File Breakdown: `src/core/knowledge_base.py`

File Location

`src/core/knowledge_base.py`

Overview

The `knowledge_base.py` file implements the central knowledge storage system for the multi-agent framework. The KnowledgeBase class provides a shared repository for storing context, results, and configuration data that needs to be accessed by different agents. It supports both synchronous and asynchronous operations, persistent storage, data categorization, and session tracking.

Key Responsibilities

- Store and retrieve data for all agents in the system
- Provide both synchronous and asynchronous data access methods
- Categorize stored data for easier retrieval
- Maintain session history and interaction logs
- Support persistent storage with automatic backups
- Provide context retrieval for agent-specific operations
- Archive old sessions to manage storage

Core Functionality

Class Definition

```python
class KnowledgeBase:
    """
    A central repository for storing context, results, and configuration data.
    Enhanced with persistent storage, categorization, and session tracking.
    """
    def __init__(self, storage_path="knowledge_db", use_persistence=True):
        self.storage = {}
        self.storage_path = storage_path
        self.use_persistence = use_persistence
        # Create a lock for thread safety in async operations
        self.lock = asyncio.Lock()

        # Create storage directory if persistence is enabled
        if use_persistence:
            os.makedirs(storage_path, exist_ok=True)
            self.load_from_disk()
```

Data Storage and Retrieval

Synchronous methods for data operations:

```python
def set_item(self, key: str, value: any, category=None):
    """
    Synchronous set with optional category tagging.
    Parameters:
        key (str): The key to store the value under
        value (any): The value to store
        category (str, optional): Category to tag this item with
    """
    # Store in categorized structure if category is provided
    if category:
        if "__categories__" not in self.storage:
            self.storage["__categories__"] = {}

        if category not in self.storage["__categories__"]:
            self.storage["__categories__"][category] = []

        # Add key to category if not already there
        if key not in self.storage["__categories__"][category]:
            self.storage["__categories__"][category].append(key)

    # Store the actual data
    self.storage[key] = value

    # Save changes to disk if persistence is enabled
    if self.use_persistence:
        self.save_to_disk()

def get_item(self, key: str) -> any:
    """Synchronous get - original method preserved"""
    return self.storage.get(key)

def update(self, data: dict):
    """
    Synchronous update with optional persistence

    Parameters:
        data (dict): Dictionary of key-value pairs to update
    """
    self.storage.update(data)

    # Save changes to disk if persistence is enabled
    if self.use_persistence:
        self.save_to_disk()
```

Asynchronous methods for thread-safe operations:

```
async def set_item_async(self, key: str, value: any, category=None):
    """Thread-safe asynchronous set with optional category"""
    async with self.lock:
        self.set_item(key, value, category)

async def get_item_async(self, key: str) -> any:
    """Thread-safe asynchronous get"""
    async with self.lock:
        return self.storage.get(key)

async def update_async(self, data: dict):
    """Thread-safe asynchronous update"""
    async with self.lock:
        self.storage.update(data)
        if self.use_persistence:
            await asyncio.to_thread(self.save_to_disk)
```

Persistence Management

Methods for saving and loading data:

```
def load_from_disk(self):
    """Load knowledge data from persistent storage"""
    try:
        main_db_path = os.path.join(self.storage_path, "main_db.json")
        if os.path.exists(main_db_path):
            with open(main_db_path, 'r') as f:
                stored_data = json.load(f)
                self.storage.update(stored_data)
            print(f"Loaded {len(stored_data)} items from persistent storage")
    except Exception as e:
        print(f"Error loading knowledge base: {str(e)}")

def save_to_disk(self):
    """Save current knowledge to persistent storage"""
    if not self.use_persistence:
        return

    try:
        # Ensure directory exists
        os.makedirs(self.storage_path, exist_ok=True)

        # Save main database
        main_db_path = os.path.join(self.storage_path, "main_db.json")
        with open(main_db_path, 'w') as f:
            json.dump(self.storage, f, indent=2, default=str)

        # Create periodic backup (once per hour)
        hour_timestamp = datetime.datetime.now().strftime("%Y%m%d_%H")
        backup_path = os.path.join(self.storage_path, f"backup_{hour_timestamp}.json")

        # Only create if doesn't exist for this hour
        if not os.path.exists(backup_path):
            with open(backup_path, 'w') as f:
                json.dump(self.storage, f, default=str)

            # Limit number of backups
            self._cleanup_old_backups()
    except Exception as e:
        print(f"Error saving knowledge base: {str(e)}")
```

