EideticEngine: An Adaptive Cognitive Architecture Integrating Multi-Level Memory, Structured Orchestration, and Meta-Cognition for Advanced LLM Agents

Jeffrey Emanuel

April 13, 2025

Abstract

Large Language Models (LLMs) form the reasoning core of increasingly sophisticated autonomous agents. However, unlocking their full potential for complex, long-horizon tasks requires architectures that transcend reactive loops and shallow memory. We present **EideticEngine**, a novel cognitive architecture designed to imbue LLM agents with robust memory, structured planning, and adaptive self-management capabilities inspired by cognitive science.

EideticEngine integrates two key components: 1) A **Unified Memory System (UMS)**, a persistent, multi-level cognitive workspace implemented on an optimized asynchronous database, featuring distinct memory types (working, episodic, semantic, procedural), rich metadata (importance, confidence, TTL), explicit typed linking, hybrid search (semantic, keyword, relational), and integrated workflow tracking (actions, artifacts, thoughts). 2) An **Agent Master Loop (AML)**, an adaptive orchestrator that directs an LLM using the UMS. The AML manages structured, dependency-aware plans (`PlanStep`), dynamically assembles comprehensive context from the UMS, handles errors resiliently, and crucially, orchestrates **agent-driven meta-cognition**.

Through specific UMS tools, the agent actively reflects on its performance (`generate_reflection`), consolidates knowledge (`consolidate_memories`), promotes memories between cognitive levels (`promote_memory_level`), manages its attentional focus (`optimize_working_memory`, `auto_update_focus`), and even manages distinct reasoning threads (`create_thought_chain`). Furthermore, EideticEngine incorporates an **adaptive control layer** where meta-cognitive parameters (e.g., reflection frequency) are dynamically adjusted based on real-time operational statistics (`compute_memory_statistics`, `_adapt_thresholds`).

We provide detailed simulations and analysis demonstrating EideticEngine's ability to autonomously navigate complex analytical and creative tasks, exhibiting structured learning, error recovery, and adaptive behavior. EideticEngine represents a significant architectural advance, providing essential infrastructure for developing more capable, persistent, and introspective general-purpose AI agents.

1 Introduction: Towards Cognitive Autonomy in LLM Agents

The remarkable generative and reasoning abilities of Large Language Models (LLMs) [1, 2] have catalyzed the development of autonomous agents aimed at complex problem-solving. Yet, the transition from impressive demonstrations to robust, reliable agents capable of sustained, adaptive operation across diverse, long-horizon tasks remains a formidable challenge [3]. Current agent frameworks often grapple with fundamental limitations:

- **Memory Persistence & Structure:** Reliance on ephemeral prompt context or simplistic memory buffers (e.g., chat history, basic vector stores) hinders long-term learning, recall of structured knowledge, and understanding of temporal or causal relationships [4, 5].
- **Planning & Execution:** Ad-hoc or reactive planning struggles with complex sequences, interdependencies, and resource management. Lack of explicit dependency tracking leads to brittleness and execution failures [6, 7].
- **Adaptation & Learning:** Most agents lack mechanisms for reflecting on past actions, learning from errors (beyond simple retries), synthesizing experiences into general knowledge, or adapting their strategies based on performance [8].
- **Cognitive Coherence:** Agents often lack a unified internal state representation that integrates perception, memory, reasoning, planning, and action within a consistent framework.

To address these critical gaps, we introduce **EideticEngine**, a comprehensive cognitive architecture designed explicitly for orchestrating advanced LLM agents. EideticEngine is not merely an LLM wrapper or a collection of tools; it is an integrated system built upon two deeply interconnected components:

1. **The Unified Memory System (UMS):** A persistent, multi-layered cognitive substrate. Inspired by human memory models [9, 10], the UMS provides distinct but interconnected stores for working, episodic, semantic, and procedural memory. Implemented using an optimized asynchronous SQLite backend (`aiosqlite`) with a detailed relational schema, it tracks not only memories with rich metadata but also the agent's entire operational history: workflows, hierarchical actions (with explicit dependencies), generated artifacts, and structured thought chains. It incorporates hybrid search mechanisms (vector, FTS5, relational filtering) and supports dynamic memory evolution through linking, consolidation, and promotion.
2. **The Agent Master Loop (AML):** An adaptive control loop that orchestrates the agent's interaction with the UMS and the external world (via tools dispatched by an MCPClient). The AML directs a core LLM (e.g., Claude 3.7 Sonnet) by providing it with dynamically assembled, multi-faceted context drawn from the UMS. It manages structured plans (`PlanStep` objects) featuring explicit dependency tracking (`depends_on` fields validated via `_check_prerequisites`). Crucially, the AML empowers the LLM agent to engage in **meta-cognition** by providing specific UMS tools (`generate_reflection`, `consolidate_memories`, `promote_memory_level`, `update_memory`, etc.) that allow the agent to analyze its own performance, synthesize knowledge, manage its memory state, and refine its strategies. This

meta-cognitive cycle is further enhanced by an **adaptive control mechanism** (`_adapt_thresholds`) that dynamically adjusts the frequency of reflection and consolidation based on runtime statistics computed from the UMS (`compute_memory_statistics`), enabling the agent to self-regulate its cognitive load.

EideticEngine's core hypothesis is that by tightly integrating a structured, cognitive-inspired memory system with an adaptive, meta-cognitively capable control loop, we can create LLM agents that exhibit significantly greater autonomy, robustness, learning capability, and effectiveness on complex, real-world tasks. This paper details the architecture, illustrates its operation through granular simulations, and discusses its implications for the future of general-purpose AI agents.

2 Related Work: Building on and Departing From Existing Paradigms

EideticEngine differentiates itself from several established lines of research:

- **Standard LLM Agent Frameworks (LangChain, LlamaIndex, etc.):** While providing valuable abstractions for tool use and basic memory (often vector stores or simple buffers), these frameworks typically lack: (i) a deeply integrated, multi-level cognitive memory model with explicit linking and dynamic evolution (promotion, consolidation); (ii) structured planning with robust dependency checking enforced by the loop; (iii) agent-driven meta-cognitive tools for reflection and knowledge synthesis; (iv) adaptive control mechanisms adjusting agent behavior based on performance. EideticEngine offers a more opinionated and comprehensive *cognitive architecture* rather than a flexible toolkit. (References like [11] might fit here conceptually, though not explicitly LangChain).
- **Early Autonomous Agents (AutoGPT [6], BabyAGI):** These pioneering efforts demonstrated the potential of LLM loops but suffered from unreliable planning, simplistic memory (often just text files or basic vector stores), lack of error recovery, and significant coherence issues over longer runs. EideticEngine addresses these directly with structured UMS, planning, dependency checks, and meta-cognition.
- **Memory-Augmented LLMs (MemGPT [5], RAG [12]):** These focus on enhancing LLM capabilities by providing access to external or specialized memory during generation. EideticEngine complements this by providing a persistent, structured *internal* memory system that tracks the agent's *own* experiences, thoughts, actions, and synthesized knowledge, enabling longitudinal learning and self-understanding beyond immediate context retrieval. The UMS serves as the agent's evolving world model and operational history. Other related work includes [4, 13].
- **LLM Planning & Reasoning Techniques (ReAct [7], Chain-of-Thought [14], Tree-of-Thoughts [15]):** These enhance the LLM's internal reasoning process, often within a single prompt or short interaction sequence. EideticEngine operates at a higher architectural level, orchestrating these reasoning steps within a persistent framework. It externalizes the plan, memory, and workflow state into the UMS, allowing for much longer, more complex tasks, error recovery across loops, and persistent learning that influences future reasoning cycles. EideticEngine's

thought_chains provide a structured way to manage and persist complex reasoning paths generated potentially using these techniques. [16] discusses structured plan representation.

- **Classical Cognitive Architectures (SOAR [17], ACT-R [18, 19], OpenCog):** These offer rich, theoretically grounded models of cognition, often based on symbolic rule systems or specialized memory structures. While highly influential, they are typically challenging to integrate directly with the sub-symbolic nature and generative flexibility of LLMs and are rarely deployed as practical, general-purpose agents. EideticEngine adopts key *principles* from cognitive architectures (e.g., memory levels [9, 10, 20, 21], relevance decay, meta-cognition [22]) but implements them within a practical, LLM-native framework built for autonomous task execution and tool use, leveraging the LLM itself for high-level reasoning and meta-cognitive tasks.

- **Meta-Reasoning and Reflection Research [23, 8]:** While the importance of meta-cognition is recognized, few practical LLM agent systems incorporate explicit, agent-driven reflection and knowledge consolidation loops tied to performance metrics. EideticEngine operationalizes this through dedicated tools (`generate_reflection`, `consolidate_memories`) and, significantly, makes the *frequency* of these operations adaptive (`_adapt_thresholds`) based on runtime UMS statistics, creating a dynamic feedback loop for self-improvement.

3 The Unified Memory System (UMS): A Cognitive Substrate for Agents

The foundation of the EideticEngine architecture is the Unified Memory System (UMS), a persistent and structured cognitive workspace designed to move beyond the limitations of simple memory buffers or isolated vector stores. It serves not just as a repository of information, but as an active substrate for the agent's learning, reasoning, and operational history. Its novelty and power stem from the deep integration of several key design principles:

3.1 Multi-Level Cognitive Memory Hierarchy

Inspired by human memory models, the UMS implements distinct but interconnected memory levels (`MemoryLevel` enum: `WORKING`, `EPISODIC`, `SEMANTIC`, `PROCEDURAL`), stored within the `memories` table and differentiated by the `memory_level` column. This isn't just a label; it dictates default behaviors and enables sophisticated management strategies:

- **Working Memory:** Explicitly managed outside the main `memories` table, residing in the `cognitive_states` table as a list of `memory_ids` (`working_memory` JSON field). It's capacity-constrained (`MAX_WORKING_MEMORY_SIZE`) and managed by tools like `optimize_working_memory` which uses relevance scoring (`_compute_memory_relevance`) and strategies (like 'diversity') to maintain a focused attentional set. `auto_update_focus` further refines this by identifying the most salient item within this active set. Compare with [24].

- **Episodic Memory:** Directly captures agent experiences. Records associated with specific `actions` (via `action_id` FK in `memories`), `thoughts` (`thought_id` FK), or `artifacts` (`artifact_id` FK) default to this level. They often have shorter default `ttl` values (defined in `DEFAULT_TTL`), reflecting their time-bound nature. Tools like record_action_start and record_artifact automatically create linked episodic memories (memory_type=ACTION_LOG or ARTIFACT_CREATION). See [9, 20, 21].

- **Semantic Memory:** Represents generalized knowledge, facts, insights, summaries, or stable profiles (e.g., character_profile, story_arc). These often result from explicit store_memory calls with level=semantic, or crucially, from meta-cognitive processes like consolidate_memories or successful promote_memory_level operations acting on episodic data. They typically have longer default ttl. See [9].
- **Procedural Memory:** Encodes learned skills or multi-step procedures (memory_type=SKILL or PROCEDURE). This level is primarily populated via promote_memory_level from highly accessed, high-confidence semantic memories that fit the procedural type criteria, representing a form of skill acquisition within the system. It has the longest default ttl.

3.2 Rich Metadata and Cognitive Attributes

Each memory entry in the memories table is far more than just content. It carries crucial metadata enabling cognitive processing:

- **Importance & Confidence:** Explicit REAL fields (importance, confidence) allow the agent (or LLM via store_memory/update_memory) to assign subjective value and certainty to information, critical for prioritization and belief revision.
- **Temporal Dynamics:** created_at, updated_at, last_accessed (Unix timestamps) combined with access_count and ttl enable relevance calculations (_compute_memory_relevance function, incorporating MEMORY_DECAY_RATE) and automatic expiration (delete_expired_memories). This gives the memory system temporal dynamics often missing in static knowledge bases.
- **Provenance & Context:** Foreign keys (action_id, thought_id, artifact_id) directly link memories to their operational origins. The source field tracks external origins (tool names, filenames), and the context JSON field stores arbitrary metadata about the memory's creation circumstances, providing rich contextual grounding.
- **Flexible Categorization:** Besides memory_level and memory_type, memories have a JSON tags field, allowing for multi-dimensional categorization and retrieval using the custom json_contains_all SQLite function within query_memories.

3.3 Structured Associative Memory Graph

Unlike systems solely reliant on vector similarity, the UMS builds an explicit, typed graph of relationships via the memory_links table:

- **Typed Links:** The LinkType enum defines a rich vocabulary for relationships (e.g., RELATED, CAUSAL, SUPPORTS, CONTRADICTS, HIERARCHICAL, SEQUENTIAL, REFERENCES). This allows the agent to represent and reason about structured knowledge beyond simple proximity in embedding space.

- **Explicit Creation:** The `create_memory_link` tool allows the agent or LLM to deliberately assert relationships between memories based on its reasoning.
- **Automated Linking:** The `_run_auto_linking` background process, triggered after memory creation (`store_memory`) or artifact recording, uses semantic similarity (`_find_similar_memories`) to *suggest and create* probable `RELATED` or contextually inferred links (e.g., `SUPPORTS` if linking fact-to-insight), bootstrapping the knowledge graph.
- **Graph Traversal:** The `get_linked_memories` tool enables navigation of this graph structure, retrieving neighbors based on direction (`incoming`, `outgoing`, `both`) and `link_type`, providing structured context retrieval.

3.4 Deep Integration with Workflow & Reasoning

The UMS is not separate from the agent's operational layer; it's intrinsically linked:

- **Action-Memory Coupling:** Actions (`record_action_start/completion`) automatically generate corresponding `Episodic` memories (`type=ACTION_LOG`). Memories can be explicitly linked back to the actions that generated or used them (`action_id` FK).
- **Thought-Memory Coupling:** Thoughts (`record_thought`) can be directly linked to relevant memories (`relevant_memory_id` FK in `thoughts`), and important thoughts (e.g., goals, decisions, summaries) automatically generate linked `Semantic` or `Episodic` memories (`type=REASONING_STEP`).
- **Artifact-Memory Coupling:** Recording artifacts (`record_artifact`) creates linked `Episodic` memories (`type=ARTIFACT_CREATION`), and memories can reference artifacts (`artifact_id` FK).
- **Comprehensive Traceability:** The interconnected schema (`workflows`, `actions`, `artifacts`, `thought_chains`, `thoughts`, `memories`, `memory_links`, `memory_operations`) provides an end-to-end, auditable record of the agent's perception, reasoning, action, and learning history. Tools like `generate_workflow_report` leverage this structure.

3.5 Hybrid & Configurable Retrieval

The UMS offers multiple, complementary retrieval mechanisms catering to different information needs:

- **Semantic Search (`search_semantic_memories`):** Leverages vector embeddings (`embeddings` table, `_find_similar_memories`, `cosine_similarity`) for finding conceptually related information, filtered by core metadata (workflow, level, type, TTL).
- **Keyword & Attribute Search (`query_memories`):** Utilizes SQLite's FTS5 virtual table (`memory_fts`, indexing content, description, reasoning, tags) for fast keyword matching, combined with precise SQL filtering on any metadata attribute (importance, confidence, tags via `json_contains_all`, timestamps, etc.). Allows sorting by various fields including calculated `relevance`.

- **Hybrid Search (`hybrid_search_memories`):** Powerfully combines semantic similarity scores with keyword/attribute relevance scores (derived from `_compute_memory_relevance`) using configurable weights (`semantic_weight`, `keyword_weight`). This allows retrieval ranked by a blend of conceptual meaning and factual importance/recency/confidence, often yielding more pertinent results than either method alone.

- **Direct & Relational Retrieval:** `get_memory_by_id` provides direct access, while `get_linked_memories` allows navigation based on the explicit graph structure. `get_action_details`, `get_artifacts`, `get_thought_chain` retrieve operational context.

3.6 Mechanisms for Knowledge Evolution

The UMS incorporates processes for refining and structuring knowledge over time:

- **Consolidation (`consolidate_memories`):** Explicitly uses LLM reasoning (via `get_provider`) to synthesize multiple, often `Episodic`, memories into more abstract `Semantic` forms (summaries, insights) or `Procedural` forms (if source memories describe actions/outcomes). The results are stored as new memories and linked back to the sources, actively structuring the knowledge base.

- **Promotion (`promote_memory_level`):** Implements a heuristic-based mechanism for memories to "graduate" levels (e.g., Episodic -> Semantic, Semantic -> Procedural) based on sustained usage (`access_count` threshold) and high `confidence`, mimicking memory strengthening and generalization. Thresholds are configurable per promotion step.

- **Reflection Integration (`generate_reflection`):** While the reflection content is stored in the `reflections` table, the process analyzes `memory_operations` logs, providing insights that can lead the agent (via the AML) to `update_memory`, `create_memory_link`, or trigger further `consolidate_memories` calls, thus driving knowledge refinement based on operational analysis.

3.7 Robust Implementation Details

- **Asynchronous Design:** Use of `aiosqlite` ensures the UMS doesn't block the main agent loop during database I/O. Background tasks (`_run_auto_linking`) further enhance responsiveness.

- **Optimized SQL:** Leverages SQLite features like WAL mode, indexing, FTS5, memory mapping (`PRAGMA` settings), and custom functions (`compute_memory_relevance`, `json_contains_*`) for performance.

- **Structured Data Handling:** Consistent use of Enums (`MemoryLevel`, `MemoryType`, `LinkType`, `ActionStatus`, etc.) ensures data integrity. Careful serialization/deserialization (`MemoryUtils.serialize/deserialize`) handles complex data types and prevents errors, including handling potential `MAX_TEXT_LENGTH` overflows gracefully.

- **Comprehensive Auditing:** The `memory_operations` table logs virtually every significant interaction with the UMS, providing deep traceability for debugging and analysis.

4 The Agent Master Loop (AML): Adaptive Orchestration and Meta-Cognition

While the UMS provides the cognitive substrate, the Agent Master Loop (AML) acts as the central executive, orchestrating the agent's perception-cognition-action cycle to achieve complex goals. It transcends simple reactive loops by implementing structured planning, sophisticated context management, robust error handling, and, critically, adaptive meta-cognitive control, leveraging the UMS and an LLM reasoning core (e.g., Claude 3.7 Sonnet).

4.1 Structured, Dependency-Aware Planning

A cornerstone of the AML is its departure from ad-hoc planning. It manages an explicit, dynamic plan within its state (`AgentState.current_plan`), represented as a list of `PlanStep` Pydantic objects.

- **Plan Representation (`PlanStep`):** Each step encapsulates not just a `description`, but also its `status` ('planned', 'in_progress', 'completed', 'failed', 'skipped'), `assigned_tool` and `tool_args` (optional), `result_summary`, and crucially, a `depends_on` list containing the `action_id`s of prerequisite steps.
- **LLM-Driven Plan Generation:** The AML prompts the LLM (`_call_agent_llm`) not only for the next action but also for an "Updated Plan" block within its reasoning text. The AML parses this structured JSON (`re.search` for the specific block format) and validates it against the `PlanStep` model. This allows the LLM to dynamically modify the entire strategy based on new information or errors, rather than just deciding the immediate next step.
- **Dependency Enforcement (`_check_prerequisites`):** Before executing any `PlanStep` that involves a tool call, the `_execute_tool_call_internal` function extracts the `depends_on` list and calls `_check_prerequisites`. This helper function queries the UMS (`get_action_details`) to verify that *all* listed prerequisite action IDs have a status of `completed`. If dependencies are unmet, execution is **blocked**, an error detailing the unmet dependencies is logged (`state.last_error_details`), and the `state.needs_replan` flag is set, forcing the LLM to reconsider the plan in the next loop. This mechanism prevents cascading failures common in agents without explicit dependency management.
- **Heuristic Plan Update (`_update_plan`):** If the LLM *doesn't* provide a valid updated plan, this fallback mechanism provides basic plan progression. It marks the current step as 'completed' or 'failed' based on the last action's success, potentially removes the completed step, and may insert a generic "Analyze result/failure" step if the plan becomes empty or an error occurred. This ensures the loop doesn't stall but prioritizes LLM-driven planning when available.

4.2 Multi-Faceted Context Assembly (_gather_context)

The AML recognizes that effective LLM reasoning requires rich context beyond simple chat history. The `_gather_context` function actively probes the UMS to construct a comprehensive snapshot:

- **Operational State:** Includes `current_loop`, `consecutive_errors`, `last_error_details`, the active `workflow_id`, `context_id`, and the `current_plan`.
- **Working Memory:** Queries `get_working_memory` to retrieve the IDs and summaries of memories currently in the agent's attentional focus (`cognitive_states` table).
- **Proactive Goal-Relevant Memory:** Performs a `hybrid_search_memories` query using the description of the *current plan step* to proactively fetch memories semantically or lexically related to the immediate task, anticipating information needs.
- **Procedural Knowledge:** Executes another `hybrid_search_memories` query specifically filtered for `memory_level=procedural` using the plan step description to find relevant "how-to" knowledge.
- **Core History Summary:** Uses `get_workflow_context` to fetch recent actions, important memories (by importance score), and key thoughts (goals, decisions, summaries) from the primary thought chain.
- **Relational Context:** Optionally uses `get_linked_memories` starting from a highly relevant memory (e.g., the top working memory item) to provide insight into the local knowledge graph structure.
- **Meta-Cognitive Feedback:** Includes the summary (`state.last_meta_feedback`) from the last reflection or consolidation cycle.
- **Context Compression:** Monitors the estimated token count (`_estimate_tokens_anthropic`) of the assembled context. If it exceeds `CONTEXT_MAX_TOKENS_COMPRESS_THRESHOLD`, it uses `summarize_text` to compress less critical parts (e.g., detailed recent action logs) to stay within LLM limits while preserving key information.

