.task.status}")
            # Notify TaskManager
            if self.agent_manager._agent_manager_ref and self.agent_manager.task_manager:
                self.agent_manager.task_manager.task_finished_callback(self.task.task_id)
            self._task_running = False
```

```
    def _select_final_result(self, history: List[Dict[str, Any]]) -> Optional[Any]:
        """Selects the final result from history. (Placeholder)"""
        if not history: return None
        # Strategy: Find last iteration with proposals, return the first one's data
        last_iter_with_proposals = next((i for i in reversed(history) if i.get("proposals"))
                                                    , None)
        if last_iter_with_proposals and last_iter_with_proposals["proposals"]:
            # Return the 'response_data' dictionary
            return last_iter_with_proposals["proposals"][0].get("response_data")
        logger.warning(f"Task {self.task.task_id}: Could not select final result from
                                                    history.")
        return None

# --- pydagi.utilities: Configuration Loading ---
# Requires PyYAML: pip install pyyaml
def load_config(filepath: str) -> Dict[str, Any]:
    """Loads configuration from a YAML file."""
    try: import yaml
    except ImportError: raise ConfigurationError("PyYAML required: 'pip install pyyaml'.")
    try:
        with open(filepath, 'r') as f: config_data = yaml.safe_load(f)
        if not isinstance(config_data, dict): raise ConfigurationError(f"Config file {
                                                    filepath} not a dictionary.")
        logger.info(f"Configuration loaded from {filepath}")
        return config_data
    except FileNotFoundError: raise ConfigurationError(f"Config file not found: {filepath}")
    except yaml.YAMLError as e: raise ConfigurationError(f"Error parsing YAML file {filepath
                                                }: {e}")
    except Exception as e: raise ConfigurationError(f"Failed loading config {filepath}: {e}"
                                                )

# --- pydagi.utilities: Prompt Templating (Placeholder) ---
class PromptTemplateManager:
     """Manages prompt templates (placeholder)."""
     def __init__(self): self.templates = {}
     def get_prompt(self, template_name: str, context: Dict[str, Any]) -> str:
          # Replace with actual templating (e.g., Jinja2)
          return f"Mock Prompt for {template_name}: {context}"

# --- pydagi.utilities: Evaluation Helpers (Placeholder) ---
def evaluate_task_result(task: Task) -> Dict[str, Any]:
     """Evaluates task result based on criteria (placeholder)."""
     if not task.evaluation_criteria: return {"message": "No evaluation criteria."}
     if task.status not in ["completed", "converged"]: return {"message": f"Cannot evaluate
                                                  status {task.status}."}
     logger.info(f"Placeholder: Evaluating task {task.task_id}...")
     return {"mock_score": random.uniform(0.6, 0.95)}


# --- DAGINetwork: Main Orchestration Class ---
class DAGINetwork:
    """Top-level class managing the DAGI ecosystem."""
    def __init__(self, config_path: Optional[str] = None, config_dict: Optional[Dict[str,
                                                Any]] = None):
        if config_path: self.config = load_config(config_path)
        elif config_dict: self.config = config_dict
        else: self.config = {}
        logger.info("Initializing DAGINetwork...")

        # Init Comm Channel
        comm_config = self.config.get('communication', {'type': 'local'})
        comm_type = comm_config.get('type')
        if comm_type == 'local': self.comm_channel: CommunicationChannel =
                                                    LocalCommunicationChannel()
        elif comm_type == 'distributed': raise NotImplementedError("Distributed channel
                                                    needs implementation.")
        else: raise ConfigurationError(f"Unsupported communication type: {comm_type}")

        # Init Managers
        self.agent_manager = AgentManager(self.comm_channel)
        if isinstance(self.comm_channel, LocalCommunicationChannel): self.comm_channel.
                                                    link_manager(self.agent_manager)

        allocator_config = self.config.get('task_allocation', {'type': 'capability_based'})
        allocator_type = allocator_config.get('type')
        if allocator_type == 'capability_based': self.task_allocator: TaskAllocationStrategy
                                                     = CapabilityBasedAllocator()
        # Add other allocators here
        else: raise ConfigurationError(f"Unsupported task allocator type: {allocator_type}")

        self.task_manager = TaskManager(self.agent_manager, self.task_allocator)
        self.task_manager.link_network(self)
        self.agent_manager.link_task_manager(self.task_manager)

        self.topology_manager = TopologyManager(self.config.get('topology'))
        self.prompt_manager = PromptTemplateManager()
        self._is_running = False

        self._load_agents_from_config()
        logger.info("DAGINetwork initialized.")

    def _load_agents_from_config(self):
        """Loads agents from the configuration."""
        agents_config = self.config.get('agents', [])
        if not agents_config: logger.warning("No agents defined in configuration.")
        agent_classes = {'LLMAgent': LLMAgent, 'CriticAgent': CriticAgent, 'CoderAgent':
                                                    CoderAgent}

        for agent_conf in agents_config:
            agent_type_str = agent_conf.get('type', 'LLMAgent')
            agent_class = agent_classes.get(agent_type_str)
            if not agent_class:
                logger.error(f"Unknown agent type '{agent_type_str}' for agent {agent_conf.
                                                            get('id')}. Skipping.")
                continue
            try:
                agent = agent_class(
                    agent_id=agent_conf.get('id'),
                    capabilities=agent_conf.get('capabilities', {}),
                    config=agent_conf.get('config', {})
                )
                self.agent_manager.add_agent(agent)
            except Exception as e:
                logger.error(f"Failed to load agent {agent_conf.get('id')} (type {
                                                            agent_type_str}): {e}")