Session Management

Methods for tracking and logging interactions:

```
def create_session(self, session_id=None):
    """
    Create a new session for tracking interactions

    Parameters:
        session_id (str, optional): Custom session ID

    Returns:
        str: The session ID
    """
    if session_id is None:
        session_id = f"session_{int(time.time())}"

    session_data = {
        "id": session_id,
        "start_time": datetime.datetime.now().isoformat(),
        "interactions": [],
        "models_created": [],
        "analyses_performed": [],
        "reports_generated": []
    }

    self.set_item(f"session_{session_id}", session_data, category="sessions")
    self.set_item("current_session", session_id)
    return session_id

def log_interaction(self, prompt, response, agent="Nova", function=None):
    """
    Log an interaction within the current session

    Parameters:
        prompt (str): The prompt or input
        response (str): The response or output
        agent (str): The agent that handled the interaction
        function (str, optional): The function that was called
    """
    current_session = self.get_item("current_session")
    if not current_session:
        current_session = self.create_session()

    session_data = self.get_item(f"session_{current_session}")
    if session_data:
        interaction = {
            "timestamp": datetime.datetime.now().isoformat(),
            "prompt": prompt,
            "response": response,
            "agent": agent,
            "function": function
        }

        session_data["interactions"].append(interaction)
        self.set_item(f"session_{current_session}", session_data, category="sessions")
```

Context and Data Retrieval

Methods for retrieving agent-specific context and categorized data:

```
def get_context_for_agent(self, agent_name, context_depth=5):
    """
    Get relevant context information for a specific agent

    Parameters:
        agent_name (str): Name of the agent (Nova, Emil, Ivan, Lola)
        context_depth (int): Number of recent interactions to include

    Returns:
        dict: Context information relevant to the agent
    """
    current_session = self.get_item("current_session")
    if not current_session:
        return {}

    # Get current session data
    session_data = self.get_item(f"session_{current_session}")
    if not session_data:
        return {}

    # Get recent interactions
    recent_interactions = []
    for interaction in reversed(session_data.get("interactions", [])):
        if len(recent_interactions) >= context_depth:
            break
        recent_interactions.append(interaction)

    # Get agent-specific context based on agent role
    agent_context = {"recent_interactions": recent_interactions}

    if agent_name == "Emil":
        # Energy modeling context
        agent_context["latest_model"] = self.get_item("latest_model_details")
        agent_context["energy_models"] = session_data.get("models_created", [])

    elif agent_name == "Lola":
        # Report writing context
        agent_context["latest_model"] = self.get_item("latest_model_details")
        agent_context["latest_analysis"] = self.get_item("latest_analysis_results")
        agent_context["reports"] = session_data.get("reports_generated", [])

    elif agent_name == "Ivan":
        # Code and image generation context
        agent_context["latest_image"] = self.get_item("last_dalle_prompt")
```

```python
    return agent_context

def get_items_by_category(self, category):
    """
    Retrieve all items in a specific category

    Parameters:
        category (str): The category to retrieve items from

    Returns:
        dict: Dictionary of items in the category
    """
    if "__categories__" not in self.storage or category not in
self.storage["__categories__"]:
        return {}

    return {key: self.storage.get(key)
            for key in self.storage["__categories__"][category]}
```