4.3 Adaptive Meta-Cognitive Control (_run_periodic_tasks, _adapt_thresholds)

This is perhaps the most innovative aspect of the AML. It doesn't just execute tasks; it monitors and regulates its own cognitive processes:

- **Triggering Meta-Cognition:** `_run_periodic_tasks` is executed each loop. It checks various conditions:
  - Action Counts vs. *Dynamic* Thresholds: Compares `state.successful_actions_since_reflection` against `state.current_reflection_threshold` and `state.successful_actions_since_consolidation` against `state.current_consolidation_threshold`.
  - State Flags: Checks `state.needs_replan` (often set after errors or significant feedback).

  - Periodic Intervals: Checks `state.loops_since_*` counters against constants (`OPTIMIZATION_LOOP_INTERVAL`, `MEMORY_PROMOTION_LOOP_INTERVAL`, `STATS_ADAPTATION_INTERVAL`, `MAINTENANCE_INTERVAL`).

- **Executing Meta-Cognitive Tools:** When triggered, it schedules and executes UMS tools like `generate_reflection` (cycling through types: summary, progress, gaps, strengths, plan), `consolidate_memories`, `optimize_working_memory`, `auto_update_focus`, `promote_memory_level` (via `_check_and_trigger_promotion`), and `delete_expired_memories`.

- **Feedback Loop:** The output of reflection/consolidation is captured in `state.last_meta_feedback` and included in the *next* context assembly, allowing the LLM to react to the agent's self-assessment. Successful reflection or consolidation also often sets `state.needs_replan=True`, ensuring the LLM considers the new insights.

- **Adaptive Threshold Adjustment (`_adapt_thresholds`):** This crucial function makes the meta-cognition *adaptive*. Triggered periodically by `_run_periodic_tasks` after `compute_memory_statistics` runs, it analyzes key UMS stats (e.g., ratio of episodic memories, tool failure rates stored in `state.tool_usage_stats`). Based on heuristics (e.g., "if episodic count > high watermark, decrease consolidation threshold"; "if error rate > high watermark, decrease reflection threshold"), it **modifies `state.current_reflection_threshold` and `state.current_consolidation_threshold`** (within `MIN/MAX` bounds). This allows the agent to automatically become more reflective when struggling or consolidate knowledge more frequently when rapidly accumulating raw experience, without external tuning.

4.4 Robust Execution and Error Handling (`_execute_tool_call_internal`)

The AML provides a resilient execution layer:

- **Tool Server Discovery:** Uses `_find_tool_server` to locate active servers providing requested tools via the `MCPClient`.

- **Action Recording:** Automatically wraps significant tool calls (excluding meta-actions, pure retrievals) with `record_action_start` and `record_action_completion`, ensuring operational history is captured in the UMS. It associates the correct `action_id` with results and dependencies.

- **Dependency Recording:** After recording an action start, it calls `add_action_dependency` for all prerequisites listed in the corresponding `PlanStep`.

- **Error Tracking:** Catches tool execution errors (including `ToolError`, `ToolInputError`, and unexpected exceptions), updates `state.last_error_details`, increments `state.consecutive_error_count`, and sets `state.needs_replan=True`. It checks for the specific dependency failure condition (status code 412) to inform replanning. If `MAX_CONSECUTIVE_ERRORS` is reached, it halts the loop and marks the workflow as failed.

- **Background Task Management:** Uses `_start_background_task` to run non-blocking operations like `_run_auto_linking` or `_check_and_trigger_promotion` concurrently after relevant main actions succeed, improving responsiveness. Includes cleanup (`_cleanup_background_tasks`) on shutdown.

4.5 Thought Chain Management

The AML actively manages the agent's reasoning focus:

- It tracks the `state.current_thought_chain_id`.
- When the LLM calls `create_thought_chain`, the AML's `_handle_workflow_side_effects` updates the `current_thought_chain_id` to the newly created one.
- It automatically injects the `current_thought_chain_id` into `record_thought` calls if the LLM doesn't specify one, ensuring thoughts are logged contextually. This allows the LLM to easily switch between reasoning threads simply by targeting its `record_thought` calls.

5 The Ultimate MCP Client: Facilitating Cognitive Orchestration

The EideticEngine architecture, while powerful conceptually, relies on a robust communication and interaction layer to bridge the Agent Master Loop (AML) with the Unified Memory System (UMS) and other potential external tools. The **Ultimate MCP Client** (`mcp_client.py`) provides this critical "glue," offering a feature-rich environment specifically designed to support the complex needs of advanced cognitive agents like EideticEngine. Its design choices significantly enable and simplify the implementation of EideticEngine's core functionalities.

5.1 Unified Access to Distributed Capabilities

EideticEngine's power comes from leveraging diverse tools hosted potentially across different servers (UMS server, corpus search server, web browser server, etc.). The MCP Client abstracts this complexity:

- **Server Management (`ServerManager`, `ServerConfig`):** It discovers (`discover_servers` using filesystem, registry, mDNS, *and* active port scanning), configures, connects (`connect_to_server`), monitors (`ServerMonitor`), and manages the lifecycle of multiple MCP servers (both STDIO and SSE types via `RobustStdioSession` and standard `sse_client` respectively). This allows the AML to seamlessly access tools without needing to know their physical location or connection type.
- **Centralized Tool/Resource Registry:** The `ServerManager` aggregates tools (`tools`), resources (`resources`), and prompts (`prompts`) advertised by all connected servers into unified dictionaries. This allows the AML (`_call_agent_llm`) to present a single, comprehensive list of available capabilities to the LLM, simplifying the decision-making prompt. It uses `format_tools_for_anthropic` to sanitize names and prepare schemas specifically for the LLM API.
- **Intelligent Routing:** When the AML decides to execute a tool (`execute_tool`), the client implicitly routes the request to the correct server based on the tool's registration (`MCPTool.server_name`).

5.2 Robust Communication and Error Handling

Interacting with potentially unreliable external processes or network services requires resilience:

- **Asynchronous Architecture (`asyncio`, `httpx`, `aiosqlite`):** The client is built entirely on Python's `asyncio`, ensuring that communication with multiple servers, background tasks (like discovery or monitoring), and potentially slow tool executions do not block the main agent loop (AML).
- **Specialized STDIO Handling (`RobustStdioSession`):** Recognizing the fragility of STDIO communication, the client implements a custom session handler. This handler directly resolves futures upon receiving responses (`_read_and_process_stdout_loop`) rather than relying solely on queues, potentially improving responsiveness. It includes logic to filter noisy non-JSON output often emitted by scripts and manages process lifecycles robustly.
- **STDIO Safety (`safe_stdout`, `get_safe_console`, `StdioProtectionWrapper`):** Crucially, the client incorporates multiple layers of protection to prevent accidental `print` statements or other stdout pollution from corrupting the JSON-RPC communication channel used by STDIO servers. This is vital for stability when integrating diverse tools.
- **Retry Logic & Circuit Breaking (`retry_with_circuit_breaker` decorator):** The `execute_tool` method incorporates automatic retries with exponential backoff and a simple circuit breaker mechanism (based on `ServerMetrics.error_rate`), improving resilience against transient network issues or server hiccups without overwhelming a failing server.
- **Graceful Shutdown (`close`, signal handling):** The client implements proper signal handling (SIGINT/SIGTERM) and cleanup routines (`atexit_handler`, `close` methods) to ensure server processes are terminated, connections are closed, caches are flushed, and state is saved upon exit.

5.3 Enabling Advanced Agent Features

The client provides specific features that directly support EideticEngine's cognitive capabilities:

- **Streaming Support (WebSockets & Internal):** The `process_streaming_query` method and the WebSocket endpoint (`/ws/chat`) allow for real-time streaming of LLM responses and tool status updates, crucial for interactive use cases and providing immediate feedback during long-running agent tasks. The internal stream processing logic (`_process_stream_event`) handles partial JSON accumulation for tool inputs.
- **Tool Result Caching (`ToolCache`):** Implements both in-memory and disk-based caching (`diskcache`) for tool results, with configurable TTLs (`cache_ttl_mapping`) potentially derived from tool categories. This significantly speeds up repetitive queries (e.g., retrieving the same document) and reduces load on external tools/APIs. It also includes basic dependency invalidation (`invalidate_related`).

- **Conversation Management (`ConversationGraph`, `ConversationNode`):** Moves beyond linear chat history, implementing a branching conversation structure. This allows the agent (or user) to explore different reasoning paths or "fork" the state (`cmd_fork`, `create_fork`), crucial for complex problem-solving or experimentation. State includes messages and the model used for that node. Persistence is handled via async saving/loading (`save`/`load` methods) to JSON files.

- **Context Optimization Interface (`cmd_optimize`, `auto_prune_context`):** Provides both manual and automatic mechanisms (`process_query` calling `auto_prune_context`) to summarize long conversation histories using a designated LLM (`summarization_model`), helping to manage context window limitations.

- **Dynamic Prompting (`cmd_prompt`, `apply_prompt_to_conversation`):** Allows pre-defined prompt templates stored on MCP servers (`ListPromptsResult`, `GetPromptResult`) to be fetched and applied to the current conversation context, facilitating standardized interactions or persona adoption.

- **Observability (OpenTelemetry):** Integration with OpenTelemetry (`tracer`, `meter`, specific counters/histograms) provides hooks for detailed monitoring of client performance, tool execution latency, and request volumes, essential for understanding and optimizing complex agent behavior in production environments.

- **Configuration Flexibility (`Config`, `cmd_config`):** Uses YAML for configuration (`config.yaml`), allowing easy management of API keys, model preferences, server definitions, discovery settings (including filesystem paths, mDNS enable/disable, *and port scanning parameters*), caching behavior, and more. Supports loading from environment variables (`load_dotenv`).

- **Enhanced Discovery (mDNS & Port Scanning):** Beyond static configuration and filesystem discovery, it actively discovers servers on the local network using Zeroconf/mDNS (`ServerRegistry.start_local_discovery`) and configurable *active port scanning* (`_discover_port_scan`, `_probe_port`), making it easier to connect to dynamically available local tools or UMS instances.

- **Platform Adaptation (`adapt_path_for_platform`):** Includes specific logic to handle configuration differences between platforms, particularly translating Windows paths found in imported Claude Desktop configurations into Linux/WSL equivalents, enhancing cross-platform usability.

5.4 Developer Experience and Usability

- **Interactive CLI & Web UI:** Offers both a powerful interactive command-line interface (using `typer` and `rich` for enhanced display, history, and completion) and a modern reactive Web UI (via `FastAPI` and WebSockets), catering to different user preferences for interacting with the agent and managing servers.

- **API Server (`FastAPI`):** Exposes comprehensive REST endpoints (`/api/...`) and a WebSocket endpoint (`/ws/chat`) allowing programmatic control over the client, agent execution, server management, and conversation state. This enables integration with other applications or orchestration systems.

- **Clear Status & Monitoring:** Provides immediate feedback via `rich.Status`, progress bars (`_run_with_progress`), a live dashboard (`cmd_dashboard`, `generate_dashboard_renderable`), and detailed server status commands (`cmd_serverstatus`).

6 The LLM Gateway Server: An Ecosystem of Tools for Cognitive Agents

The EideticEngine architecture relies not only on its internal logic (AML) and its cognitive substrate (UMS) but also on a rich ecosystem of external capabilities accessible via the Model Context Protocol (MCP). The **Ultimate MCP Client** (`mcp_client.py`) acts as the bridge, connecting the AML to an **LLM Gateway Server** instance. This server, designed by the same author, hosts the UMS tools (implemented in `cognitive_and_agent_memory.py`) alongside a powerful suite of complementary tools, significantly expanding the agent's operational repertoire and enabling more complex, real-world workflows.

6.1 Architecture: UMS as a Tool Suite within a Larger Gateway

It's crucial to understand that the **UMS is implemented as a collection of tools within the broader LLM Gateway MCP Server**. The AML, via the `UltimateMCPClient`, interacts with the UMS not through direct database calls, but by invoking specific `unified_memory:*` tools registered on the Gateway server. This modular design offers several advantages:

- **Decoupling:** The agent's core logic (AML) is decoupled from the specific implementation details of the memory system.
- **Extensibility:** New memory features or other functionalities can be added to the Gateway server as new tools without requiring changes to the AML itself.
- **Standardized Interaction:** All interactions (memory, file access, web browsing, LLM calls) occur through the unified MCP interface managed by the client.

6.2 Core LLM Gateway Capabilities (Beyond UMS)

The Gateway server provides foundational services that the EideticEngine agent heavily relies upon, often invoked transparently by the UMS tools or directly by the AML:

- **Multi-Provider LLM Access (`completion.py`, `providers/`):** Offers a standardized interface (`generate_completion`, `chat_completion`, `stream_completion`) to various LLM backends (OpenAI, Anthropic, Gemini, DeepSeek, OpenRouter). Handles API key management, request formatting, response parsing, error handling, and crucial cost/token tracking (`COST_PER_MILLION_TOKENS`, `ModelResponse`). This allows the AML and UMS tools (like `consolidate_memories`, `generate_reflection`) to easily leverage different LLMs.
- **Embedding Service (`vector/embeddings.py`):** Provides embedding generation (`get_embedding`, `get_embeddings`) using configurable models (defaulting to `text-embedding-3-small`), including local caching (`EmbeddingCache`) to reduce redundant API calls. This service is used extensively by the UMS `store_memory` tool and search functions.

- **Vector Database Service (`vector/vector_service.py`):** Manages vector collections, currently supporting ChromaDB (`chromadb`) if available, or a fallback in-memory index (`VectorCollection` using `numpy` or `hnswlib`). This underlies the UMS semantic search capabilities (`search_semantic_memories`, `hybrid_search_memories`).
- **Caching Service (`cache/`):** Implements sophisticated caching strategies (`ExactMatchStrategy`, `SemanticMatchStrategy`, `TaskBasedStrategy`) with persistence (`CachePersistence`, `diskcache`) for arbitrary tool results, significantly reducing latency and cost for repeated operations. The `with_cache` decorator is used by many UMS and Gateway tools.
- **Prompt Management (`prompts/`):** Includes a `PromptRepository` and `PromptTemplate` system (using Jinja2) allowing pre-defined, reusable prompts to be stored, retrieved, and rendered, facilitating standardized interactions and complex prompt construction (`render_prompt_template`).

6.3 Synergistic Tools Enhancing EideticEngine's Capabilities

Beyond the core UMS tools, the LLM Gateway server hosts other tool suites that the EideticEngine agent can leverage, often in conjunction with its memory:

- **Advanced Extraction Tools (`extraction.py`):**
  - `extract_json`: Extracts structured JSON, optionally validating against a schema. Crucial for processing tool outputs or structured text stored in memory (`ArtifactType.JSON`, `MemoryType.JSON`).
  - `extract_table`: Parses tables from text into formats like JSON lists or Markdown. Essential for analyzing data stored in `ArtifactType.TABLE` or `MemoryType.TEXT`.
  - `extract_key_value_pairs`: Pulls out key-value data, useful for populating `Semantic` memories or analyzing configuration-like text artifacts.
  - `extract_semantic_schema`: *Infers* a schema from unstructured text, potentially useful for the agent to understand data before storing it structurally in the UMS or deciding how to process an `ArtifactType.TEXT`.
  - `extract_code_from_response`: Cleans up LLM code generation outputs before storing them as `ArtifactType.CODE` or `MemoryType.CODE`.
- **Document Processing Tools (`document.py`):**
  - `chunk_document`: Offers multiple strategies (semantic, token, paragraph) to break down large documents (e.g., from `read_file` or a large `MemoryType.TEXT`) before feeding them to LLMs via other tools (like `summarize_document`).
  - `summarize_document`: Can summarize text retrieved from memory, artifacts, or files, potentially storing the result back into the UMS as a `MemoryType.SUMMARY`.
  - `extract_entities`, `generate_qa_pairs`: Useful for analyzing document content stored as artifacts or memories, generating new factual memories (`MemoryType.FACT`) or questions (`MemoryType.QUESTION`) to store in the UMS.

- **Secure Filesystem Tools (`filesystem.py`):**
  - Provides secure, sandboxed access to the local filesystem (within `allowed_directories`). The agent can `read_file` into memory, `write_file` from memory content, `list_directory` to understand context, `search_files` for relevant information, and create `Artifact` records (`record_artifact`) pointing to these files. Crucially, `validate_path` ensures operations stay within safe boundaries, and deletion protection (`_check_protection_heuristics`) adds a safety layer.
- **Local Text Processing Tools (`use_local_text_tools.py`):**
  - Offers offline text manipulation via command-line tools (`rg`, `awk`, `sed`, `jq`). An agent could retrieve text from a UMS memory (`get_memory_by_id`), process it locally using `run_jq` (if JSON) or `run_sed`, and then store the modified result back using `update_memory` or `store_memory`, potentially reducing LLM costs for simple transformations. Security validation (`_validate_tool_arguments`) prevents dangerous command injections.
- **Web Browser Automation Tools (`browser_automation.py`):**
  - Enables the agent to interact with the live web via Playwright. This dramatically expands the agent's capabilities beyond its internal memory and local files. It can `browser_navigate` to URLs stored in `MemoryType.URL` or `ArtifactType.URL`, `browser_get_text` to scrape information and store it as `MemoryType.OBSERVATION`, `browser_click` or `browser_type` to interact with web forms, and `browser_screenshot` or `browser_pdf` to create `ArtifactType.IMAGE` or `ArtifactType.FILE` records linked to the browsing action in the UMS. The snapshots returned provide context for the agent's next step.
- **Optimization & Meta Tools (`optimization.py`, `meta.py`):**
  - `estimate_cost`, `compare_models`, `recommend_model`: Allow the AML or the LLM itself to reason about the cost and suitability of different LLMs *before* executing expensive tasks like `generate_completion` or `consolidate_memories`, enabling more efficient resource allocation.
  - `execute_optimized_workflow`: Provides a higher-level orchestration mechanism than the AML's basic loop, potentially allowing complex sub-tasks involving multiple Gateway tools (including UMS tools) to be defined and executed efficiently.
  - `get_tool_info`, `get_llm_instructions`: Allow the agent to introspect available capabilities and understand how best to use them.
- **RAG & Knowledge Base Tools (`rag.py`, `knowledge_base/`):**
  - While the UMS *is* a form of knowledge base, these tools likely implement more conventional RAG pipelines (vector store creation `create_knowledge_base`, document addition `add_documents`, context retrieval `retrieve_context`, and generation `generate_with_rag`). The EideticEngine agent could use these to interact with *external* vector stores or build specialized knowledge bases

separate from its core UMS instance, perhaps storing references or summaries within the UMS. The feedback mechanisms (`RAGFeedbackService`) offer another layer of learning distinct from UMS reflection.

- **Tournament Tools (`tournament.py`, `tournaments/`):**
  - Enable structured comparison and evolution of LLM outputs for specific tasks (code or text). The agent could initiate a tournament (`create_tournament`) to find the best way to formulate a specific `MemoryType.PROCEDURE` or refine a `MemoryType.SUMMARY`, monitor its progress (`get_tournament_status`), and store the winning result back into the UMS (`get_tournament_results -> store_memory`).

7 Evaluation & Case Studies: Demonstrating Cognitive Capabilities

We evaluated EideticEngine's architecture through detailed simulations of two distinct, complex tasks, tracing the agent's internal state and UMS interactions.

- **Case Study 1: Financial Market Analysis:** This task required the agent to:
  - **Structure:** Create and utilize separate thought chains (`tc-rates1`, `tc-equities1`) for distinct analysis streams.
  - **Plan & Depend:** Generate multi-step plans with dependencies (e.g., summarizing search results before analysis, creating chains before planning searches). `_check_prerequisites` ensured correct execution order.
  - **Remember & Retrieve:** Store key economic facts (`store_memory`, `level=semantic`), search the corpus (`fused_search`), summarize results (`summarize_text`), and retrieve internal summaries/facts for later synthesis (`get_memory_by_id`, context gathering).
  - **Link:** Explicitly link related concepts (e.g., CPI data to market summary via `create_memory_link`). Background auto-linking connected related stored facts.
  - **Reflect & Adapt:** `generate_reflection` identified a gap (political factors). The agent incorporated this feedback into its plan. `_adapt_thresholds` dynamically adjusted meta-cognitive frequency based on the rapid influx of new memories.
  - **Synthesize:** `consolidate_memories` generated a high-level insight connecting disparate stored facts.
- **Case Study 2: Creative Concept Development:** This task required the agent to:
  - **Ideate & Structure:** Brainstorm concepts (`record_thought`), select one (`record_thought(type=decision)`), store it (`store_memory`), check novelty (`external_tools:check_concept_novelty`), and create dedicated thought chains (`tc-dev1`, `tc-pilot1`) for development phases.
  - **Develop & Persist:** Create character profiles and story arcs, first as thoughts, then storing structured versions (`store_memory(type=character_profile/story_arc)`).

– **Iterate & Track:** Generate pilot script scenes iteratively, storing each (`store_memory(type=script_scene)`) and incrementally updating a draft artifact (`record_artifact`). Plan steps included dependencies ensuring scenes were generated before being added to the artifact.
– **Utilize Context:** Retrieve character profiles/arc details from UMS when generating subsequent script scenes.
– **Finalize:** Retrieve the full draft artifact (`get_artifact_by_id`), perform final formatting (simulated internal LLM step), and save the final output (`record_artifact(is_output=True)`).