    def get_communication_channel(self) -> CommunicationChannel: return self.comm_channel

    async def start(self):
        if self._is_running: logger.warning("DAGINetwork already running."); return
        logger.info("Starting DAGINetwork...")
        try:
            await self.comm_channel.setup()
            await self.agent_manager.start_all_agents()
            self._is_running = True
            logger.info("DAGINetwork started successfully.")
        except Exception as e:
            logger.exception(f"Failed to start DAGINetwork: {e}")
            await self.stop() # Attempt cleanup

    async def stop(self):
        if not self._is_running: return
        logger.info("Stopping DAGINetwork...")
        try: await self.agent_manager.stop_all_agents()
        except Exception as e: logger.exception(f"Error stopping agents: {e}")
        try: await self.comm_channel.teardown()
        except Exception as e: logger.exception(f"Error tearing down comm channel: {e}")
        self._is_running = False
        logger.info("DAGINetwork stopped.")

    async def run_task(self, description: str, **kwargs) -> Optional[Dict[str, Any]]:
         """Submits and runs a task, returning status and results."""
         if not self._is_running:
              logger.error("Cannot run task: DAGINetwork not running.")
              return {"status": "failed", "error": "Network not running."}
         task_id = None
         try:
              task_id = await self.task_manager.submit_task(description, **kwargs)
              logger.info(f"Task {task_id} submitted. Waiting for completion...")
              task_timeout = float(self.config.get("task_execution_timeout", 300.0))
              start_time = time.monotonic()
              while True:
                   status = self.task_manager.get_task_status(task_id)
                   if status in ["completed", "failed", "converged", "failed_timeout"]:
                                                             break
                   if time.monotonic() - start_time > task_timeout:
                        logger.error(f"Task {task_id} timed out after {task_timeout}s.")
                        if (task := self.task_manager.get_task(task_id)) and task.status ==
                                                                  "running":
                             task.update_status("failed_timeout", "Execution timed out.")
                        status = "failed_timeout"
                        break
                   await asyncio.sleep(0.5) # Check status less frequently

              result = self.task_manager.get_result(task_id)
              task = self.task_manager.get_task(task_id)
              evaluation = evaluate_task_result(task) if task else {}
              return {"task_id": task_id, "status": status, "result": result, "evaluation":
                                                       evaluation, "history_summary": f
                                                       "{len(task.history if task else
                                                       [])} iterations", "error": task.
                                                       error_info if task else None}

         except (TaskAllocationError, ConfigurationError, DAGIError) as e:
              logger.error(f"Task '{description}' failed: {e}")
              if task_id and (task := self.task_manager.get_task(task_id)) and task.status
                                                       not in ["failed"]: task.
                                                       update_status("failed", str(e))
              return {"description": description, "status": "failed", "error": str(e)}
         except Exception as e:
             logger.exception(f"Unexpected error running task '{description}': {e}")
             if task_id and (task := self.task_manager.get_task(task_id)) and task.status
                                                      not in ["failed"]: task.
                                                      update_status("failed", "
                                                      Unexpected error during execution
                                                      .")
             return {"description": description, "status": "failed", "error": "Unexpected
                                                      error during execution."}

# --- Main Entry Point (Example Usage, requires config.yaml) ---
async def main():
    config_file = "config.yaml"
    network = None
    try:
          # Create dummy config if needed
          import os
          if not os.path.exists(config_file):
               dummy_config = """
communication:
  type: local
agents:
  - id: proposer_agent_1
    type: LLMAgent
    config: { model: "mock-proposer" }
    capabilities: { task_types: ["proposal", "text_generation"], domain: "creative" }
  - id: critic_agent_1
    type: CriticAgent
    config: { model: "mock-critic" }
    capabilities: { domain: "analytical" } # Augments internal role capability
task_allocation: { type: capability_based }
iteration_defaults:
  anonymity: True
  stopping_condition_class: MaxIterationsCondition
  stopping_condition_config: { max_iterations: 3 }
task_execution_timeout: 60.0 # Shorter timeout for demo
"""
               with open(config_file, 'w') as f: f.write(dummy_config)
               logger.info(f"Created dummy configuration file: {config_file}")

          network = DAGINetwork(config_path=config_file)
          await network.start()

          # --- Example Task ---
          task_desc = "Suggest 3 taglines for a collaborative AI framework."
          task_reqs = {
              "proposer_capabilities": {"task_types": ["text_generation"], "domain": "
                                                       creative"},
              "critic_capabilities": {"roles": ["critic"]}
          }
          iter_conf = {"stopping_condition_class": MaxIterationsCondition, "
                                                   stopping_condition_config": {"
                                                   max_iterations": 2}}

          logger.info(f"\n--- Submitting Task: '{task_desc}' ---")
          result = await network.run_task(description=task_desc, requirements=task_reqs,
                                                   iteration_config=iter_conf)

          logger.info(f"\n--- Task Result ---")
          if result:
              print(json.dumps(result, indent=2, default=str))
              # Optionally print full history from task object
              # if result.get("task_id") and (task := network.task_manager.get_task(result["
                                                       task_id"])):
              #     print("\n--- Task History ---")
              #     print(json.dumps(task.history, indent=2, default=str))
          else: print("Task execution did not return results.")

    except (ConfigurationError, DAGIError) as e: logger.error(f"Execution failed: {e}")
    except Exception as e: logger.exception(f"An unexpected error occurred in main: {e}")
    finally:
         if network and network._is_running:
              logger.info("Shutting down network...")
              await network.stop()
         logger.info("Main function finished.")

if __name__ == "__main__":
    asyncio.run(main())
```