Search and Archival

Methods for searching data and archiving old sessions:

```python
def search_knowledge_base(self, query, categories=None):
    """
    Search the knowledge base for relevant information

    Parameters:
        query (str): Search query
        categories (list, optional): List of categories to search in

    Returns:
        list: List of matching items
    """
    results = []

    if categories is None:
        # Search all items
        for key, value in self.storage.items():
            if key == "__categories__":
                continue

            # Try to match in key or values
            if self._matches_query(key, query) or self._matches_query(value, query):
                results.append({"key": key, "value": value})
    else:
        # Search only in specified categories
        for category in categories:
            if "__categories__" in self.storage and category in
self.storage["__categories__"]:
                for key in self.storage["__categories__"][category]:
                    value = self.storage.get(key)
                    if self._matches_query(key, query) or self._matches_query(value,
query):
                        results.append({"key": key, "value": value, "category":
category})

    return results

def archive_old_sessions(self, days_threshold=30):
    """
    Archive sessions older than the threshold

    Parameters:
        days_threshold (int): Age in days after which to archive
    """
    if "__categories__" not in self.storage or "sessions" not in
self.storage["__categories__"]:
        return

    current_time = datetime.datetime.now()
    sessions_to_archive = []

    for session_key in self.storage["__categories__"]["sessions"]:
        session_data = self.storage.get(session_key)
        if not session_data or "start_time" not in session_data:
            continue

        try:
            start_time = datetime.datetime.fromisoformat(session_data["start_time"])
            age_days = (current_time - start_time).days

            if age_days > days_threshold:
                sessions_to_archive.append(session_key)
        except Exception:
            continue

    # Archive the old sessions
    if sessions_to_archive:
        archive_data = {"archived_sessions": {}}

        for session_key in sessions_to_archive:
            # Move to archive
            archive_data["archived_sessions"][session_key] = self.storage[session_key]

            # Remove from main storage
            del self.storage[session_key]
            self.storage["__categories__"]["sessions"].remove(session_key)

        # Save archive
        archive_path = os.path.join(self.storage_path,
f"archive_{int(time.time())}.json")
        with open(archive_path, 'w') as f:
            json.dump(archive_data, f, indent=2, default=str)
        print(f"Archived {len(sessions_to_archive)} old sessions")

        # Save main database after archiving
        if self.use_persistence:
            self.save_to_disk()
```

Key Features

1. **Shared Data Storage**: Central repository for all data needed by agents
2. **Data Categorization**: Organizes data into categories for easier access
3. **Async/Sync API**: Supports both synchronous and asynchronous operations
4. **Persistence**: Maintains data between program runs with automatic backups
5. **Session Management**: Tracks interaction history for context-aware responses
6. **Thread Safety**: Uses locks to ensure thread-safe operations in async mode
7. **Agent-Specific Context**: Provides relevant context data for each agent type
8. **Search Capability**: Allows searching across all stored data or within categories
9. **Archival**: Automatically archives old sessions to manage storage

Integration

- Used by all agents (Nova, Emil, Ivan, Lola) to store and retrieve data
- Central to the task management system for storing task results
- Maintains session state across multiple interactions
- Provides persistence for the entire application
- Enables cross-agent communication through shared data

Workflow

1. System initializes the knowledge base, potentially loading existing data
2. Agents store and retrieve data using the knowledge base API
3. Each interaction is logged in the current session
4. Task results are categorized and stored for future reference
5. Agent-specific context is provided based on the agent's role
6. Sessions are archived when they become old
7. Data is regularly saved to disk for persistence

Implementation Notes

- Uses asyncio for asynchronous operations
- Implements categorization through a special "**categories**" dictionary
- Provides helper methods for common operations
- Uses JSON for persistence format
- Creates automatic backups on hourly intervals
- Keeps session histories organized by session ID
- Implements search functionality through recursive key/value matching