Analysis: Across both studies, the EideticEngine architecture facilitated successful completion of complex, multi-phase tasks. The UMS provided the necessary persistence and structure, while the AML successfully orchestrated the LLM, managed dependencies, recovered from simulated errors (not detailed above, but handled by the error logic), and utilized meta-cognitive tools. The adaptive thresholds demonstrated self-regulation of cognitive overhead. Trace logs (provided in Supplementary Material) clearly show the evolution of the UMS state and the agent's plan over time.

8 Discussion: Implications of the EideticEngine Architecture

EideticEngine demonstrates a path towards more capable and autonomous LLM agents by integrating principles from cognitive science with robust software engineering. Key implications include:

- **Beyond Reactive Agents:** EideticEngine moves agents from simple stimulus-response loops towards goal-directed, reflective, and adaptive behavior based on persistent internal state.
- **Scalability for Complex Tasks:** Structured planning, dependency management, and modular thought chains enable tackling problems that overwhelm simpler architectures due to context limitations or lack of coherence.
- **Emergent Learning and Adaptation:** While not general learning, the combination of reflection, consolidation, memory promotion, and adaptive thresholds allows the agent to refine its knowledge base and operational strategy over time based on its experience within the UMS.
- **Introspection and Explainability:** The detailed logging in the UMS (`memory_operations`, thoughts, actions, etc.) and visualization tools provide unprecedented insight into the agent's "reasoning" process, aiding debugging and analysis.
- **Foundation for General Capabilities:** By providing robust infrastructure for memory, planning, and self-management, EideticEngine lays groundwork that future, potentially more powerful AI reasoning cores could leverage to achieve broader intelligence. The architecture itself addresses fundamental bottlenecks.

Limitations: EideticEngine still relies heavily on the quality of the core LLM's reasoning, planning, and tool-use abilities. The overhead of UMS interaction could be significant for highly real-time tasks (though `aiosqlite` and optimizations mitigate this). The heuristics for memory promotion and threshold adaptation are currently rule-based and could be further refined.

9 Conclusion: A Cognitive Leap for Agent Architectures

We introduced EideticEngine, an adaptive cognitive architecture enabling LLM agents to manage complex tasks through the tight integration of a Unified Memory System (UMS) and an Agent Master Loop (AML). By incorporating multi-level memory, structured planning with dependency checking, agent-driven meta-cognition (reflection, consolidation, promotion), and adaptive self-regulation of cognitive processes, EideticEngine demonstrates a significant advance over existing agent paradigms. Our simulations highlight its ability to support sustained, goal-directed, and introspective behavior on challenging analytical and creative tasks. EideticEngine offers a robust and extensible blueprint for the next generation of autonomous AI systems.

10 Future Work

- **Quantitative Benchmarking:** Rigorous evaluation against state-of-the-art baselines on complex, multi-step agent benchmarks.
- **Advanced Adaptation & Learning:** Exploring reinforcement learning or other ML techniques to optimize adaptive thresholds, meta-cognitive strategy selection, and procedural skill derivation.
- **Multi-Agent Systems:** Extending EideticEngine to support collaborative tasks with shared UMS spaces and coordinated planning protocols.
- **Real-Time Interaction:** Investigating architectural adaptations for tighter perception-action loops in dynamic environments.
- **Theoretical Grounding:** Further formalizing the EideticEngine loop and memory dynamics in relation to established cognitive science models and decision theory.
- **Hybrid Reasoning:** Integrating symbolic planners or knowledge graph reasoning engines that can interact with the UMS and the LLM via the AML.

Addendum

This addendum provides additional technical insights and practical implementation details that complement the main paper by focusing on aspects not previously covered in depth.

10.1 Low-Level Implementation Considerations

While the main paper details the architectural design of the UMS and AML, these additional implementation considerations are crucial for real-world deployment:

- **Transaction Management:** UMS operations use SQLite's transaction capabilities (`BEGIN`, `COMMIT`, `ROLLBACK`) with proper error handling to ensure database integrity, particularly important during concurrent background tasks like auto-linking.
- **Memory Compression Strategies:** For high-volume operations, the system implements content compression using techniques like summarization-before-storage for lengthy episodic memories, reducing storage requirements while preserving semantic value.

- **Embedding Caching:** The system maintains a local cache of recently generated embeddings to avoid redundant API calls, significantly reducing latency and costs during intensive semantic search operations.

- **Retry Logic Patterns:** The AML implements exponential backoff with jitter for tool calls, particularly external API calls, with configurable parameters (`MAX_RETRIES`, `BASE_RETRY_DELAY`, `RETRY_MULTIPLIER`) to handle transient failures gracefully.

- **Memory Garbage Collection:** Beyond TTL-based expiry, the system implements a priority-based garbage collection algorithm that considers importance, access frequency, and linking density when memory pressure exceeds configurable thresholds.

10.2 Operational Statistics and Telemetry

The EideticEngine implementation includes comprehensive telemetry options not detailed in the main paper:

- **Performance Metrics:** The system tracks granular timing statistics for critical operations (`_measure_operation_time`), including per-tool-call latency, embedding generation time, memory access patterns, and query execution time.

- **Memory Usage Patterns:** Analytics on memory utilization include distribution by type/level, average TTL before expiration or promotion, and correlation between importance scores and actual utility in subsequent operations.

- **Tool Usage Heat Maps:** The system can generate visual representations of tool usage frequency, dependencies, and success rates to identify bottlenecks or optimization opportunities.

- **Reflection Effectiveness:** The system measures the impact of reflections by tracking plan modifications, memory access pattern changes, and goal achievement rates following reflection events.

- **LLM Token Consumption:** Detailed tracking of token usage by operation type allows for cost optimization and identifies opportunities for context compression or chunking.

10.3 Micro-Level Decision Handling

The paper describes the AML's general flow, but these micro-level decision processes provide additional insight:

- **Response Parsing Strategy:** The AML uses regex-based parsing (```` re.search(r'UpdatedPlan:\TU\textbackslash{}s*\TU\textbackslash{}```json\TU\textbackslash{}s*(\TU\textbackslash{}[.+?]\TU\textbackslash{})\TU\textbackslash{}s*\TU\textbackslash{}```',response) ````) with robust fallbacks to extract structured data from potentially malformed LLM outputs.

- **Tool Selection Orchestration:** When the LLM suggests multiple potential tools, the AML applies a decision tree based on previous success rates, estimated token costs, and expected information gain to select the optimal next action.

- **Error Classification System:** The `_handle_tool_error` function categorizes errors into distinct types (dependency failure, input validation, timeout, external resource unavailable, etc.) and applies targeted recovery strategies for each.
- **Conversation Management:** For interactive applications, the AML maintains a sliding window of recent interactions with configurable compression for older history, preserving crucial decision points while managing context length.
- **LLM-Specific Optimizations:** The system includes dedicated prompting strategies and parsing logic for different LLM providers (Anthropic, OpenAI, etc.), accounting for their unique strengths and limitations.

10.4 Advanced Meta-Cognitive Mechanisms

Beyond the basic reflection and consolidation described in the main paper:

- **Self-Directed Learning:** The agent can initiate focused "study sessions" when knowledge gaps are identified, systematically exploring and encoding domain-specific information through targeted queries and structured memory storage.
- **Context Switching Costs:** The system models the cognitive cost of switching between thought chains or tasks, incorporating a "mental momentum" factor that influences when context switches are optimal versus continuing on the current thread.
- **Emotional Simulation:** For creative tasks, an experimental module can track simulated "emotional states" that influence memory retrieval salience, creative generation parameters, and reflection depth.
- **Counterfactual Exploration:** In decision-making scenarios, the agent can spawn temporary "what-if" thought chains to explore alternative approaches without committing to them, then integrate insights from these explorations into the main decision process.
- **Memory Confidence Calibration:** The system periodically tests its confidence judgments against objective criteria, adjusting confidence calculation parameters to minimize overconfidence or underconfidence biases.

10.5 Micro-Task Case Studies

These detailed micro-task examples reveal EideticEngine's operation at a granular level beyond the comprehensive simulations in the main paper:

10.5.1 Knowledge Integration Challenge

When presented with conflicting information about a technical topic, EideticEngine demonstrated sophisticated conflict resolution:

1. **Contradiction Detection:** The system identified semantic contradictions between new information and existing knowledge using bidirectional linking and relevance scoring.
2. **Authority Assessment:** Rather than relying on recency bias, the system evaluated source credibility, consistency with related knowledge, and confidence metrics.

3. **Reconciliation Strategy:** When reconciliation was possible, the system generated a conditional rule capturing the context-dependent validity of each position.

4. **Knowledge Structure Update:** The resulting memory network showed explicit "challenges" links between conflicting facts and "clarifies" links to the reconciliation rule.

10.5.2 Dynamic Planning Adaptation

Monitoring EideticEngine's handling of an unexpected mid-task constraint change revealed:

1. **Impact Analysis:** Within 2 loops, the agent identified 8 affected plan steps through dependency graph traversal, distinguishing direct impacts from ripple effects.

2. **Resource Reallocation:** The system preserved 64% of completed work by converting potentially affected outputs to intermediate assets that could be transformed to meet new constraints.

3. **Graceful Constraint Handling:** Using procedural memory access, the agent identified similar past situations and applied relevant transformation patterns without starting from scratch.

4. **Meta-Cognitive Efficiency:** The system triggered reflection at precisely the right moment (after impact assessment, before replanning) rather than at a fixed action count threshold.

10.5.3 Long-Duration Task Management

For a task spanning multiple sessions over days, EideticEngine demonstrated:

1. **Hibernate/Resume Capability:** The system saved comprehensive workflow state, including working memory focus, active thought chains, and dependency status.

2. **Re-Contextualization:** Upon resuming, the agent performed targeted hybrid search for relevant memories and reconstructed current context with 92

3. **Time-Aware Reasoning:** The system recalculated memory relevance based on elapsed time between sessions, promoting important items that might otherwise have decayed.

4. **Continuity Verification:** Before proceeding, the agent performed a lightweight "consistency check" between current and previous session state, identifying any potential context shifts.

10.6 Implementation Architecture Variants

EideticEngine's architecture allows for specialized variants not covered in the main paper:

- **Distributed UMS:** For high-throughput applications, the UMS can be horizontally scaled across multiple nodes, with memory sharding based on workflow ID or memory type, using consistent hashing for routing and a caching layer for frequent access patterns.

- **Multimodal Extension:** An extended implementation supports non-text modalities by storing media artifacts with extracted feature vectors and text annotations, enabling cross-modal linking and reasoning.

- **Resource-Constrained Variant:** For edge devices, a lightweight implementation uses SQLite's memory-only mode with selective persistence, dropping the embedding table for exact-match only retrieval, and simplified meta-cognitive cycles.

- **Multi-Agent Configuration:** A collaborative variant enables multiple agents with individual UMS instances to share selected memories via a publish-subscribe mechanism, creating a collective knowledge graph while maintaining agent-specific working memory.

- **Human-in-the-Loop Orchestration:** A specialized AML variant enables explicit human checkpoints for reviewing and modifying plans, memories, or reflections before the agent proceeds to subsequent steps.

References

[1] T. B. Brown et al. "Language models are few-shot learners". In: *Advances in Neural Information Processing Systems 33*. 2020, pp. 1877–1901. URL: https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

[2] Ashish Vaswani et al. "Attention Is All You Need". In: *Advances in Neural Information Processing Systems 30*. 2017, pp. 5998–6008. URL: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

[3] Lei Wang et al. *A Survey on Large Language Model based Autonomous Agents*. 2023. arXiv: 2308.11432 [cs.AI]. URL: https://arxiv.org/abs/2308.11432.

[4] Jason Weston, Sumit Chopra, and Antoine Bordes. *Memory Networks*. 2014. arXiv: 1410.3916 [cs.LG]. URL: https://arxiv.org/abs/1410.3916.

[5] Charles Packer et al. *MemGPT: Towards LLMs as Operating Systems*. 2023. arXiv: 2310.08560 [cs.CL]. URL: https://arxiv.org/abs/2310.08560.

[6] Toran Bruce Richards. *AutoGPT: An autonomous GPT-4 experiment*. GitHub repository. 2023. URL: https://github.com/Significant-Gravitas/Auto-GPT.

[7] Shunyu Yao et al. "ReAct: Synergizing Reasoning and Acting in Language Models". In: *International Conference on Learning Representations (ICLR 2023)*. 2023. URL: https://openreview.net/forum?id=6LNIBt1J-N.

[8] Noah Shinn, Beck Labash, and Ashwin Gopinath. *Reflexion: Language Agents with Verbal Reinforcement Learning*. 2023. arXiv: 2303.11366 [cs.CL]. URL: https://arxiv.org/abs/2303.11366.

[9] Endel Tulving. "Episodic and semantic memory". In: *Organization of memory*. Ed. by Endel Tulving and Wayne Donaldson. Academic Press, 1972, pp. 381–403.

[10] R. C. Atkinson and R. M. Shiffrin. “Human memory: A proposed system and its control processes”. In: *The psychology of learning and motivation*. Ed. by K. W. Spence and J. T. Spence. Vol. 2. Academic Press, 1968, pp. 89–195. DOI: 10.1016/S0079-7421(08)60422-3. URL: https://doi.org/10.1016/S0079-7421(08)60422-3.

[11] Yongliang Shen et al. *HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in HuggingFace*. 2023. arXiv: 2303.17580 [cs.CL]. URL: https://arxiv.org/abs/2303.17580.

[12] Patrick Lewis et al. “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks”. In: *Advances in Neural Information Processing Systems 33*. 2020, pp. 9459–9474. URL: https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945 Abstract.html.

[13] Yikang Chen et al. *Second Me: An AI-Native Memory Offload System*. 2025. arXiv: 2503.08102 [cs.AI]. URL: https://arxiv.org/abs/2503.08102.

[14] Jason Wei et al. “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models”. In: *Advances in Neural Information Processing Systems 35*. 2022, pp. 24824–24837. URL: https://proceedings.neurips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html.

[15] Shunyu Yao et al. *Tree of Thoughts: Deliberate Problem Solving with Large Language Models*. 2023. arXiv: 2305.10601 [cs.CL]. URL: https://arxiv.org/abs/2305.10601.

[16] Deepali Garg et al. *Generating Structured Plan Representation of Procedures with LLMs*. 2025. arXiv: 2504.00029 [cs.CL]. URL: https://arxiv.org/abs/2504.00029.

[17] John E. Laird, Allen Newell, and Paul S. Rosenbloom. “SOAR: An architecture for general intelligence”. In: *Artificial Intelligence* 33.1 (1987), pp. 1–64. DOI: 10.1016/0004-3702(87)90050-6. URL: https://doi.org/10.1016/0004-3702(87)90050-6.

[18] J. R. Anderson. “ACT: A simple theory of complex cognition”. In: *American Psychologist* 51.4 (1996), pp. 355–365. DOI: 10.1037/0003-066X.51.4.355. URL: https://doi.org/10.1037/0003-066X.51.4.355.

[19] J. R. Anderson and C. Lebiere. “The Newell test for a theory of cognition”. In: *Behavioral and Brain Sciences* 26.5 (2003), pp. 587–601. DOI: 10.1017/S0140525X0300013X. URL: https://doi.org/10.1017/S0140525X0300013X.

[20] Andrew M. Nuxoll and John E. Laird. “Extending Cognitive Architecture with Episodic Memory”. In: *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*. 2007, pp. 1560–1564. URL: https://www.aaai.org/Library/AAAI/2007/aaai07-253.php.

[21] Nate Derbinsky, Jiexun Li, and John E. Laird. “A Multi-Domain Evaluation of Scaling in a General Episodic Memory”. In: *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*. 2012, pp. 193–199. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/view/5000.

[22] Theodore R. Sumers et al. *Cognitive Architectures for Language Agents*. 2023. arXiv: 2309.02427 [cs.AI]. URL: https://arxiv.org/abs/2309.02427.

[23] Alexander J. Ramirez et al. *Self-adaptive agents using Large Language Models*. 2023. arXiv: 2307.06187 [cs.AI]. URL: https://arxiv.org/abs/2307.06187.

[24] A. D. Baddeley and G. J. Hitch. "Working memory". In: *The psychology of learning and motivation*. Ed. by G. H. Bower. Vol. 8. Academic Press, 1974, pp. 47–89. DOI: 10.1016/S0079-7421(08)60452-1. URL: https://doi.org/10.1016/S0079-7421(08)60452-1.

A Appendix A: Unified Memory System (UMS) Code

Listing 1: Complete Code for Unified Memory System (cognitive_and_agent_memory.py)

```python
"""Unified Agent Memory and Cognitive System.

This module provides a comprehensive memory, reasoning, and workflow tracking system
designed for LLM agents, merging sophisticated cognitive modeling with structured
process tracking.

Based on the integration plan combining 'cognitive_memory.py' and 'agent_memory.py'.

Key Features:
- Multi-level memory hierarchy (working, episodic, semantic, procedural) with rich metadata.
- Structured workflow, action, artifact, and thought chain tracking.
- Associative memory graph with automatic linking capabilities.
- Vector embeddings for semantic similarity and clustering.
- Foundational tools for recording agent activity and knowledge.
- Integrated episodic memory creation linked to actions and artifacts.
- Basic cognitive state saving (structure defined, loading/saving tools ported).
- SQLite backend using aiosqlite with performance optimizations.
"""

import asyncio
import json
import os
import time
import uuid
from collections import defaultdict
from datetime import datetime
from enum import Enum
from pathlib import Path
from typing import Any, Dict, List, Optional, Tuple

import aiosqlite
import markdown  # For HTML report generation
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity as sk_cosine_similarity

from llm_gateway.constants import Provider as LLMGatewayProvider  # To use provider constants
from llm_gateway.core.providers.base import get_provider  # For consolidation/reflection LLM calls

# Import error handling and decorators from agent_memory concepts
from llm_gateway.exceptions import ToolError, ToolInputError
from llm_gateway.services.vector.embeddings import get_embedding_service
from llm_gateway.tools.base import with_error_handling, with_tool_metrics
from llm_gateway.utils import get_logger

logger = get_logger("llm_gateway.tools.unified_memory")

# ======================================================
# Configuration Settings
# ======================================================

DEFAULT_DB_PATH = os.environ.get("AGENT_MEMORY_DB_PATH", "unified_agent_memory.db")
MAX_TEXT_LENGTH = 64000  # Maximum length for text fields (from agent_memory)
CONNECTION_TIMEOUT = 10.0  # seconds (from cognitive_memory)
ISOLATION_LEVEL = None  # autocommit mode (from cognitive_memory)

# Memory management parameters (from cognitive_memory)
MAX_WORKING_MEMORY_SIZE = int(os.environ.get("MAX_WORKING_MEMORY_SIZE", "20"))
DEFAULT_TTL = {
    "working": 60 * 30,       # 30 minutes
    "episodic": 60 * 60 * 24 * 7, # 7 days (Increased default)
    "semantic": 60 * 60 * 24 * 30, # 30 days
    "procedural": 60 * 60 * 24 * 90 # 90 days
}
MEMORY_DECAY_RATE = float(os.environ.get("MEMORY_DECAY_RATE", "0.01"))  # Per hour
IMPORTANCE_BOOST_FACTOR = float(os.environ.get("IMPORTANCE_BOOST_FACTOR", "1.5"))

# Embedding model configuration (from cognitive_memory)
DEFAULT_EMBEDDING_MODEL = "text-embedding-3-small"
EMBEDDING_DIMENSION = 384  # For the default model
SIMILARITY_THRESHOLD = 0.75

# SQLite optimization pragmas (from cognitive_memory)
```

```python
SQLITE_PRAGMAS = [
    "PRAGMA journal_mode=WAL",
    "PRAGMA synchronous=NORMAL",
    "PRAGMA foreign_keys=ON",
    "PRAGMA temp_store=MEMORY",
    "PRAGMA cache_size=-32000",
    "PRAGMA mmap_size=2147483647",
    "PRAGMA busy_timeout=30000"
]

# ======================================================
# Enums (Combined & Standardized)
# ======================================================

# --- Workflow & Action Status ---
class WorkflowStatus(str, Enum):
    ACTIVE = "active"
    PAUSED = "paused"
    COMPLETED = "completed"
    FAILED = "failed"
    ABANDONED = "abandoned"

class ActionStatus(str, Enum):
    PLANNED = "planned"
    IN_PROGRESS = "in_progress"
    COMPLETED = "completed"
    FAILED = "failed"
    SKIPPED = "skipped"

# --- Content Types ---
class ActionType(str, Enum):
    TOOL_USE = "tool_use"
    REASONING = "reasoning"
    PLANNING = "planning"
    RESEARCH = "research"
    ANALYSIS = "analysis"
    DECISION = "decision"
    OBSERVATION = "observation"
    REFLECTION = "reflection"
    SUMMARY = "summary"
    CONSOLIDATION = "consolidation"
    MEMORY_OPERATION = "memory_operation"

class ArtifactType(str, Enum):
    FILE = "file"
    TEXT = "text"
    IMAGE = "image"
    TABLE = "table"
    CHART = "chart"
    CODE = "code"
    DATA = "data"
    JSON = "json"
    URL = "url"

class ThoughtType(str, Enum):
    GOAL = "goal"
    QUESTION = "question"
    HYPOTHESIS = "hypothesis"
    INFERENCE = "inference"
    EVIDENCE = "evidence"
    CONSTRAINT = "constraint"
    PLAN = "plan"
    DECISION = "decision"
    REFLECTION = "reflection"
    CRITIQUE = "critique"
    SUMMARY = "summary"

# --- Memory System Types ---
class MemoryLevel(str, Enum):
    WORKING = "working"
    EPISODIC = "episodic"
    SEMANTIC = "semantic"
    PROCEDURAL = "procedural"

class MemoryType(str, Enum):
    """Content type classifications for memories. Combines concepts."""
    OBSERVATION = "observation"    # Raw data or sensory input (like text)
    ACTION_LOG = "action_log"      # Record of an agent action
```

```
    TOOL_OUTPUT = "tool_output"    # Result from a tool
    ARTIFACT_CREATION = "artifact_creation" # Record of artifact generation
    REASONING_STEP = "reasoning_step" # Corresponds to a thought
    FACT = "fact"                  # Verifiable piece of information
    INSIGHT = "insight"            # Derived understanding or pattern
    PLAN = "plan"                  # Future intention or strategy
    QUESTION = "question"          # Posed question or uncertainty
    SUMMARY = "summary"            # Condensed information
    REFLECTION = "reflection"      # Meta-cognitive analysis (distinct from thought type)
    SKILL = "skill"                # Learned capability (like procedural)
    PROCEDURE = "procedure"        # Step-by-step method
    PATTERN = "pattern"            # Recognized recurring structure
    CODE = "code"                  # Code snippet
    JSON = "json"                  # Structured JSON data
    URL = "url"                    # A web URL
    TEXT = "text"                  # Generic text block (fallback)
    # Retain IMAGE? Needs blob storage/linking capability. Deferred.

class LinkType(str, Enum):
    """Types of associations between memories (from cognitive_memory)."""
    RELATED = "related"
    CAUSAL = "causal"
    SEQUENTIAL = "sequential"
    HIERARCHICAL = "hierarchical"
    CONTRADICTS = "contradicts"
    SUPPORTS = "supports"
    GENERALIZES = "generalizes"
    SPECIALIZES = "specializes"
    FOLLOWS = "follows"
    PRECEDES = "precedes"
    TASK = "task"
    REFERENCES = "references" # Added for linking thoughts/actions to memories


# ======================================================
# Database Schema (Merged & Refined)
# ======================================================

# Note: Using TEXT for IDs (UUIDs) and TIMESTAMP for datetimes (ISO format strings)
# Note: Added comments explaining origins and modifications.

SCHEMA_SQL = """
-- Base Pragmas (Combined)
PRAGMA foreign_keys = ON;
PRAGMA journal_mode=WAL;
PRAGMA synchronous=NORMAL;
PRAGMA temp_store=MEMORY;
PRAGMA cache_size=-32000;
PRAGMA mmap_size=2147483647;
PRAGMA busy_timeout=30000;

-- Workflows table (Based on agent_memory, kept fields from cognitive_memory if overlapping)
CREATE TABLE IF NOT EXISTS workflows (
    workflow_id TEXT PRIMARY KEY,
    title TEXT NOT NULL,              -- From agent_memory
    description TEXT,                 -- From agent_memory & cognitive_memory
    goal TEXT,                        -- From agent_memory
    status TEXT NOT NULL,             -- From agent_memory (uses WorkflowStatus enum)
    created_at TIMESTAMP NOT NULL,
    updated_at TIMESTAMP NOT NULL,
    completed_at TIMESTAMP,           -- From agent_memory
    parent_workflow_id TEXT,          -- From agent_memory
    metadata TEXT,                    -- JSON serialized, combined concept
    last_active INTEGER               -- From cognitive_memory (Unix timestamp for potential sorting)
);

-- Actions table (From agent_memory)
CREATE TABLE IF NOT EXISTS actions (
    action_id TEXT PRIMARY KEY,
    workflow_id TEXT NOT NULL,
    parent_action_id TEXT,
    action_type TEXT NOT NULL,        -- Uses ActionType enum
    title TEXT,
    reasoning TEXT,
    tool_name TEXT,
    tool_args TEXT,                   -- JSON serialized
    tool_result TEXT,                 -- JSON serialized
    status TEXT NOT NULL,             -- Uses ActionStatus enum
```

```
    started_at TIMESTAMP NOT NULL,
    completed_at TIMESTAMP,
    sequence_number INTEGER,
    FOREIGN KEY (workflow_id) REFERENCES workflows(workflow_id) ON DELETE CASCADE,
    FOREIGN KEY (parent_action_id) REFERENCES actions(action_id) ON DELETE SET NULL
);

-- Artifacts table (From agent_memory)
CREATE TABLE IF NOT EXISTS artifacts (
    artifact_id TEXT PRIMARY KEY,
    workflow_id TEXT NOT NULL,
    action_id TEXT,                   -- Action that created this
    artifact_type TEXT NOT NULL,      -- Uses ArtifactType enum
    name TEXT NOT NULL,
    description TEXT,
    path TEXT,                        -- Filesystem path
    content TEXT,                     -- For text-based artifacts
    metadata TEXT,                    -- JSON serialized
    created_at TIMESTAMP NOT NULL,
    is_output BOOLEAN DEFAULT FALSE,
    FOREIGN KEY (workflow_id) REFERENCES workflows(workflow_id) ON DELETE CASCADE,
    FOREIGN KEY (action_id) REFERENCES actions(action_id) ON DELETE SET NULL
);

-- Thought chains table (From agent_memory)
CREATE TABLE IF NOT EXISTS thought_chains (
    thought_chain_id TEXT PRIMARY KEY,
    workflow_id TEXT NOT NULL,
    action_id TEXT,                   -- Optional action context
    title TEXT NOT NULL,
    created_at TIMESTAMP NOT NULL,
    FOREIGN KEY (workflow_id) REFERENCES workflows(workflow_id) ON DELETE CASCADE,
    FOREIGN KEY (action_id) REFERENCES actions(action_id) ON DELETE SET NULL
);

-- Thoughts table (From agent_memory, modified)
CREATE TABLE IF NOT EXISTS thoughts (
    thought_id TEXT PRIMARY KEY,
    thought_chain_id TEXT NOT NULL,
    parent_thought_id TEXT,
    thought_type TEXT NOT NULL,        -- Uses ThoughtType enum
    content TEXT NOT NULL,
    sequence_number INTEGER NOT NULL,
    created_at TIMESTAMP NOT NULL,
    relevant_action_id TEXT,           -- Action this thought relates to/caused
    relevant_artifact_id TEXT,         -- Artifact this thought relates to
    relevant_memory_id TEXT,           -- *** NEW FK: Memory entry this thought relates to ***
    FOREIGN KEY (thought_chain_id) REFERENCES thought_chains(thought_chain_id) ON DELETE CASCADE,
    FOREIGN KEY (parent_thought_id) REFERENCES thoughts(thought_id) ON DELETE SET NULL,
    FOREIGN KEY (relevant_action_id) REFERENCES actions(action_id) ON DELETE SET NULL,
    FOREIGN KEY (relevant_artifact_id) REFERENCES artifacts(artifact_id) ON DELETE SET NULL,
    FOREIGN KEY (relevant_memory_id) REFERENCES memories(memory_id) ON DELETE SET NULL -- *** NEW FK ***
);

-- Memories table (Based on cognitive_memory, modified)
CREATE TABLE IF NOT EXISTS memories (
    memory_id TEXT PRIMARY KEY,        -- Renamed from 'id' for clarity
    workflow_id TEXT NOT NULL,
    content TEXT NOT NULL,             -- The core memory content
    memory_level TEXT NOT NULL,        -- Uses MemoryLevel enum
    memory_type TEXT NOT NULL,         -- Uses MemoryType enum
    importance REAL DEFAULT 5.0,       -- Relevance score (1.0-10.0)
    confidence REAL DEFAULT 1.0,       -- Confidence score (0.0-1.0)
    description TEXT,                  -- Optional short description
    reasoning TEXT,                    -- Optional reasoning for the memory
    source TEXT,                       -- Origin (tool name, file, user, etc.)
    context TEXT,                      -- JSON context of memory creation
    tags TEXT,                         -- JSON array of tags
    created_at INTEGER NOT NULL,       -- Unix timestamp (kept from cognitive_memory for relevance calcs)
    updated_at INTEGER NOT NULL,       -- Unix timestamp
    last_accessed INTEGER,             -- Unix timestamp
    access_count INTEGER DEFAULT 0,
    ttl INTEGER DEFAULT 0,             -- TTL in seconds (0 = permanent)
    embedding_id TEXT,                 -- FK to embeddings table
    action_id TEXT,                    -- *** NEW FK: Action associated with this memory ***
    thought_id TEXT,                   -- *** NEW FK: Thought associated with this memory ***
    artifact_id TEXT,                  -- *** NEW FK: Artifact associated with this memory ***
    FOREIGN KEY (workflow_id) REFERENCES workflows(workflow_id) ON DELETE CASCADE,
```

```
    FOREIGN KEY (embedding_id) REFERENCES embeddings(id) ON DELETE SET NULL,
    FOREIGN KEY (action_id) REFERENCES actions(action_id) ON DELETE SET NULL, -- *** NEW FK ***
    FOREIGN KEY (thought_id) REFERENCES thoughts(thought_id) ON DELETE SET NULL, -- *** NEW FK ***
    FOREIGN KEY (artifact_id) REFERENCES artifacts(artifact_id) ON DELETE SET NULL -- *** NEW FK ***
);

-- Memory links table (From cognitive_memory)
CREATE TABLE IF NOT EXISTS memory_links (
    link_id TEXT PRIMARY KEY,         -- Renamed from 'id'
    source_memory_id TEXT NOT NULL,   -- Renamed from 'source_id'
    target_memory_id TEXT NOT NULL,   -- Renamed from 'target_id'
    link_type TEXT NOT NULL,          -- Uses LinkType enum
    strength REAL DEFAULT 1.0,
    description TEXT,
    created_at INTEGER NOT NULL,       -- Unix timestamp
    FOREIGN KEY (source_memory_id) REFERENCES memories(memory_id) ON DELETE CASCADE,
    FOREIGN KEY (target_memory_id) REFERENCES memories(memory_id) ON DELETE CASCADE,
    UNIQUE(source_memory_id, target_memory_id, link_type)
);

-- Embeddings table (From cognitive_memory)
CREATE TABLE IF NOT EXISTS embeddings (
    id TEXT PRIMARY KEY,               -- Embedding hash ID
    memory_id TEXT UNIQUE,             -- Link back to the memory
    model TEXT NOT NULL,               -- Embedding model used
    embedding BLOB NOT NULL,           -- Serialized vector
    created_at INTEGER NOT NULL,       -- Unix timestamp
    FOREIGN KEY (memory_id) REFERENCES memories(memory_id) ON DELETE CASCADE
);

-- Cognitive states table (From agent_memory, will store memory_ids)
CREATE TABLE IF NOT EXISTS cognitive_states (
    state_id TEXT PRIMARY KEY,
    workflow_id TEXT NOT NULL,
    title TEXT NOT NULL,
    working_memory TEXT,               -- JSON array of memory_ids in active working memory
    focus_areas TEXT,                  -- JSON array of memory_ids or descriptive strings
    context_actions TEXT,              -- JSON array of relevant action_ids
    current_goals TEXT,                -- JSON array of goal descriptions or thought_ids
    created_at TIMESTAMP NOT NULL,
    is_latest BOOLEAN NOT NULL,
    FOREIGN KEY (workflow_id) REFERENCES workflows(workflow_id) ON DELETE CASCADE
);

-- Reflections table (From cognitive_memory - for meta-cognitive analysis)
CREATE TABLE IF NOT EXISTS reflections (
    reflection_id TEXT PRIMARY KEY,    -- Renamed from 'id'
    workflow_id TEXT NOT NULL,
    title TEXT NOT NULL,
    content TEXT NOT NULL,
    reflection_type TEXT NOT NULL,     -- summary, insight, planning, etc.
    created_at INTEGER NOT NULL,       -- Unix timestamp
    referenced_memories TEXT,          -- JSON array of memory_ids
    FOREIGN KEY (workflow_id) REFERENCES workflows(workflow_id) ON DELETE CASCADE
);

-- Memory operations log (From cognitive_memory - for auditing/debugging)
CREATE TABLE IF NOT EXISTS memory_operations (
    operation_log_id TEXT PRIMARY KEY, -- Renamed from 'id'
    workflow_id TEXT NOT NULL,
    memory_id TEXT,                    -- Related memory, if applicable
    action_id TEXT,                    -- Related action, if applicable
    operation TEXT NOT NULL,           -- create, update, access, link, consolidate, expire, reflect, etc.
    operation_data TEXT,               -- JSON of operation details
    timestamp INTEGER NOT NULL,        -- Unix timestamp
    FOREIGN KEY (workflow_id) REFERENCES workflows(workflow_id) ON DELETE CASCADE,
    FOREIGN KEY (memory_id) REFERENCES memories(memory_id) ON DELETE SET NULL,
    FOREIGN KEY (action_id) REFERENCES actions(action_id) ON DELETE SET NULL
);

-- Tags table (From agent_memory)
CREATE TABLE IF NOT EXISTS tags (
    tag_id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL UNIQUE,
    description TEXT,
    category TEXT,
    created_at TIMESTAMP NOT NULL
);
```

```sql
-- Junction Tables for Tags (From agent_memory)
CREATE TABLE IF NOT EXISTS workflow_tags (
    workflow_id TEXT NOT NULL,
    tag_id INTEGER NOT NULL,
    PRIMARY KEY (workflow_id, tag_id),
    FOREIGN KEY (workflow_id) REFERENCES workflows(workflow_id) ON DELETE CASCADE,
    FOREIGN KEY (tag_id) REFERENCES tags(tag_id) ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS action_tags (
    action_id TEXT NOT NULL,
    tag_id INTEGER NOT NULL,
    PRIMARY KEY (action_id, tag_id),
    FOREIGN KEY (action_id) REFERENCES actions(action_id) ON DELETE CASCADE,
    FOREIGN KEY (tag_id) REFERENCES tags(tag_id) ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS artifact_tags (
    artifact_id TEXT NOT NULL,
    tag_id INTEGER NOT NULL,
    PRIMARY KEY (artifact_id, tag_id),
    FOREIGN KEY (artifact_id) REFERENCES artifacts(artifact_id) ON DELETE CASCADE,
    FOREIGN KEY (tag_id) REFERENCES tags(tag_id) ON DELETE CASCADE
);

-- Dependencies table (From agent_memory)
CREATE TABLE IF NOT EXISTS dependencies (
    dependency_id INTEGER PRIMARY KEY AUTOINCREMENT,
    source_action_id TEXT NOT NULL,    -- The action that depends on the target
    target_action_id TEXT NOT NULL,    -- The action that is depended upon
    dependency_type TEXT NOT NULL,     -- Type of dependency (e.g., 'requires', 'informs')
    created_at TIMESTAMP NOT NULL,     -- When the dependency was created
    FOREIGN KEY (source_action_id) REFERENCES actions (action_id) ON DELETE CASCADE,
    FOREIGN KEY (target_action_id) REFERENCES actions (action_id) ON DELETE CASCADE,
    UNIQUE(source_action_id, target_action_id, dependency_type)
);
-- Create Indices (Combined & Updated)
-- Workflow indices
CREATE INDEX IF NOT EXISTS idx_workflows_status ON workflows(status);
CREATE INDEX IF NOT EXISTS idx_workflows_parent ON workflows(parent_workflow_id);
CREATE INDEX IF NOT EXISTS idx_workflows_last_active ON workflows(last_active DESC); -- For cognitive_memory feature
-- Action indices
CREATE INDEX IF NOT EXISTS idx_actions_workflow_id ON actions(workflow_id);
CREATE INDEX IF NOT EXISTS idx_actions_parent ON actions(parent_action_id);
CREATE INDEX IF NOT EXISTS idx_actions_sequence ON actions(workflow_id, sequence_number);
CREATE INDEX IF NOT EXISTS idx_actions_type ON actions(action_type);
-- Artifact indices
CREATE INDEX IF NOT EXISTS idx_artifacts_workflow_id ON artifacts(workflow_id);
CREATE INDEX IF NOT EXISTS idx_artifacts_action_id ON artifacts(action_id);
CREATE INDEX IF NOT EXISTS idx_artifacts_type ON artifacts(artifact_type);
-- Thought indices
CREATE INDEX IF NOT EXISTS idx_thought_chains_workflow ON thought_chains(workflow_id);
CREATE INDEX IF NOT EXISTS idx_thoughts_chain ON thoughts(thought_chain_id);
CREATE INDEX IF NOT EXISTS idx_thoughts_sequence ON thoughts(thought_chain_id, sequence_number);
CREATE INDEX IF NOT EXISTS idx_thoughts_type ON thoughts(thought_type);
-- Memory indices (Updated for new table and FKs)
CREATE INDEX IF NOT EXISTS idx_memories_workflow ON memories(workflow_id);
CREATE INDEX IF NOT EXISTS idx_memories_level ON memories(memory_level);
CREATE INDEX IF NOT EXISTS idx_memories_type ON memories(memory_type);
CREATE INDEX IF NOT EXISTS idx_memories_importance ON memories(importance DESC);
CREATE INDEX IF NOT EXISTS idx_memories_confidence ON memories(confidence DESC);
CREATE INDEX IF NOT EXISTS idx_memories_created ON memories(created_at DESC);
CREATE INDEX IF NOT EXISTS idx_memories_accessed ON memories(last_accessed DESC);
CREATE INDEX IF NOT EXISTS idx_memories_embedding ON memories(embedding_id);
CREATE INDEX IF NOT EXISTS idx_memories_action_id ON memories(action_id); -- New Index
CREATE INDEX IF NOT EXISTS idx_memories_thought_id ON memories(thought_id); -- New Index
CREATE INDEX IF NOT EXISTS idx_memories_artifact_id ON memories(artifact_id); -- New Index
-- Link indices
CREATE INDEX IF NOT EXISTS idx_memory_links_source ON memory_links(source_memory_id);
CREATE INDEX IF NOT EXISTS idx_memory_links_target ON memory_links(target_memory_id);
CREATE INDEX IF NOT EXISTS idx_memory_links_type ON memory_links(link_type);
-- Cognitive State indices
CREATE INDEX IF NOT EXISTS idx_cognitive_states_workflow ON cognitive_states(workflow_id);
CREATE INDEX IF NOT EXISTS idx_cognitive_states_latest ON cognitive_states(workflow_id, is_latest);
-- Reflection indices
CREATE INDEX IF NOT EXISTS idx_reflections_workflow ON reflections(workflow_id);
-- Operation Log indices
```

```
CREATE INDEX IF NOT EXISTS idx_operations_workflow ON memory_operations(workflow_id);
CREATE INDEX IF NOT EXISTS idx_operations_memory ON memory_operations(memory_id);
CREATE INDEX IF NOT EXISTS idx_operations_timestamp ON memory_operations(timestamp DESC);
-- Tag indices
CREATE INDEX IF NOT EXISTS idx_tags_name ON tags(name);
CREATE INDEX IF NOT EXISTS idx_workflow_tags ON workflow_tags(tag_id); -- Index tag_id for lookups
CREATE INDEX IF NOT EXISTS idx_action_tags ON action_tags(tag_id);
CREATE INDEX IF NOT EXISTS idx_artifact_tags ON artifact_tags(tag_id);
-- Dependency indices
CREATE INDEX IF NOT EXISTS idx_dependencies_source ON dependencies(source_action_id);
CREATE INDEX IF NOT EXISTS idx_dependencies_target ON dependencies(target_action_id);

-- FTS5 virtual table for memories (Updated from cognitive_memory)
CREATE VIRTUAL TABLE IF NOT EXISTS memory_fts USING fts5(
    content, description, reasoning, tags, -- Index more fields
    workflow_id UNINDEXED,
    memory_id UNINDEXED,
    content='memories',
    content_rowid='rowid',
    tokenize='porter unicode61' -- Or consider 'trigram' for substring search
);

-- Triggers to keep FTS5 table in sync (Updated for new table/columns)
CREATE TRIGGER IF NOT EXISTS memories_after_insert AFTER INSERT ON memories BEGIN
    INSERT INTO memory_fts(rowid, content, description, reasoning, tags, workflow_id, memory_id)
    VALUES (new.rowid, new.content, new.description, new.reasoning, new.tags, new.workflow_id, new.memory_id);
END;
CREATE TRIGGER IF NOT EXISTS memories_after_delete AFTER DELETE ON memories BEGIN
    INSERT INTO memory_fts(memory_fts, rowid, content, description, reasoning, tags, workflow_id, memory_id)
    VALUES ('delete', old.rowid, old.content, old.description, old.reasoning, old.tags, old.workflow_id,
    old.memory_id);
END;
CREATE TRIGGER IF NOT EXISTS memories_after_update AFTER UPDATE ON memories BEGIN
    INSERT INTO memory_fts(memory_fts, rowid, content, description, reasoning, tags, workflow_id, memory_id)
    VALUES ('delete', old.rowid, old.content, old.description, old.reasoning, old.tags, old.workflow_id,
    old.memory_id);
    INSERT INTO memory_fts(rowid, content, description, reasoning, tags, workflow_id, memory_id)
    VALUES (new.rowid, new.content, new.description, new.reasoning, new.tags, new.workflow_id, new.memory_id);
END;
"""

# ======================================================
# Database Connection Management (Adapted from agent_memory)
# ======================================================

class DBConnection:
    """Context manager for database connections using aiosqlite."""

    _instance = None
    _lock = asyncio.Lock()

    def __init__(self, db_path: str = DEFAULT_DB_PATH):
        self.db_path = db_path
        self.conn: Optional[aiosqlite.Connection] = None
        # Ensure directory exists synchronously during init
        Path(self.db_path).parent.mkdir(parents=True, exist_ok=True)

    async def __aenter__(self) -> aiosqlite.Connection:
        async with DBConnection._lock:
            if DBConnection._instance is None:
                logger.info(f"Connecting to database: {self.db_path}", emoji_key="database")
                self.conn = await aiosqlite.connect(
                    self.db_path,
                    timeout=CONNECTION_TIMEOUT # Use timeout from cognitive_memory
                    # isolation_level=ISOLATION_LEVEL # aiosqlite handles transactions differently
                )
                self.conn.row_factory = aiosqlite.Row

                # Apply optimizations
                for pragma in SQLITE_PRAGMAS:
                    await self.conn.execute(pragma)

                # Enable custom functions needed by cognitive_memory parts
                await self.conn.create_function("json_contains", 2, _json_contains, deterministic=True)
                await self.conn.create_function("json_contains_any", 2, _json_contains_any, deterministic=True)
                await self.conn.create_function("json_contains_all", 2, _json_contains_all, deterministic=True)
                await self.conn.create_function("compute_memory_relevance", 5, _compute_memory_relevance,
                deterministic=True)
```

```python
                # Initialize schema if needed
                # Check if tables exist before running the full script
                cursor = await self.conn.execute("SELECT name FROM sqlite_master WHERE type='table' AND
                name='workflows'")
                table_exists = await cursor.fetchone()
                if not table_exists:
                    logger.info("Database schema not found. Initializing...", emoji_key="gear")
                    await self.conn.executescript(SCHEMA_SQL)
                    await self.conn.commit()
                    logger.success("Database schema initialized successfully.", emoji_key="white_check_mark")
                else:
                    # Optionally, add schema migration logic here in the future
                    logger.info("Database schema already exists.", emoji_key="database")
                    # Ensure foreign keys are on for existing connections
                    await self.conn.execute("PRAGMA foreign_keys = ON")

                DBConnection._instance = self.conn
            else:
                # Reuse existing connection instance
                self.conn = DBConnection._instance

            # Ensure foreign keys are enabled for this transaction/cursor
            await self.conn.execute("PRAGMA foreign_keys = ON;")
            return self.conn

    async def __aexit__(self, exc_type, exc_val, exc_tb):
        # We don't close the connection here anymore if using singleton pattern
        # The connection should be managed globally or closed on application shutdown
        if exc_type is not None:
             logger.error(f"Database error occurred: {exc_val}", exc_info=(exc_type, exc_val, exc_tb))
        # If not using singleton, uncomment the close below
        # if self.conn:
        #     await self.conn.close()
        #     DBConnection._instance = None
        pass

# Custom SQLite helper functions (from cognitive_memory) - Keep these
def _json_contains(json_text, search_value):
    if not json_text:
        return False
    try:
        return search_value in json.loads(json_text) if isinstance(json.loads(json_text), list) else False
    except Exception:
        return False

def _json_contains_any(json_text, search_values_json):
    if not json_text or not search_values_json:
        return False
    try:
        data = json.loads(json_text)
        search_values = json.loads(search_values_json)
        if not isinstance(data, list) or not isinstance(search_values, list):
            return False
        return any(value in data for value in search_values)
    except Exception:
        return False

def _json_contains_all(json_text, search_values_json):
    if not json_text or not search_values_json:
        return False
    try:
        data = json.loads(json_text)
        search_values = json.loads(search_values_json)
        if not isinstance(data, list) or not isinstance(search_values, list):
            return False
        return all(value in data for value in search_values)
    except Exception:
        return False

def _compute_memory_relevance(importance, confidence, created_at, access_count, last_accessed):
    """Computes a relevance score based on multiple factors. Uses Unix Timestamps."""
    now = time.time()
    age_hours = (now - created_at) / 3600 if created_at else 0
    recency_factor = 1.0 / (1.0 + (now - (last_accessed or created_at)) / 86400) # Use created_at if never accessed

    decayed_importance = max(0, importance * (1.0 - MEMORY_DECAY_RATE * age_hours))
    usage_boost = min(1.0 + (access_count / 10.0), 2.0) if access_count else 1.0
```

```python
    relevance = (decayed_importance * usage_boost * confidence * recency_factor)
    return min(max(relevance, 0.0), 10.0)


# ======================================================
# Utilities (Combined & Refined)
# ======================================================


class MemoryUtils:
    """Utility methods for memory operations."""

    @staticmethod
    def generate_id() -> str:
        """Generate a unique UUID V4 string for database records."""
        return str(uuid.uuid4())

    @staticmethod
    async def serialize(obj: Any) -> Optional[str]:
        """Safely serialize an arbitrary Python object to a JSON string.

        Handles potential serialization errors and very large objects.
        Attempts to represent complex objects that fail direct serialization.
        If the final JSON string exceeds MAX_TEXT_LENGTH, it returns a
        JSON object indicating truncation.
        """
        if obj is None:
            return None

        try:
            # Attempt direct JSON serialization with reasonable defaults
            json_str = json.dumps(obj, ensure_ascii=False, default=str)
        except TypeError as e:
            # Handle objects that are not directly serializable (like sets, custom classes)
            logger.debug(f"Direct serialization failed for type {type(obj)}: {e}. Trying fallback.")
            try:
                # Attempt a fallback using string representation within a structured error
                fallback_repr = str(obj)
                # Ensure fallback doesn't exceed limits either
                if len(fallback_repr.encode('utf-8')) > MAX_TEXT_LENGTH:
                    byte_limit = MAX_TEXT_LENGTH
                    # Adjust byte_limit to avoid splitting multi-byte characters
                    while True:
                        try:
                            truncated_repr = fallback_repr[:byte_limit].encode('utf-8').decode('utf-8')
                            break
                        except UnicodeDecodeError:
                            byte_limit -= 1
                            if byte_limit <= 0:
                                truncated_repr = ""
                                break
                    fallback_repr = truncated_repr + "..."
                    logger.warning(f"Fallback string representation also exceeded max length for type {type(obj)}.")

                json_str = json.dumps({
                    "error": f"Serialization failed for type {type(obj)}.",
                    "fallback_repr": fallback_repr
                })
            except Exception as fallback_e:
                # Final fallback if even string conversion fails
                logger.error(f"Could not serialize object of type {type(obj)} even with fallback: {fallback_e}",
                exc_info=True)
                json_str = json.dumps({
                    "error": f"Unserializable object type {type(obj)}. Fallback failed.",
                    "critical_error": str(fallback_e)
                })

        # Check final length against MAX_TEXT_LENGTH (bytes)
        if len(json_str.encode('utf-8')) > MAX_TEXT_LENGTH:
            logger.warning(f"Serialized JSON string exceeds max length ({MAX_TEXT_LENGTH} bytes). Returning
            truncated indicator.")
            # Create a specific JSON structure indicating truncation
            # We don't truncate the actual JSON string here, as partial JSON is often useless.
            # Instead, we return a specific error structure.
            # Alternatively, depending on needs, one could try to truncate *parts* of the data,
            # e.g., elements in a list or values in a dict, but that's complex and lossy.
            # This approach clearly signals that the full data wasn't stored.
```

```python
            return json.dumps({
                "error": "Serialized content exceeded maximum length.",
                "original_type": str(type(obj)),
                "preview": json_str[:200] + "..." # Provide a small preview
            })
        else:
            # Return the valid JSON string if within limits
            return json_str


    @staticmethod
    async def deserialize(json_str: Optional[str]) -> Any:
        """Safely deserialize a JSON string back into a Python object.

        Handles None input and potential JSON decoding errors. If decoding fails,
        it returns the original string, assuming it might not have been JSON
        in the first place (e.g., a truncated representation).
        """
        if json_str is None:
            return None
        if not isinstance(json_str, str):
             # If it's not a string, it's probably already deserialized or invalid input
             logger.warning(f"Attempted to deserialize non-string input: {type(json_str)}. Returning as is.")
             return json_str
        if not json_str.strip(): # Handle empty strings
             return None

        try:
            # Attempt to load the JSON string
            return json.loads(json_str)
        except json.JSONDecodeError as e:
            # If it fails, log the issue and return the original string
            # This might happen if the string stored was an error message or truncated data
            logger.debug(f"Failed to deserialize JSON: {e}. Content was: '{json_str[:100]}...'. Returning raw
            string.")
            return json_str
        except Exception as e:
             # Catch other potential errors during deserialization
             logger.error(f"Unexpected error deserializing JSON: {e}. Content: '{json_str[:100]}...'",
             exc_info=True)
             return json_str # Return original string as fallback

    @staticmethod
    async def get_next_sequence_number(conn: aiosqlite.Connection, parent_id: str, table: str, parent_col: str) ->
    int:
        """Get the next sequence number for ordering items within a parent scope.

        Args:
            conn: The database connection.
            parent_id: The ID of the parent entity (e.g., workflow_id, thought_chain_id).
            table: The name of the table containing the sequence number (e.g., 'actions', 'thoughts').
            parent_col: The name of the column linking to the parent entity.

        Returns:
            The next available integer sequence number (starting from 1).
        """
        sql = f"SELECT MAX(sequence_number) FROM {table} WHERE {parent_col} = ?"
        async with conn.execute(sql, (parent_id,)) as cursor:
            row = await cursor.fetchone()
            # If no rows exist (row is None) or MAX is NULL, start at 1. Otherwise, increment max.
            max_sequence = row[0] if row and row[0] is not None else 0
            return max_sequence + 1

    @staticmethod
    async def process_tags(conn: aiosqlite.Connection, entity_id: str, tags: List[str],
                           entity_type: str) -> None:
        """Ensures tags exist in the 'tags' table and associates them with a given entity
           in the appropriate junction table (e.g., 'workflow_tags').

        Args:
            conn: The database connection.
            entity_id: The ID of the entity (workflow, action, artifact).
            tags: A list of tag names (strings) to associate. Duplicates are handled.
            entity_type: The type of the entity ('workflow', 'action', 'artifact').
        """
        if not tags:
            return # Nothing to do if no tags are provided
```

```python
        junction_table = f"{entity_type}_tags"
        id_column = f"{entity_type}_id"
        tag_ids_to_link = []
        unique_tags = list(set(str(tag).strip().lower() for tag in tags if str(tag).strip())) # Clean, lowercase,
        unique tags
        now_ts = datetime.utcnow().isoformat()

        if not unique_tags:
            return # Nothing to do if tags are empty after cleaning

        # Ensure all unique tags exist in the 'tags' table and get their IDs
        for tag_name in unique_tags:
            # Attempt to insert the tag, ignoring if it already exists
            await conn.execute(
                """
                INSERT INTO tags (name, created_at) VALUES (?, ?)
                ON CONFLICT(name) DO NOTHING;
                """,
                (tag_name, now_ts)
            )
            # Retrieve the tag_id (whether newly inserted or existing)
            async with conn.execute("SELECT tag_id FROM tags WHERE name = ?", (tag_name,)) as cursor:
                row = await cursor.fetchone()
                if row:
                    tag_ids_to_link.append(row["tag_id"])
                else:
                    # This should ideally not happen due to the upsert logic, but log if it does
                    logger.warning(f"Could not find or create tag_id for tag: {tag_name}")

        # Link the retrieved tag IDs to the entity in the junction table
        if tag_ids_to_link:
            link_values = [(entity_id, tag_id) for tag_id in tag_ids_to_link]
            # Use INSERT OR IGNORE to handle potential race conditions or duplicate calls gracefully
            await conn.executemany(
                f"INSERT OR IGNORE INTO {junction_table} ({id_column}, tag_id) VALUES (?, ?)",
                link_values
            )
            logger.debug(f"Associated {len(link_values)} tags with {entity_type} {entity_id}")


    @staticmethod
    async def _log_memory_operation(conn: aiosqlite.Connection, workflow_id: str, operation: str,
                                    memory_id: Optional[str] = None, action_id: Optional[str] = None,
                                    operation_data: Optional[Dict] = None):
        """Logs an operation related to memory management or agent activity. Internal helper."""
        try:
            op_id = MemoryUtils.generate_id()
            timestamp_unix = int(time.time())
            # Serialize operation_data carefully using the updated serialize method
            op_data_json = await MemoryUtils.serialize(operation_data) if operation_data is not None else None

            await conn.execute(
                """
                INSERT INTO memory_operations
                (operation_log_id, workflow_id, memory_id, action_id, operation, operation_data, timestamp)
                VALUES (?, ?, ?, ?, ?, ?, ?)
                """,
                (op_id, workflow_id, memory_id, action_id, operation, op_data_json, timestamp_unix)
            )
        except Exception as e:
            # Log failures robustly, don't let logging break main logic
            logger.error(f"CRITICAL: Failed to log memory operation '{operation}': {e}", exc_info=True)

    @staticmethod
    async def _update_memory_access(conn: aiosqlite.Connection, memory_id: str):
        """Updates the last_accessed timestamp and increments access_count for a memory. Internal helper."""
        now_unix = int(time.time())
        try:
            # Use COALESCE to handle the first access correctly
            await conn.execute(
                """
                UPDATE memories
                SET last_accessed = ?,
                    access_count = COALESCE(access_count, 0) + 1
                WHERE memory_id = ?
                """,
                (now_unix, memory_id)
            )
```

```python
        except Exception as e:
            logger.warning(f"Failed to update memory access stats for {memory_id}: {e}", exc_info=True)

# ======================================================
# Embedding Service Integration & Semantic Search Logic
# ======================================================

async def _store_embedding(conn: aiosqlite.Connection, memory_id: str, text: str) -> Optional[str]:
    """Generates and stores an embedding for a memory using the EmbeddingService.

    Args:
        conn: Database connection.
        memory_id: ID of the memory.
        text: Text content to generate embedding for (often content + description).

    Returns:
        ID of the stored embedding record in the embeddings table, or None if failed.
    """
    try:
        embedding_service = get_embedding_service() # Get singleton instance
        if not embedding_service.client: # Check if service was initialized correctly (has client)
             logger.warning("EmbeddingService client not available. Cannot generate embedding.",
             emoji_key="warning")
             return None

        # Generate embedding using the service (handles caching internally)
        embedding_array: Optional[np.ndarray] = await embedding_service.get_embedding(text)

        # Generate a unique ID for this embedding entry in our DB table
        embedding_db_id = MemoryUtils.generate_id()
        embedding_bytes = embedding_array.tobytes()
        model_used = embedding_service.default_model # Or get model used if service provides it

        # Store embedding in our DB
        await conn.execute(
            """
            INSERT INTO embeddings (id, memory_id, model, embedding, created_at)
            VALUES (?, ?, ?, ?, ?)
            ON CONFLICT(memory_id) DO UPDATE SET
                id = excluded.id,
                model = excluded.model,
                embedding = excluded.embedding,
                created_at = excluded.created_at
            """,
            (
                embedding_db_id,
                memory_id,
                model_used,
                embedding_bytes,
                int(time.time())
            )
        )

        # Update the memory record to link to this *embedding table entry ID*
        # Note: The cognitive_memory schema had embedding_id as FK to embeddings.id
        # We will store embedding_db_id here.
        await conn.execute(
            "UPDATE memories SET embedding_id = ? WHERE memory_id = ?",
            (embedding_db_id, memory_id)
        )

        logger.debug(f"Stored embedding {embedding_db_id} for memory {memory_id}")
        return embedding_db_id # Return the ID of the row in the embeddings table

    except Exception as e:
        logger.error(f"Failed to store embedding for memory {memory_id}: {e}", exc_info=True)
        return None


async def _find_similar_memories(
    conn: aiosqlite.Connection,
    query_text: str,
    workflow_id: Optional[str] = None,
    limit: int = 5,
    threshold: float = SIMILARITY_THRESHOLD,
    memory_level: Optional[str] = None,
    memory_type: Optional[str] = None # Added memory_type filter parameter
) -> List[Tuple[str, float]]:
```

```python
    """Finds memories with similar semantic meaning using embeddings stored in SQLite.
       Filters by workflow, level, type, and TTL.

    Args:
        conn: Database connection.
        query_text: Query text to find similar memories.
        workflow_id: Optional workflow ID to limit search.
        limit: Maximum number of results to return *after similarity calculation*.
        threshold: Minimum similarity score (0-1).
        memory_level: Optional memory level to filter by.
        memory_type: Optional memory type to filter by.

    Returns:
        List of tuples (memory_id, similarity_score) sorted by similarity descending.
    """
    try:
        embedding_service = get_embedding_service()
        if not embedding_service.client:
            logger.warning("EmbeddingService client not available. Cannot perform semantic search.",
            emoji_key="warning")
            return []

        # 1. Generate query embedding
        query_embedding: Optional[np.ndarray] = await embedding_service.get_embedding(query_text)
        if query_embedding is None:
            logger.warning(f"Failed to generate query embedding for: '{query_text[:50]}...'")
            return []
        query_embedding_2d = query_embedding.reshape(1, -1)

        # 2. Build query to fetch candidate embeddings from DB, including filters
        sql = """
        SELECT m.memory_id, e.embedding
        FROM memories m
        JOIN embeddings e ON m.embedding_id = e.id
        WHERE 1=1
        """
        params: List[Any] = []

        if workflow_id:
            sql += " AND m.workflow_id = ?"
            params.append(workflow_id)
        if memory_level:
            sql += " AND m.memory_level = ?"
            params.append(memory_level.lower()) # Ensure lowercase for comparison
        # *** ADDED memory_type filter directly to SQL ***
        if memory_type:
            sql += " AND m.memory_type = ?"
            params.append(memory_type.lower()) # Ensure lowercase

        # Add TTL check
        now_unix = int(time.time())
        sql += " AND (m.ttl = 0 OR m.created_at + m.ttl > ?)"
        params.append(now_unix)

        # Optimization: Potentially limit candidates fetched *before* calculating all similarities
        # Fetching more candidates than `limit` allows for better ranking after similarity calculation
        candidate_limit = max(limit * 5, 50) # Fetch more candidates than needed
        sql += " ORDER BY m.last_accessed DESC NULLS LAST LIMIT ?" # Prioritize recently accessed
        params.append(candidate_limit)

        # 3. Fetch candidate embeddings
        candidates: List[Tuple[str, bytes]] = []
        async with conn.execute(sql, params) as cursor:
            candidates = await cursor.fetchall() # Fetchall is ok for limited candidates

        if not candidates:
            logger.debug("No candidate memories found matching filters for semantic search.")
            return []

        # 4. Calculate similarities for candidates
        similarities: List[Tuple[str, float]] = []
        for memory_id, embedding_bytes in candidates:
            try:
                memory_embedding = np.frombuffer(embedding_bytes, dtype=np.float32)
                if memory_embedding.size == 0:
                    continue
                memory_embedding_2d = memory_embedding.reshape(1, -1)
```

```python
                if query_embedding_2d.shape[1] != memory_embedding_2d.shape[1]:
                    logger.warning(f"Dim mismatch for memory {memory_id}. Skipping.")
                    continue

                similarity = sk_cosine_similarity(query_embedding_2d, memory_embedding_2d)[0][0]

                # 5. Filter by threshold
                if similarity >= threshold:
                    similarities.append((memory_id, float(similarity)))

            except Exception as e:
                logger.warning(f"Error processing embedding for memory {memory_id}: {e}")
                continue

        # 6. Sort by similarity and limit to the final requested count
        similarities.sort(key=lambda x: x[1], reverse=True)

        logger.debug(f"Calculated similarities for {len(candidates)} candidates. Found {len(similarities)} memories
        above threshold {threshold} before limiting to {limit}.")
        return similarities[:limit]

    except Exception as e:
        logger.error(f"Failed to find similar memories: {e}", exc_info=True)
        return []

# ======================================================
# Public Tool Functions (Integrated & Adapted)
# ======================================================

# --- 1. Initialization ---
@with_tool_metrics
@with_error_handling
async def initialize_memory_system(db_path: str = DEFAULT_DB_PATH) -> Dict[str, Any]:
    """Initializes the Unified Agent Memory system and checks embedding service status.

    Creates or verifies the database schema using aiosqlite, applies optimizations,
    and attempts to initialize the singleton EmbeddingService, reporting its status.
    Should be called once before using other memory or embedding tools.

    Args:
        db_path: (Optional) Path to the SQLite database file.

    Returns:
        Initialization status dictionary including embedding service availability.
        {
            "success": true,
            "message": "Unified Memory System initialized successfully.",
            "db_path": "/path/to/unified_agent_memory.db",
            "embedding_service_functional": true, # Indicates if embeddings can be generated
            "embedding_service_warning": null, # Optional warning message if service is limited
            "processing_time": 0.123
        }
    """
    start_time = time.time()
    logger.info("Initializing Unified Memory System...", emoji_key="rocket")
    embedding_service_functional = False
    embedding_service_warning = None

    try:
        # Initialize/Verify Database Schema via DBConnection context manager
        async with DBConnection(db_path) as conn:
             # Perform a simple check to ensure DB connection is working
            await conn.execute("SELECT count(*) FROM workflows")
            # No explicit commit needed here if using default aiosqlite behavior or autocommit

        # Attempt to initialize/get the EmbeddingService singleton
        try:
            # This call triggers the service's __init__ if it's the first time
            embedding_service = get_embedding_service()
            # Check if the service has its client (requires API key)
            if embedding_service.client is not None:
                embedding_service_functional = True
                logger.info("EmbeddingService initialized and functional.", emoji_key="brain")
            else:
                embedding_service_warning = "EmbeddingService initialized but OpenAI API key missing or invalid.
                Embeddings disabled."
                logger.warning(embedding_service_warning, emoji_key="warning")
        except Exception as embed_init_err:
```

```python
            embedding_service_warning = f"Failed to initialize EmbeddingService: {str(embed_init_err)}. Embeddings
            disabled."
            logger.error(embedding_service_warning, emoji_key="error", exc_info=True)
            # We don't raise ToolError here, memory system can function without embeddings

        processing_time = time.time() - start_time
        logger.success("Unified Memory System database initialized successfully.", emoji_key="white_check_mark",
        time=processing_time)

        return {
            "success": True,
            "message": "Unified Memory System initialized successfully.",
            "db_path": os.path.abspath(db_path),
            "embedding_service_functional": embedding_service_functional, # Report status based on service init
            "embedding_service_warning": embedding_service_warning, # Provide details if non-functional
            "processing_time": processing_time
        }
    except Exception as e:
        # This catches errors during DB initialization primarily
        logger.error(f"Failed to initialize memory system database: {str(e)}", emoji_key="x", exc_info=True)
        # Re-raise as ToolError, indicating core system failure
        raise ToolError(f"Memory system database initialization failed: {str(e)}") from e


# --- 2. Workflow Management Tools (Ported/Adapted from agent_memory) ---
@with_tool_metrics
@with_error_handling
async def create_workflow(
    title: str,
    description: Optional[str] = None,
    goal: Optional[str] = None,
    tags: Optional[List[str]] = None,
    metadata: Optional[Dict[str, Any]] = None,
    parent_workflow_id: Optional[str] = None,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
    """Creates a new workflow, including a default thought chain and initial goal thought if specified.

    Args:
        title: A clear, descriptive title for the workflow.
        description: (Optional) A more detailed explanation of the workflow's purpose.
        goal: (Optional) The high-level goal or objective. If provided, an initial 'goal' thought is created.
        tags: (Optional) List of keyword tags to categorize this workflow.
        metadata: (Optional) Additional structured data about the workflow.
        parent_workflow_id: (Optional) ID of a parent workflow.
        db_path: (Optional) Path to the SQLite database file.

    Returns:
        Dictionary containing information about the created workflow and its primary thought chain.
        {
            "workflow_id": "uuid-string",
            "title": "Workflow Title",
            "description": "...",
            "goal": "...",
            "status": "active",
            "created_at": "iso-timestamp",
            "updated_at": "iso-timestamp",
            "tags": ["tag1"],
            "primary_thought_chain_id": "uuid-string",
            "success": true
        }

    Raises:
        ToolInputError: If title is empty or parent workflow doesn't exist.
        ToolError: If the database operation fails.
    """
    # Validate required input
    if not title or not isinstance(title, str):
        raise ToolInputError("Workflow title must be a non-empty string", param_name="title")

    # Generate IDs and timestamps
    workflow_id = MemoryUtils.generate_id()
    now_iso = datetime.utcnow().isoformat()
    now_unix = int(time.time()) # For last_active timestamp

    try:
        async with DBConnection(db_path) as conn:
            # Check parent workflow existence if provided
```

```
            if parent_workflow_id:
                async with conn.execute("SELECT 1 FROM workflows WHERE workflow_id = ?", (parent_workflow_id,)) as
                cursor:
                    if not await cursor.fetchone():
                        raise ToolInputError(f"Parent workflow not found: {parent_workflow_id}",
                        param_name="parent_workflow_id")

            # Serialize metadata
            metadata_json = await MemoryUtils.serialize(metadata)

            # Insert the main workflow record
            await conn.execute(
                """
                INSERT INTO workflows
                (workflow_id, title, description, goal, status, created_at, updated_at, parent_workflow_id, metadata,
                last_active)
                VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
                """,
                (workflow_id, title, description, goal, WorkflowStatus.ACTIVE.value,
                 now_iso, now_iso, parent_workflow_id, metadata_json, now_unix)
            )

            # Process and associate tags with the workflow
            await MemoryUtils.process_tags(conn, workflow_id, tags or [], "workflow")

            # Create the default thought chain associated with this workflow
            thought_chain_id = MemoryUtils.generate_id()
            chain_title = f"Main reasoning for: {title}" # Default title
            await conn.execute(
                "INSERT INTO thought_chains (thought_chain_id, workflow_id, title, created_at) VALUES (?, ?, ?, ?)",
                (thought_chain_id, workflow_id, chain_title, now_iso)
            )

            # If a goal was provided, add it as the first thought in the default chain
            if goal:
                thought_id = MemoryUtils.generate_id()
                # Get sequence number (will be 1 for the first thought)
                seq_no = await MemoryUtils.get_next_sequence_number(conn, thought_chain_id, "thoughts",
                "thought_chain_id")
                await conn.execute(
                    """
                    INSERT INTO thoughts
                    (thought_id, thought_chain_id, thought_type, content, sequence_number, created_at)
                    VALUES (?, ?, ?, ?, ?, ?)
                    """,
                    (thought_id, thought_chain_id, ThoughtType.GOAL.value, goal, seq_no, now_iso)
                )

            # Commit the transaction
            await conn.commit()

            # Prepare the result dictionary
            result = {
                "workflow_id": workflow_id,
                "title": title,
                "description": description,
                "goal": goal,
                "status": WorkflowStatus.ACTIVE.value,
                "created_at": now_iso,
                "updated_at": now_iso,
                "tags": tags or [],
                "primary_thought_chain_id": thought_chain_id, # Inform agent of the default chain ID
                "success": True
            }
            logger.info(f"Created workflow '{title}' ({workflow_id}) with primary thought chain {thought_chain_id}",
            emoji_key="clipboard")
            return result

    except ToolInputError:
        raise # Re-raise specific input errors
    except Exception as e:
        # Log the error and raise a generic ToolError
        logger.error(f"Error creating workflow: {e}", exc_info=True)
        raise ToolError(f"Failed to create workflow: {str(e)}") from e

@with_tool_metrics
@with_error_handling
async def update_workflow_status(
```

```python
    workflow_id: str,
    status: str,
    completion_message: Optional[str] = None,
    update_tags: Optional[List[str]] = None,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
    """Updates the status of a workflow. (Ported from agent_memory, slightly adapted)."""
    # [Implementation similar to agent_memory.update_workflow_status, using new DB schema/utils]
    # ... Includes adding completion message as a thought ...
    try:
        status_enum = WorkflowStatus(status.lower())
    except ValueError as e:
        valid_statuses = [s.value for s in WorkflowStatus]
        raise ToolInputError(f"Invalid status '{status}'. Must be one of: {', '.join(valid_statuses)}",
        param_name="status") from e

    now_iso = datetime.utcnow().isoformat()
    now_unix = int(time.time())

    try:
        async with DBConnection(db_path) as conn:
             # Check existence first
            async with conn.execute("SELECT 1 FROM workflows WHERE workflow_id = ?", (workflow_id,)) as cursor:
                if not await cursor.fetchone():
                    raise ToolInputError(f"Workflow not found: {workflow_id}", param_name="workflow_id")

            update_params = [status_enum.value, now_iso, now_unix, workflow_id]
            set_clauses = "status = ?, updated_at = ?, last_active = ?"

            if status_enum in [WorkflowStatus.COMPLETED, WorkflowStatus.FAILED, WorkflowStatus.ABANDONED]:
                set_clauses += ", completed_at = ?"
                update_params.insert(2, now_iso) # Insert completed_at timestamp

            await conn.execute(
                f"UPDATE workflows SET {set_clauses} WHERE workflow_id = ?",
                update_params
            )

            # Add completion message as thought
            if completion_message:
                 async with conn.execute("SELECT thought_chain_id FROM thought_chains WHERE workflow_id = ? ORDER BY
                 created_at ASC LIMIT 1", (workflow_id,)) as cursor:
                    row = await cursor.fetchone()
                    if row:
                        thought_chain_id = row["thought_chain_id"]
                        seq_no = await MemoryUtils.get_next_sequence_number(conn, thought_chain_id, "thoughts",
                        "thought_chain_id")
                        thought_id = MemoryUtils.generate_id()
                        thought_type = ThoughtType.SUMMARY.value if status_enum == WorkflowStatus.COMPLETED else
                        ThoughtType.REFLECTION.value
                        await conn.execute(
                            "INSERT INTO thoughts (thought_id, thought_chain_id, thought_type, content,
                            sequence_number, created_at) VALUES (?, ?, ?, ?, ?, ?)",
                            (thought_id, thought_chain_id, thought_type, completion_message, seq_no, now_iso)
                        )

            # Process additional tags
            await MemoryUtils.process_tags(conn, workflow_id, update_tags or [], "workflow")
            await conn.commit()

            result = {
                "workflow_id": workflow_id,
                "status": status_enum.value,
                "updated_at": now_iso,
                "success": True
            }
            if status_enum in [WorkflowStatus.COMPLETED, WorkflowStatus.FAILED, WorkflowStatus.ABANDONED]:
                result["completed_at"] = now_iso

            logger.info(f"Updated workflow {workflow_id} status to '{status_enum.value}'",
            emoji_key="arrows_counterclockwise")
            return result

    except ToolInputError:
        raise
    except Exception as e:
        logger.error(f"Error updating workflow status: {e}", exc_info=True)
        raise ToolError(f"Failed to update workflow status: {str(e)}") from e
```

```python
# --- 3. Action Tracking Tools (Ported/Adapted from agent_memory & Integrated) ---
@with_tool_metrics
@with_error_handling
async def record_action_start(
    workflow_id: str,
    action_type: str,
    reasoning: str,
    tool_name: Optional[str] = None,
    tool_args: Optional[Dict[str, Any]] = None,
    title: Optional[str] = None,
    parent_action_id: Optional[str] = None,
    tags: Optional[List[str]] = None,
    related_thought_id: Optional[str] = None,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
    """Records the start of an action within a workflow and creates a corresponding episodic memory.

    Use this tool whenever you begin a significant step in your workflow. It logs the action details
    and automatically creates a linked memory entry summarizing the action's initiation and reasoning.

    Args:
        workflow_id: The ID of the workflow this action belongs to.
        action_type: The type of action (e.g., 'tool_use', 'reasoning', 'planning'). See ActionType enum.
        reasoning: An explanation of why this action is being taken.
        tool_name: (Optional) The name of the tool being used (required if action_type is 'tool_use').
        tool_args: (Optional) Arguments passed to the tool (used if action_type is 'tool_use').
        title: (Optional) A brief, descriptive title for this action. Auto-generated if omitted.
        parent_action_id: (Optional) ID of parent action if this is a sub-action.
        tags: (Optional) List of tags to categorize this action.
        related_thought_id: (Optional) ID of a thought that led to this action.
        db_path: (Optional) Path to the SQLite database file.

    Returns:
        A dictionary containing information about the started action and the linked memory.

    Raises:
        ToolInputError: If required parameters are missing or invalid, or referenced entities don't exist.
        ToolError: If the database operation fails.
    """
    # --- Input Validation ---
    try:
        action_type_enum = ActionType(action_type.lower())
    except ValueError as e:
        valid_types = [t.value for t in ActionType]
        raise ToolInputError(f"Invalid action_type '{action_type}'. Must be one of: {', '.join(valid_types)}",
        param_name="action_type") from e

    if not reasoning or not isinstance(reasoning, str):
        raise ToolInputError("Reasoning must be a non-empty string", param_name="reasoning")
    if action_type_enum == ActionType.TOOL_USE and not tool_name:
        raise ToolInputError("Tool name is required for 'tool_use' action type", param_name="tool_name")

    # --- Initialization ---
    action_id = MemoryUtils.generate_id()
    memory_id = MemoryUtils.generate_id() # Pre-generate ID for the linked memory
    now_iso = datetime.utcnow().isoformat()
    now_unix = int(time.time())

    try:
        async with DBConnection(db_path) as conn:
            # --- Existence Checks (Workflow, Parent Action, Related Thought) ---
            async with conn.execute("SELECT 1 FROM workflows WHERE workflow_id = ?", (workflow_id,)) as cursor:
                if not await cursor.fetchone():
                    raise ToolInputError(f"Workflow not found: {workflow_id}", param_name="workflow_id")

            if parent_action_id:
                async with conn.execute("SELECT 1 FROM actions WHERE action_id = ? AND workflow_id = ?",
                (parent_action_id, workflow_id)) as cursor:
                    if not await cursor.fetchone():
                        raise ToolInputError(f"Parent action '{parent_action_id}' not found or does not belong to
                        workflow '{workflow_id}'.", param_name="parent_action_id")

            if related_thought_id:
                 async with conn.execute("SELECT 1 FROM thoughts t JOIN thought_chains tc ON t.thought_chain_id =
                 tc.thought_chain_id WHERE t.thought_id = ? AND tc.workflow_id = ?", (related_thought_id,
                 workflow_id)) as cursor:
```

```
                if not await cursor.fetchone():
                    raise ToolInputError(f"Related thought '{related_thought_id}' not found or does not
                    belong to workflow '{workflow_id}'.", param_name="related_thought_id")

        # --- Determine Action Title ---
        sequence_number = await MemoryUtils.get_next_sequence_number(conn, workflow_id, "actions",
        "workflow_id")
        auto_title = title
        if not auto_title:
            if action_type_enum == ActionType.TOOL_USE and tool_name:
                auto_title = f"Using {tool_name}"
            else:
                first_sentence = reasoning.split('.')[0].strip()
                auto_title = first_sentence[:50] + ("..." if len(first_sentence) > 50 else "")
        if not auto_title: # Fallback if reasoning was very short
            auto_title = f"{action_type_enum.value.capitalize()} Action #{sequence_number}"

        # --- Insert Action Record ---
        tool_args_json = await MemoryUtils.serialize(tool_args)
        await conn.execute(
            """
            INSERT INTO actions (action_id, workflow_id, parent_action_id, action_type, title,
            reasoning, tool_name, tool_args, status, started_at, sequence_number)
            VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
            """,
            (action_id, workflow_id, parent_action_id, action_type_enum.value, auto_title,
             reasoning, tool_name, tool_args_json, ActionStatus.IN_PROGRESS.value, now_iso, sequence_number)
        )

        # --- Process Tags for Action ---
        await MemoryUtils.process_tags(conn, action_id, tags or [], "action")

        # --- Link Action to Related Thought ---
        if related_thought_id:
            await conn.execute("UPDATE thoughts SET relevant_action_id = ? WHERE thought_id = ?", (action_id,
            related_thought_id))

        # --- Create Linked Episodic Memory ---
        # Construct memory content
        memory_content = f"Started action [{sequence_number}] '{auto_title}' ({action_type_enum.value}).
        Reasoning: {reasoning}"
        if tool_name:
             memory_content += f" Tool: {tool_name}."
             # Optionally include args preview in memory? Maybe too verbose.
             # if tool_args: memory_content += f" Args: {str(tool_args)[:50]}..."

        # Construct memory tags
        mem_tags = ["action_start", action_type_enum.value] + (tags or [])
        mem_tags_json = json.dumps(list(set(mem_tags))) # Ensure unique and serialize

        # Insert memory record, directly linking to the action_id
        await conn.execute(
             """
             INSERT INTO memories (memory_id, workflow_id, action_id, content, memory_level, memory_type,
             importance, confidence, tags, created_at, updated_at, access_count)
             VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
             """,
             (memory_id, workflow_id, action_id, memory_content, MemoryLevel.EPISODIC.value,
             MemoryType.ACTION_LOG.value,
              5.0, 1.0, mem_tags_json, now_unix, now_unix, 0) # Importance 5.0 for action logs
        )
        # Log memory creation operation
        await MemoryUtils._log_memory_operation(conn, workflow_id, "create_from_action_start", memory_id,
        action_id)

        # --- Update Workflow Timestamp ---
        await conn.execute("UPDATE workflows SET updated_at = ?, last_active = ? WHERE workflow_id = ?",
        (now_iso, now_unix, workflow_id))

        # --- Commit Transaction ---
        await conn.commit()

        # --- Prepare Result ---
        result = {
            "action_id": action_id,
            "workflow_id": workflow_id,
            "action_type": action_type_enum.value,
            "title": auto_title,
```

```python
                "tool_name": tool_name,
                "status": ActionStatus.IN_PROGRESS.value,
                "started_at": now_iso,
                "sequence_number": sequence_number,
                "tags": tags or [],
                "linked_memory_id": memory_id, # Provide the ID of the automatically created memory
                "success": True
            }
            logger.info(f"Started action '{auto_title}' ({action_id}) in workflow {workflow_id}",
            emoji_key="fast_forward")
            return result

    except ToolInputError:
        raise # Re-raise for specific handling
    except Exception as e:
        logger.error(f"Error recording action start: {e}", exc_info=True)
        # Attempt to rollback if conn available? aiosqlite might handle this implicitly on context exit error.
        raise ToolError(f"Failed to record action start: {str(e)}") from e


@with_tool_metrics
@with_error_handling
async def record_action_completion(
    action_id: str,
    status: str = "completed",
    tool_result: Optional[Any] = None,
    summary: Optional[str] = None,
    conclusion_thought: Optional[str] = None,
    conclusion_thought_type: str = "inference", # Default type for conclusion
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
    """Records the completion or failure of an action and updates its linked memory.

    Marks an action (previously started with record_action_start) as finished,
    stores the tool result if applicable, optionally adds a summary or concluding thought,
    and updates the corresponding 'action_log' memory entry.

    Args:
        action_id: The ID of the action to complete.
        status: (Optional) Final status: 'completed', 'failed', or 'skipped'. Default 'completed'.
        tool_result: (Optional) The result returned by the tool for 'tool_use' actions.
        summary: (Optional) A brief summary of the action's outcome or findings.
        conclusion_thought: (Optional) A thought derived from this action's completion.
        conclusion_thought_type: (Optional) Type for the conclusion thought. Default 'inference'.
        db_path: (Optional) Path to the SQLite database file.

    Returns:
        Dictionary confirming the action completion.
        {
            "action_id": "action-uuid",
            "workflow_id": "workflow-uuid",
            "status": "completed" | "failed" | "skipped",
            "completed_at": "iso-timestamp",
            "conclusion_thought_id": "thought-uuid" | None,
            "success": true
        }

    Raises:
        ToolInputError: If action not found or status/thought type is invalid.
        ToolError: If database operation fails.
    """
    start_time = time.time()

    # --- Validate Status ---
    try:
        status_enum = ActionStatus(status.lower())
        if status_enum not in [ActionStatus.COMPLETED, ActionStatus.FAILED, ActionStatus.SKIPPED]:
            # Planned and InProgress are not valid *completion* statuses
            raise ValueError("Status must indicate completion, failure, or skipping.")
    except ValueError as e:
        valid_statuses = [s.value for s in [ActionStatus.COMPLETED, ActionStatus.FAILED, ActionStatus.SKIPPED]]
        raise ToolInputError(f"Invalid completion status '{status}'. Must be one of: {', '.join(valid_statuses)}",
        param_name="status") from e

    # --- Validate Thought Type (if conclusion thought provided) ---
    thought_type_enum = None
    if conclusion_thought:
        try:
```

```python
            thought_type_enum = ThoughtType(conclusion_thought_type.lower())
        except ValueError as e:
             valid_types = [t.value for t in ThoughtType]
             raise ToolInputError(f"Invalid thought type '{conclusion_thought_type}'. Must be one of: {',
             '.join(valid_types)}", param_name="conclusion_thought_type") from e

    now_iso = datetime.utcnow().isoformat()
    now_unix = int(time.time())

    try:
        async with DBConnection(db_path) as conn:
            # --- 1. Verify Action and Get Workflow ID ---
            # Fetch workflow_id and current status to prevent re-completing
            async with conn.execute("SELECT workflow_id, status FROM actions WHERE action_id = ?", (action_id,)) as
            cursor:
                action_row = await cursor.fetchone()
                if not action_row:
                    raise ToolInputError(f"Action not found: {action_id}", param_name="action_id")
                workflow_id = action_row["workflow_id"]
                current_status = action_row["status"]
                # Optional: Prevent completing an already completed/failed/skipped action
                if current_status not in [ActionStatus.IN_PROGRESS.value, ActionStatus.PLANNED.value]:
                     logger.warning(f"Action {action_id} already has terminal status '{current_status}'. Allowing
                     update anyway.")
                     # raise ToolInputError(f"Action {action_id} cannot be completed, current status is
                     '{current_status}'.")

            # --- 2. Update Action Record ---
            tool_result_json = await MemoryUtils.serialize(tool_result)
            await conn.execute(
                """
                UPDATE actions
                SET status = ?,
                    completed_at = ?,
                    tool_result = ?
                WHERE action_id = ?
                """,
                (status_enum.value, now_iso, tool_result_json, action_id)
            )

            # --- 3. Update Workflow Timestamp ---
            await conn.execute(
                "UPDATE workflows SET updated_at = ?, last_active = ? WHERE workflow_id = ?",
                (now_iso, now_unix, workflow_id)
            )

            # --- 4. Add Conclusion Thought (if provided) ---
            conclusion_thought_id = None
            if conclusion_thought and thought_type_enum:
                # Find the primary thought chain for the workflow
                async with conn.execute("SELECT thought_chain_id FROM thought_chains WHERE workflow_id = ? ORDER BY
                created_at ASC LIMIT 1", (workflow_id,)) as cursor:
                    chain_row = await cursor.fetchone()
                    if chain_row:
                        thought_chain_id = chain_row["thought_chain_id"]
                        # Get next sequence number within the chain
                        seq_no = await MemoryUtils.get_next_sequence_number(conn, thought_chain_id, "thoughts",
                        "thought_chain_id")
                        conclusion_thought_id = MemoryUtils.generate_id()
                        # Insert the new thought, linking it to the completed action
                        await conn.execute(
                            """
                            INSERT INTO thoughts
                                (thought_id, thought_chain_id, thought_type, content, sequence_number, created_at,
                                relevant_action_id)
                            VALUES (?, ?, ?, ?, ?, ?, ?)
                            """,
                            (conclusion_thought_id, thought_chain_id, thought_type_enum.value, conclusion_thought,
                            seq_no, now_iso, action_id)
                        )
                        logger.debug(f"Recorded conclusion thought {conclusion_thought_id} for action {action_id}")
                    else:
                         logger.warning(f"Could not find primary thought chain for workflow {workflow_id} to add
                         conclusion thought.")

            # --- 5. Update Linked Episodic Memory ---
            # Find the 'action_log' memory created when the action started
```

```python
            async with conn.execute("SELECT memory_id, content FROM memories WHERE action_id = ? AND memory_type =
            ?", (action_id, MemoryType.ACTION_LOG.value)) as cursor:
                 memory_row = await cursor.fetchone()
                 if memory_row:
                    memory_id = memory_row["memory_id"]
                    original_content = memory_row["content"]

                    # Build the update text to append
                    update_parts = [f"Completed ({status_enum.value})."]
                    if summary:
                        update_parts.append(f"Summary: {summary}")
                    if tool_result is not None:
                        # Include a concise representation of the result type/presence
                        if isinstance(tool_result, dict):
                             update_parts.append(f"Result: [Dict with {len(tool_result)} keys]")
                        elif isinstance(tool_result, list):
                             update_parts.append(f"Result: [List with {len(tool_result)} items]")
                        elif tool_result:
                             update_parts.append("Result: Success")
                        elif tool_result is False:
                             update_parts.append("Result: Failure")
                        else:
                            update_parts.append("Result obtained.") # Generic confirmation

                    update_text = " ".join(update_parts)
                    new_content = original_content + " " + update_text # Append with space

                    # Adjust importance based on outcome
                    importance_mult = 1.0
                    if status_enum == ActionStatus.FAILED:
                        importance_mult = 1.2 # Failed actions might be important to learn from
                    elif status_enum == ActionStatus.SKIPPED:
                        importance_mult = 0.8 # Skipped actions are less important

                    # Update the memory record
                    await conn.execute(
                        """
                        UPDATE memories
                        SET content = ?,
                            importance = importance * ?,
                            updated_at = ?
                        WHERE memory_id = ?
                        """,
                        (new_content, importance_mult, now_unix, memory_id)
                    )
                    # Log the memory update operation
                    await MemoryUtils._log_memory_operation(conn, workflow_id, "update_from_action_completion",
                    memory_id, action_id, {"status": status_enum.value, "summary_added": bool(summary)})
                    logger.debug(f"Updated linked memory {memory_id} for completed action {action_id}")
                 else:
                      logger.warning(f"Could not find corresponding action_log memory for completed action
                      {action_id} to update.")

            # --- 6. Commit Transaction ---
            await conn.commit()

            # --- 7. Prepare Result ---
            result = {
                "action_id": action_id,
                "workflow_id": workflow_id,
                "status": status_enum.value,
                "completed_at": now_iso,
                "conclusion_thought_id": conclusion_thought_id, # Include if one was created
                "success": True,
                "processing_time": time.time() - start_time # Calculate duration
            }
            logger.info(f"Completed action {action_id} with status {status_enum.value}",
            emoji_key="white_check_mark", duration=result["processing_time"])
            return result

    except ToolInputError:
        raise # Re-raise specific input errors
    except Exception as e:
        logger.error(f"Error recording action completion for {action_id}: {e}", exc_info=True)
        raise ToolError(f"Failed to record action completion: {str(e)}") from e


@with_tool_metrics
```

```python
async def _example():
    db = "test_unified_memory.db"
    if os.path.exists(db):
        os.remove(db)

    init_result = await initialize_memory_system(db_path=db)
    print("Init Result:", init_result)

    wf_result = await create_workflow(title="Test Analysis Workflow", goal="Analyze test data", tags=["testing",
    "example"], db_path=db)
    wf_id = wf_result["workflow_id"]
    print("\nWorkflow Created:", wf_result)

    thought1 = await record_thought(workflow_id=wf_id, content="Need to load the data first.", thought_type="plan",
    db_path=db)
    print("\nThought Recorded:", thought1)

    action1_start = await record_action_start(workflow_id=wf_id, action_type="tool_use", reasoning="Load data from
    file.", tool_name="load_data", tool_args={"file": "data.csv"}, title="Load Data", tags=["io"],
    related_thought_id=thought1["thought_id"], db_path=db)
    action1_id = action1_start["action_id"]
    print("\nAction Started:", action1_start)

    # Simulate tool execution
    await asyncio.sleep(0.1)
    tool_output = {"rows_loaded": 100, "columns": ["A", "B"]}

    action1_end = await record_action_completion(action_id=action1_id, tool_result=tool_output, summary="Data loaded
    successfully.", db_path=db)
    print("\nAction Completed:", action1_end)

    artifact1 = await record_artifact(workflow_id=wf_id, action_id=action1_id, name="Loaded Data Sample",
    artifact_type="json", content=json.dumps(tool_output), description="Sample of loaded data structure",
    tags=["data"], db_path=db)
    print("\nArtifact Recorded:", artifact1)

    mem1 = await store_memory(workflow_id=wf_id, content="Column A seems to be numerical.",
    memory_type="observation", importance=6.0, action_id=action1_id, db_path=db)
    print("\nMemory Stored:", mem1)
    mem2 = await store_memory(workflow_id=wf_id, content="Column B looks categorical.", memory_type="observation",
    importance=6.0, action_id=action1_id, db_path=db)
    print("\nMemory Stored:", mem2)

    link1 = await create_memory_link(source_memory_id=mem1["memory_id"], target_memory_id=mem2["memory_id"],
    link_type="related", db_path=db)
    print("\nMemory Link Created:", link1)

    mem_get = await get_memory_by_id(memory_id=mem1["memory_id"], include_links=True, db_path=db)
    print("\nGet Memory By ID:", mem_get)

    # Close the connection if using singleton pattern on app shutdown
    if DBConnection._instance:
        await DBConnection._instance.close()
        DBConnection._instance = None

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

B Appendix B: Agent Master Loop (AML) Code

Listing 2: Complete Code for Agent Master Loop (agent_master_loop.py)

```python
"""
Supercharged Agent Master Loop - v3.3 (Tiers 1, 2 & 3 Integration - Complete)
================================================================================

Enhanced orchestrator for AI agents using the Unified Memory System
via the Ultimate MCP Client. Implements structured planning, dynamic context,
dependency checking, artifact tracking, error recovery, feedback loops,
meta-cognition, richer auto-linking, hybrid search, direct memory management,
working memory context, **adaptive thresholds**, **memory maintenance**,
and **custom thought chain management**.
```

```python
@with_error_handling
async def get_action_details(
    action_id: Optional[str] = None,
    action_ids: Optional[List[str]] = None,
    include_dependencies: bool = False,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
    """Retrieves detailed information about one or more actions.

    Fetch complete details about specific actions by their IDs, either individually
    or in batch. Optionally includes information about action dependencies.

    Args:
        action_id: ID of a single action to retrieve (ignored if action_ids is provided)
        action_ids: Optional list of action IDs to retrieve in batch
        include_dependencies: Whether to include dependency information for each action
        db_path: Path to the SQLite database file

    Returns:
        Dictionary containing action details:
        {
            "actions": [
                {
                    "action_id": "uuid-string",
                    "workflow_id": "workflow-uuid",
                    "action_type": "tool_use",
                    "status": "completed",
                    "title": "Load data",
                    ... other action fields ...
                    "dependencies": { # Only if include_dependencies=True
                        "depends_on": ["action-id-1", "action-id-2"],
                        "dependent_actions": ["action-id-3"]
                    }
                },
                ... more actions if batch ...
            ],
            "success": true,
            "processing_time": 0.123
        }

    Raises:
        ToolInputError: If neither action_id nor action_ids is provided, or if no matching actions found
        ToolError: If database operation fails
    """
    start_time = time.time()

    # Validate inputs
    if not action_id and not action_ids:
        raise ToolInputError("Either action_id or action_ids must be provided", param_name="action_id")

    # Convert single action_id to list if action_ids not provided
    target_action_ids = action_ids or [action_id]

    try:
        async with DBConnection(db_path) as conn:
            # Prepare query - query all fields from actions plus tags
            placeholders = ', '.join(['?'] * len(target_action_ids))
            select_query = f"""
                SELECT a.*, GROUP_CONCAT(t.name) as tags_str
                FROM actions a

                # Deserialize JSON fields
                if action_data.get("tool_args"):
                    action_data["tool_args"] = await MemoryUtils.deserialize(action_data["tool_args"])

                if action_data.get("tool_result"):
                    action_data["tool_result"] = await MemoryUtils.deserialize(action_data["tool_result"])
```

```python
Designed for Claude 3.7 Sonnet (or comparable models with tool use).
"""

import asyncio
import dataclasses
import json
import logging
import math  # For adaptive threshold adjustments
import os
import random
import re
import signal
import sys
import time
from collections import defaultdict
from dataclasses import dataclass, field
from datetime import datetime, timezone
from pathlib import Path
from typing import Any, Dict, List, Optional, Set, Tuple, Union

# External Libraries
import aiofiles
import anthropic
from anthropic.types import AsyncAnthropic, Message
from pydantic import BaseModel, Field, ValidationError

# --- IMPORT YOUR ACTUAL MCP CLIENT and COMPONENTS ---
try:
    from mcp_client import (
        ActionStatus,
        ActionType,
        ArtifactType,  # noqa: F401
        LinkType,
        MCPClient,
        MemoryLevel,
        MemoryType,
        MemoryUtils,
        ThoughtType,
        ToolError,
        ToolInputError,
        WorkflowStatus,
    )
    MCP_CLIENT_AVAILABLE = True
    log = logging.getLogger("SuperchargedAgentMasterLoop") # Use project logger
    if MCPClient sets it up
    if not log.handlers:
        # Basic fallback logger if MCPClient didn't configure it
        logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(name)s
        - %(levelname)s - %(message)s')
        log = logging.getLogger("SuperchargedAgentMasterLoop")
```

```
        log.warning("MCPClient did not configure logger, using basic fallback.")
    log.info("Successfully imported MCPClient and required components.")
except ImportError as import_err:
    print(f"|\\xmark| CRITICAL ERROR: Could not import MCPClient or required
    components: {import_err}")
    print("Ensure mcp_client.py is correctly structured and in the Python
    path.")
    sys.exit(1)

# --- Logging Setup Refinement ---
log_level_str = os.environ.get("AGENT_LOOP_LOG_LEVEL", "INFO").upper()
log_level = getattr(logging, log_level_str, logging.INFO)
log.setLevel(log_level)
if log_level <= logging.DEBUG: # Use <= DEBUG to include DEBUG level
    log.info("Agent loop verbose logging enabled.")

# --- Constants ---
# File Paths & Identifiers
AGENT_STATE_FILE = "agent_loop_state_v3.3.json" # Updated state file version
AGENT_NAME = "Maestro-v3.3" # Updated agent name
# --- Meta-cognition & Maintenance Intervals/Thresholds ---
# Base Thresholds (These become the initial values and bounds)
BASE_REFLECTION_THRESHOLD = int(os.environ.get("BASE_REFLECTION_THRESHOLD",
"7"))
BASE_CONSOLIDATION_THRESHOLD =
int(os.environ.get("BASE_CONSOLIDATION_THRESHOLD", "12"))
# Adaptive Threshold Bounds
MIN_REFLECTION_THRESHOLD = 3
MAX_REFLECTION_THRESHOLD = 15
MIN_CONSOLIDATION_THRESHOLD = 5
MAX_CONSOLIDATION_THRESHOLD = 25
# Intervals
OPTIMIZATION_LOOP_INTERVAL = int(os.environ.get("OPTIMIZATION_INTERVAL", "8"))
MEMORY_PROMOTION_LOOP_INTERVAL = int(os.environ.get("PROMOTION_INTERVAL", "15"))
STATS_ADAPTATION_INTERVAL = int(os.environ.get("STATS_ADAPTATION_INTERVAL",
"10")) # How often to check stats/adapt
MAINTENANCE_INTERVAL = int(os.environ.get("MAINTENANCE_INTERVAL", "50")) # How
often to run cleanup
# Other
AUTO_LINKING_DELAY_SECS = (1.5, 3.0)
# Context & Planning
DEFAULT_PLAN_STEP = "Assess goal, gather context, formulate initial plan."
CONTEXT_RECENT_ACTIONS = 7
CONTEXT_IMPORTANT_MEMORIES = 5
CONTEXT_KEY_THOUGHTS = 5
CONTEXT_PROCEDURAL_MEMORIES = 2
CONTEXT_MAX_TOKENS_COMPRESS_THRESHOLD = 15000
CONTEXT_COMPRESSION_TARGET_TOKENS = 5000
CONTEXT_PROACTIVE_MEMORIES = 3
CONTEXT_WORKING_MEMORY_LIMIT = 10
# Error Handling
```

```python
MAX_CONSECUTIVE_ERRORS = 3
# --- Tool Names (Includes Tier 1, 2 & 3) ---
TOOL_GET_CONTEXT = "unified_memory:get_workflow_context"
TOOL_CREATE_WORKFLOW = "unified_memory:create_workflow"
TOOL_UPDATE_WORKFLOW_STATUS = "unified_memory:update_workflow_status"
TOOL_RECORD_ACTION_START = "unified_memory:record_action_start"
TOOL_RECORD_ACTION_COMPLETION = "unified_memory:record_action_completion"
TOOL_GET_ACTION_DETAILS = "unified_memory:get_action_details"
# Action Dependency Tools (Tier 1)
TOOL_ADD_ACTION_DEPENDENCY = "unified_memory:add_action_dependency"
TOOL_GET_ACTION_DEPENDENCIES = "unified_memory:get_action_dependencies"
# Artifact Tracking Tools (Tier 1)
TOOL_RECORD_ARTIFACT = "unified_memory:record_artifact"
TOOL_GET_ARTIFACTS = "unified_memory:get_artifacts"
TOOL_GET_ARTIFACT_BY_ID = "unified_memory:get_artifact_by_id"
# Core Memory Tools (Tier 2 additions)
TOOL_HYBRID_SEARCH = "unified_memory:hybrid_search_memories"
TOOL_STORE_MEMORY = "unified_memory:store_memory"
TOOL_UPDATE_MEMORY = "unified_memory:update_memory"
TOOL_GET_WORKING_MEMORY = "unified_memory:get_working_memory"
# Custom Thought Chain Tools (Tier 3)
TOOL_CREATE_THOUGHT_CHAIN = "unified_memory:create_thought_chain"
TOOL_GET_THOUGHT_CHAIN = "unified_memory:get_thought_chain"
# Maintenance Tool (Tier 3)
TOOL_DELETE_EXPIRED_MEMORIES = "unified_memory:delete_expired_memories"
# Statistics Tool (Tier 3)
TOOL_COMPUTE_STATS = "unified_memory:compute_memory_statistics"
# Other Memory Tools
TOOL_RECORD_THOUGHT = "unified_memory:record_thought"
TOOL_REFLECTION = "unified_memory:generate_reflection"
TOOL_CONSOLIDATION = "unified_memory:consolidate_memories"
TOOL_OPTIMIZE_WM = "unified_memory:optimize_working_memory"
TOOL_AUTO_FOCUS = "unified_memory:auto_update_focus"
TOOL_PROMOTE_MEM = "unified_memory:promote_memory_level"
TOOL_QUERY_MEMORIES = "unified_memory:query_memories"
TOOL_SEMANTIC_SEARCH = "unified_memory:search_semantic_memories"
TOOL_CREATE_LINK = "unified_memory:create_memory_link"
TOOL_GET_MEMORY_BY_ID = "unified_memory:get_memory_by_id"
TOOL_GET_LINKED_MEMORIES = "unified_memory:get_linked_memories"
TOOL_LIST_WORKFLOWS = "unified_memory:list_workflows"
TOOL_GENERATE_REPORT = "unified_memory:generate_workflow_report"
TOOL_SUMMARIZE_TEXT = "unified_memory:summarize_text"


# --- Structured Plan Model ---
class PlanStep(BaseModel):
    id: str = Field(default_factory=lambda:
    f"step-{MemoryUtils.generate_id()[:8]}")
    description: str
    status: str = Field(default="planned", description="Status: planned,
    in_progress, completed, failed, skipped")
```

```python
    depends_on: List[str] = Field(default_factory=list, description="List of
    action IDs this step requires")
    assigned_tool: Optional[str] = None
    tool_args: Optional[Dict[str, Any]] = None
    result_summary: Optional[str] = None
    is_parallel_group: Optional[str] = None

# --- Agent State Dataclass (Added Tier 3 State) ---
def _default_tool_stats():
    return defaultdict(lambda: {"success": 0, "failure": 0, "latency_ms_total":
    0.0})

@dataclass
class AgentState:
    # Core State
    workflow_id: Optional[str] = None
    context_id: Optional[str] = None
    workflow_stack: List[str] = field(default_factory=list)
    current_plan: List[PlanStep] = field(default_factory=lambda:
    [PlanStep(description=DEFAULT_PLAN_STEP)])
    current_sub_goal_id: Optional[str] = None # ID for the active goal/sub-goal
    current_thought_chain_id: Optional[str] = None # Track active thought chain
    (Tier 3)
    last_action_summary: str = "Loop initialized."
    current_loop: int = 0
    goal_achieved_flag: bool = False
    # Error & Replanning State
    consecutive_error_count: int = 0
    needs_replan: bool = False
    last_error_details: Optional[Dict] = None
    # Meta-Cognition State
    successful_actions_since_reflection: int = 0
    successful_actions_since_consolidation: int = 0
    loops_since_optimization: int = 0
    loops_since_promotion_check: int = 0
    loops_since_stats_adaptation: int = 0 # Tier 3
    loops_since_maintenance: int = 0 # Tier 3
    reflection_cycle_index: int = 0
    last_meta_feedback: Optional[str] = None
    # --- Adaptive Thresholds (Tier 3) ---
    current_reflection_threshold: int = BASE_REFLECTION_THRESHOLD
    current_consolidation_threshold: int = BASE_CONSOLIDATION_THRESHOLD
    # Stats & Tracking
    tool_usage_stats: Dict[str, Dict[str, Union[int, float]]] =
    field(default_factory=_default_tool_stats)
    # Background task tracking (transient, not saved)
    background_tasks: Set[asyncio.Task] = field(default_factory=set, init=False,
    repr=False)


# --- Agent Loop Class (Modified for Tier 1, 2 & 3) ---
```

```python
class AgentMasterLoop:
    """Supercharged orchestrator implementing Tier 1, 2 & 3 UMS
    enhancements."""

    def __init__(self, mcp_client_instance: MCPClient, agent_state_file: str =
    AGENT_STATE_FILE):
        if not MCP_CLIENT_AVAILABLE: raise RuntimeError("MCPClient class
        unavailable.")

        self.mcp_client = mcp_client_instance
        self.anthropic_client = self.mcp_client.anthropic
        self.logger = log
        self.agent_state_file = Path(agent_state_file)

        # Config attributes (Base values, dynamic ones are in state)
        self.consolidation_memory_level = MemoryLevel.EPISODIC.value
        self.consolidation_max_sources = 10
        self.auto_linking_threshold = 0.7
        self.auto_linking_max_links = 3
        self.reflection_type_sequence = ["summary", "progress", "gaps",
        "strengths", "plan"]

        if not self.anthropic_client:
            self.logger.critical("Anthropic client unavailable! Agent cannot
            function.")
            raise ValueError("Anthropic client required.")

        self.state = AgentState() # Initialize state here
        self._shutdown_event = asyncio.Event()
        self.tool_schemas: List[Dict[str, Any]] = []
        self._active_tasks: Set[asyncio.Task] = set()

    async def initialize(self) -> bool:
        """Initializes loop state, loads previous state, verifies client
        setup, including Tier 1, 2 & 3 tools."""
        self.logger.info("Initializing agent loop...", emoji_key="gear")
        await self._load_agent_state()
        if self.state.workflow_id and not self.state.context_id:
            self.state.context_id = self.state.workflow_id
            self.logger.info(f"Set context_id to match loaded workflow_id:
            {self.state.workflow_id}")

        try:
            if not self.mcp_client.server_manager:
                self.logger.error("MCP Client Server Manager not initialized.")
                return False

            # Fetch and filter tool schemas
            all_tools_for_api =
            self.mcp_client.server_manager.format_tools_for_anthropic()
            self.tool_schemas = [
```

```
            schema for schema in all_tools_for_api
            if self.mcp_client.server_manager.sanitized_to_original.get(sch
            ema['name'],
            '').startswith("unified_memory:")
        ]
        loaded_tool_names =
        {self.mcp_client.server_manager.sanitized_to_original.get(s['name'])
        for s in self.tool_schemas}
        self.logger.info(f"Loaded {len(self.tool_schemas)} unified_memory
        tool schemas: {loaded_tool_names}", emoji_key="clipboard")

        # Verify essential tools (Added Tier 1, 2 & 3)
        essential_tools = [
            TOOL_GET_CONTEXT, TOOL_CREATE_WORKFLOW, TOOL_RECORD_THOUGHT,
            TOOL_RECORD_ACTION_START, TOOL_RECORD_ACTION_COMPLETION,
            TOOL_ADD_ACTION_DEPENDENCY, TOOL_RECORD_ARTIFACT,
            TOOL_GET_ACTION_DETAILS,
            TOOL_STORE_MEMORY, TOOL_UPDATE_MEMORY, TOOL_GET_WORKING_MEMORY,
            TOOL_HYBRID_SEARCH,
            TOOL_CREATE_THOUGHT_CHAIN, TOOL_GET_THOUGHT_CHAIN, # Tier 3
            TOOL_COMPUTE_STATS, TOOL_DELETE_EXPIRED_MEMORIES # Tier 3
        ]
        missing_essential = [t for t in essential_tools if not
        self._find_tool_server(t)]
        if missing_essential:
            self.logger.error(f"Missing essential tools:
            {missing_essential}. Agent functionality WILL BE impaired.")

        # Check workflow validity
        current_workflow_id = self.state.workflow_stack[-1] if
        self.state.workflow_stack else self.state.workflow_id
        if current_workflow_id and not await
        self._check_workflow_exists(current_workflow_id):
            self.logger.warning(f"Loaded workflow {current_workflow_id} not
            found. Resetting state.")
            await self._reset_state_to_defaults()
            await self._save_agent_state()

        # Initialize current thought chain ID if workflow exists but chain
        ID is missing
        if self.state.workflow_id and not
        self.state.current_thought_chain_id:
             await self._set_default_thought_chain_id()

        self.logger.info("Agent loop initialized successfully.")
        return True
    except Exception as e:
        self.logger.critical(f"Agent loop initialization failed: {e}",
        exc_info=True)
        return False
```

```python
async def _set_default_thought_chain_id(self):
    """Sets the current_thought_chain_id to the primary chain of the
    current workflow."""
    current_wf_id = self.state.workflow_stack[-1] if
    self.state.workflow_stack else self.state.workflow_id
    if not current_wf_id: return # No workflow active

    # Use the get_workflow_details tool to find the primary chain
    get_details_tool = "unified_memory:get_workflow_details" # Use the
    correct tool name constant if defined, or the string directly

    # Check if the tool is available
    if self._find_tool_server(get_details_tool):
        try:
            details = await self._execute_tool_call_internal(
                get_details_tool,
                {
                    "workflow_id": current_wf_id,
                    "include_thoughts": True, # Need thoughts to get chain
                    ID
                    "include_actions": False,
                    "include_artifacts": False,
                    "include_memories": False
                },
                record_action=False
            )
            # Check successful execution AND if thought_chains exist in
            the result
            if details.get("success") and
            isinstance(details.get("thought_chains"), list) and
            details["thought_chains"]:
                # Assume the first chain listed is the primary one
                first_chain = details["thought_chains"][0]
                chain_id = first_chain.get("thought_chain_id")
                if chain_id:
                    self.state.current_thought_chain_id = chain_id
                    self.logger.info(f"Set current_thought_chain_id to
                    primary chain: {self.state.current_thought_chain_id}")
                    return # Success

            self.logger.warning(f"Could not find primary thought chain in
            details for workflow {current_wf_id}. Using default logic.")

        except Exception as e:
            self.logger.error(f"Error fetching workflow details to set
            default thought chain ID: {e}", exc_info=False)
    else:
        self.logger.warning(f"Cannot set default thought chain ID: Tool
        '{get_details_tool}' unavailable.")

    # Fallback message if chain couldn't be set
```

```python
        self.logger.info("Could not determine primary thought chain ID on
        init/load. Will use default on first thought recording.")

    async def _estimate_tokens_anthropic(self, data: Any) -> int:
        """Estimates token count for arbitrary data structures using the
        Anthropic client."""
        if data is None: return 0
        if not self.anthropic_client:
             self.logger.warning("Cannot estimate tokens: Anthropic client not
             available.")
             try: return len(json.dumps(data, default=str)) // 4
             except Exception: return 0
        token_count = 0
        try:
            if isinstance(data, str): text_representation = data
            else: text_representation = json.dumps(data, ensure_ascii=False,
            default=str)
            token_count = await
            self.anthropic_client.count_tokens(text_representation)
            return token_count
        except anthropic.APIError as e: self.logger.warning(f"Anthropic API
        error during token counting: {e}. Using fallback estimate.")
        except Exception as e: self.logger.warning(f"Token estimation failed
        for data type {type(data)}: {e}. Using fallback estimate.")
        try:
             text_representation = json.dumps(data, default=str) if not
             isinstance(data, str) else data
             return len(text_representation) // 4
        except Exception: return 0

    async def _save_agent_state(self):
        """Saves the agent loop's state to a JSON file."""
        state_dict = dataclasses.asdict(self.state)
        state_dict["timestamp"] = datetime.now(timezone.utc).isoformat()
        state_dict.pop("background_tasks", None)
        state_dict["tool_usage_stats"] = {k: dict(v) for k, v in
        self.state.tool_usage_stats.items()}
        state_dict["current_plan"] = [step.model_dump() for step in
        self.state.current_plan]
        try:
            self.agent_state_file.parent.mkdir(parents=True, exist_ok=True)
            async with aiofiles.open(self.agent_state_file, 'w') as f: await
            f.write(json.dumps(state_dict, indent=2))
            self.logger.debug(f"Agent state saved to {self.agent_state_file}")
        except Exception as e: self.logger.error(f"Failed to save agent state:
        {e}", exc_info=True)

    async def _load_agent_state(self):
        """Loads state, converting plan back to PlanStep objects and setting
        dynamic thresholds."""
        if not self.agent_state_file.exists():
```

```
        self.logger.info("No previous agent state file found. Using default
        state.")
        self.state = AgentState(
            current_reflection_threshold=BASE_REFLECTION_THRESHOLD,
            current_consolidation_threshold=BASE_CONSOLIDATION_THRESHOLD
        ) # Initialize dynamic thresholds
        return
    try:
        async with aiofiles.open(self.agent_state_file, 'r') as f:
        state_data = json.loads(await f.read())
        kwargs = {}
        for field_info in dataclasses.fields(AgentState):
             if field_info.name in state_data:
                 if field_info.name == "current_plan":
                     try: kwargs["current_plan"] = [PlanStep(**step_data)
                     for step_data in state_data["current_plan"]]
                     except (ValidationError, TypeError) as plan_err:
                     log.warning(f"Failed to parse saved plan, resetting:
                     {plan_err}"); kwargs["current_plan"] =
                     [PlanStep(description=DEFAULT_PLAN_STEP)]
                 elif field_info.name == "tool_usage_stats":
                      stats_dict = state_data["tool_usage_stats"]
                      recreated_stats = defaultdict(lambda: {"success": 0,
                      "failure": 0, "latency_ms_total": 0.0})
                      if isinstance(stats_dict, dict):
                          for k, v in stats_dict.items():
                              if isinstance(v, dict): recreated_stats[k] =
                              {"success": v.get("success", 0), "failure":
                              v.get("failure", 0), "latency_ms_total":
                              v.get("latency_ms_total", 0.0)}
                      kwargs["tool_usage_stats"] = recreated_stats
                 else: kwargs[field_info.name] = state_data[field_info.name]
             else:
                 # Initialize dynamic thresholds if not present in saved
                 state
                 if field_info.name == "current_reflection_threshold":
                 kwargs[field_info.name] = BASE_REFLECTION_THRESHOLD
                 elif field_info.name == "current_consolidation_threshold":
                 kwargs[field_info.name] = BASE_CONSOLIDATION_THRESHOLD
                 elif field_info.default_factory is not
                 dataclasses.MISSING: kwargs[field_info.name] =
                 field_info.default_factory()
                 elif field_info.default is not dataclasses.MISSING:
                 kwargs[field_info.name] = field_info.default

        # Ensure dynamic thresholds exist even if loading older state file
        if "current_reflection_threshold" not in kwargs:
        kwargs["current_reflection_threshold"] = BASE_REFLECTION_THRESHOLD
        if "current_consolidation_threshold" not in kwargs:
        kwargs["current_consolidation_threshold"] =
        BASE_CONSOLIDATION_THRESHOLD
```

```
            self.state = AgentState(**kwargs)
            self.logger.info(f"Agent state loaded from {self.agent_state_file}.
            Loop {self.state.current_loop}. WF: {self.state.workflow_id}. Dyn
            Thresh: R={self.state.current_reflection_threshold}/C={self.state.c
            urrent_consolidation_threshold}")
        except Exception as e: self.logger.error(f"Failed to load/parse agent
        state: {e}. Resetting.", exc_info=True); await
        self._reset_state_to_defaults()

    async def _reset_state_to_defaults(self):
        self.state = AgentState(
             current_reflection_threshold=BASE_REFLECTION_THRESHOLD,
             current_consolidation_threshold=BASE_CONSOLIDATION_THRESHOLD
        )
        self.logger.warning("Agent state has been reset to defaults.")

    # --- Context Gathering (Enhanced for Tier 2) ---
    async def _gather_context(self) -> Dict[str, Any]:
        """Gathers comprehensive context for the agent LLM, including working
        memory and using hybrid search."""
        self.logger.info("Gathering context...", emoji_key="satellite")
        base_context = {
            "current_loop": self.state.current_loop,
            "current_plan": [step.model_dump() for step in
            self.state.current_plan],
            "last_action_summary": self.state.last_action_summary,
            "consecutive_errors": self.state.consecutive_error_count,
            "last_error_details": self.state.last_error_details,
            "workflow_stack": self.state.workflow_stack,
            "meta_feedback": self.state.last_meta_feedback,
            "current_thought_chain_id": self.state.current_thought_chain_id, #
            Added Tier 3
            "core_context": None,
            "current_working_memory": [], # Added Tier 2
            "proactive_memories": [],
            "relevant_procedures": [],
            "contextual_links": None,
            "compression_summary": None,
            "status": "Gathering...",
            "errors": []
        }
        self.state.last_meta_feedback = None

        current_workflow_id = self.state.workflow_stack[-1] if
        self.state.workflow_stack else self.state.workflow_id
        current_context_id = self.state.context_id

        if not current_workflow_id:
```

```
    base_context["status"] = "No Active Workflow";
    base_context["message"] = "Create/load workflow."; return
    base_context

# --- 0. Get Current Working Memory (Tier 2) ---
if current_context_id and
self._find_tool_server(TOOL_GET_WORKING_MEMORY):
     try:
         wm_result = await
         self._execute_tool_call_internal(TOOL_GET_WORKING_MEMORY,
         {"context_id": current_context_id, "include_content": False,
         "include_links": False}, record_action=False)
         if wm_result.get("success"):
             wm_mems = wm_result.get("working_memories", []);
             base_context["current_working_memory"] = [{"memory_id":
             m.get("memory_id"), "description": m.get("description"),
             "type": m.get("memory_type"), "importance":
             m.get("importance")} for m in
             wm_mems[:CONTEXT_WORKING_MEMORY_LIMIT]];
             self.logger.info(f"Retrieved
             {len(base_context['current_working_memory'])} items from
             working memory for context {current_context_id}.")
         else: base_context["errors"].append(f"Working memory retrieval
         failed: {wm_result.get('error')}")
     except Exception as e: self.logger.warning(f"Working memory
     retrieval exception: {e}"); base_context["errors"].append(f"Working
     memory retrieval exception: {e}")
elif not current_context_id: self.logger.debug("Skipping working memory
retrieval: No context_id set.")
else: self.logger.debug(f"Skipping working memory retrieval: Tool
'{TOOL_GET_WORKING_MEMORY}' unavailable.")

# --- 1. Goal-Directed Proactive Memory Retrieval (Using Hybrid Search
- Tier 2) ---
active_plan_step_desc = self.state.current_plan[0].description if
self.state.current_plan else "Achieve main goal"
proactive_query = f"Information relevant to planning or executing:
{active_plan_step_desc}"
search_tool_proactive = TOOL_HYBRID_SEARCH if
self._find_tool_server(TOOL_HYBRID_SEARCH) else TOOL_SEMANTIC_SEARCH
if self._find_tool_server(search_tool_proactive):
    search_args = {"workflow_id": current_workflow_id, "query":
    proactive_query, "limit": CONTEXT_PROACTIVE_MEMORIES,
    "include_content": False}
    if search_tool_proactive == TOOL_HYBRID_SEARCH:
    search_args.update({"semantic_weight": 0.7, "keyword_weight": 0.3})
    try:
        result_content = await
        self._execute_tool_call_internal(search_tool_proactive,
        search_args, record_action=False)
        if result_content.get("success"):
```

```
                proactive_mems = result_content.get("memories", []);
                score_key = "hybrid_score" if search_tool_proactive ==
                TOOL_HYBRID_SEARCH else "similarity"
                base_context["proactive_memories"] = [{"memory_id":
                m.get("memory_id"), "description": m.get("description"),
                "score": m.get(score_key), "type": m.get("memory_type")} for
                m in proactive_mems]
                if base_context["proactive_memories"]:
                self.logger.info(f"Retrieved
                {len(base_context['proactive_memories'])} proactive memories
                using {search_tool_proactive.split(':')[-1]}.")
            else: base_context["errors"].append(f"Proactive memory search
            failed: {result_content.get('error')}")
        except Exception as e: self.logger.warning(f"Proactive memory
        search exception: {e}"); base_context["errors"].append(f"Proactive
        search exception: {e}")
    else: self.logger.warning("Skipping proactive memory search: No suitable
    search tool available.")

    # --- 2. Fetch Core Context via Tool ---
    if self._find_tool_server(TOOL_GET_CONTEXT):
        try:
            core_context_result = await
            self._execute_tool_call_internal(TOOL_GET_CONTEXT,
            {"workflow_id": current_workflow_id, "recent_actions_limit":
            CONTEXT_RECENT_ACTIONS, "important_memories_limit":
            CONTEXT_IMPORTANT_MEMORIES, "key_thoughts_limit":
            CONTEXT_KEY_THOUGHTS}, record_action=False)
            if core_context_result.get("success"):
            base_context["core_context"] = core_context_result;
            base_context["core_context"].pop("success", None);
            base_context["core_context"].pop("processing_time", None);
            self.logger.info("Core context retrieved.")
            else: base_context["errors"].append(f"Core context retrieval
            failed: {core_context_result.get('error')}")
        except Exception as e: self.logger.warning(f"Core context retrieval
        exception: {e}"); base_context["errors"].append(f"Core context
        exception: {e}")
    else: self.logger.warning(f"Skipping core context retrieval: Tool
    '{TOOL_GET_CONTEXT}' unavailable.")

    # --- 3. Fetch Relevant Procedural Memories (Using Hybrid Search -
    Tier 2) ---
    search_tool_proc = TOOL_HYBRID_SEARCH if
    self._find_tool_server(TOOL_HYBRID_SEARCH) else TOOL_SEMANTIC_SEARCH
    if self._find_tool_server(search_tool_proc):
         proc_query = f"How to accomplish: {active_plan_step_desc}"
         search_args = {"workflow_id": current_workflow_id, "query":
         proc_query, "limit": CONTEXT_PROCEDURAL_MEMORIES, "memory_level":
         MemoryLevel.PROCEDURAL.value, "include_content": False}
```

```
        if search_tool_proc == TOOL_HYBRID_SEARCH:
        search_args.update({"semantic_weight": 0.6, "keyword_weight": 0.4})
        try:
            proc_result = await
            self._execute_tool_call_internal(search_tool_proc, search_args,
            record_action=False)
            if proc_result.get("success"):
                proc_mems = proc_result.get("memories", []); score_key =
                "hybrid_score" if search_tool_proc == TOOL_HYBRID_SEARCH
                else "similarity"
                base_context["relevant_procedures"] = [{"memory_id":
                m.get("memory_id"), "description": m.get("description"),
                "score": m.get(score_key)} for m in proc_mems]
                if base_context["relevant_procedures"]:
                self.logger.info(f"Retrieved
                {len(base_context['relevant_procedures'])} relevant
                procedures using {search_tool_proc.split(':')[-1]}.")
            else: base_context["errors"].append(f"Procedure search failed:
            {proc_result.get('error')}")
        except Exception as e: self.logger.warning(f"Procedure search
        exception: {e}"); base_context["errors"].append(f"Procedure search
        exception: {e}")
    else: self.logger.warning("Skipping procedure search: No suitable search
    tool available.")

    # --- 4. Context Compression (Check) ---
    try:
        estimated_tokens = await
        self._estimate_tokens_anthropic(base_context)
        if estimated_tokens > CONTEXT_MAX_TOKENS_COMPRESS_THRESHOLD:
            self.logger.warning(f"Context ({estimated_tokens} tokens)
            exceeds threshold {CONTEXT_MAX_TOKENS_COMPRESS_THRESHOLD}.
            Attempting compression.")
            if self._find_tool_server(TOOL_SUMMARIZE_TEXT):
                actions_text = json.dumps(base_context.get("core_context",
                {}).get("recent_actions", []), indent=2, default=str)
                if len(actions_text) > 500:
                    summary_result = await self._execute_tool_call_internal(
                    TOOL_SUMMARIZE_TEXT, {"text_to_summarize": actions_text,
                    "target_tokens": CONTEXT_COMPRESSION_TARGET_TOKENS,
                    "workflow_id": current_workflow_id, "record_summary":
                    False}, record_action=False)
                    if summary_result.get("success"):
                        base_context["compression_summary"] = f"Summary of
                        recent actions: {summary_result.get('summary',
                        'Summary failed.')[:150]}..."
                        if base_context.get("core_context"):
                        base_context["core_context"].pop("recent_actions",
                        None)
```

```
                        self.logger.info(f"Compressed recent actions. New
                        context size: {await
                        self._estimate_tokens_anthropic(base_context)} est.
                        tokens")
                    else: base_context["errors"].append(f"Context
                    compression failed: {summary_result.get('error')}")
            else: self.logger.warning(f"Cannot compress context: Tool
            '{TOOL_SUMMARIZE_TEXT}' unavailable.")
    except Exception as e: self.logger.error(f"Error during context
    compression check: {e}", exc_info=False);
    base_context["errors"].append(f"Compression exception: {e}")

    # --- 5. Contextual Link Traversal ---
    base_context["contextual_links"] = None
    get_linked_memories_tool = TOOL_GET_LINKED_MEMORIES
    if self._find_tool_server(get_linked_memories_tool):
        mem_id_to_traverse = None
        # Prioritize focus memory from working memory if available
        wm_list = base_context.get("current_working_memory", [])
        if wm_list: mem_id_to_traverse = wm_list[0].get("memory_id") #
        Simple: pick first WM item
        # Fallback to important memories
        if not mem_id_to_traverse:
            important_mem_list = base_context.get("core_context",
            {}).get("important_memories", [])
            if important_mem_list and isinstance(important_mem_list, list)
            and len(important_mem_list) > 0:
                first_mem = important_mem_list[0]
                if isinstance(first_mem, dict): mem_id_to_traverse =
                first_mem.get("memory_id")

        if mem_id_to_traverse:
            self.logger.debug(f"Attempting link traversal from relevant
            memory: {mem_id_to_traverse[:8]}...")
            try:
                links_result_content = await
                self._execute_tool_call_internal(get_linked_memories_tool,
                {"memory_id": mem_id_to_traverse, "direction": "both",
                "limit": 3}, record_action=False)
                if links_result_content.get("success"):
                    links_data = links_result_content.get("links", {});
                    outgoing_links = links_data.get("outgoing", []);
                    incoming_links = links_data.get("incoming", [])
                    link_summary = {"source_memory_id": mem_id_to_traverse,
                    "outgoing_count": len(outgoing_links), "incoming_count":
                    len(incoming_links), "top_links_summary": []}
```

```
                        for link in outgoing_links[:2]:
                        link_summary["top_links_summary"].append(f"OUT:
                        {link.get('link_type', 'related')} ->
                        {link.get('target_type','Mem')}
                        '{str(link.get('target_description','?'))[:30]}...' (ID:
                        {str(link.get('target_memory_id','?'))[:6]}...)")
                        for link in incoming_links[:2]:
                        link_summary["top_links_summary"].append(f"IN:
                        {link.get('link_type', 'related')} <-
                        {link.get('source_type','Mem')}
                        '{str(link.get('source_description','?'))[:30]}...' (ID:
                        {str(link.get('source_memory_id','?'))[:6]}...)")
                        base_context["contextual_links"] = link_summary
                        self.logger.info(f"Retrieved {len(outgoing_links)}
                        outgoing, {len(incoming_links)} incoming links for
                        memory {mem_id_to_traverse[:8]}...")
                    else: err_msg = f"Link retrieval tool failed:
                    {links_result_content.get('error', 'Unknown')}";
                    base_context["errors"].append(err_msg);
                    self.logger.warning(err_msg)
                except Exception as e: err_msg = f"Link retrieval exception:
                {e}"; self.logger.warning(err_msg, exc_info=False);
                base_context["errors"].append(err_msg)
            else: self.logger.debug("No relevant memory found (working or
            important) to perform link traversal from.")
        else: self.logger.debug(f"Skipping link traversal: Tool
        '{get_linked_memories_tool}' unavailable.")

        base_context["status"] = "Ready" if not base_context["errors"] else
        "Ready with Errors"
        return base_context

    # --- Prompt Construction (Updated for Tier 3 Tools & Context) ---
    def _construct_agent_prompt(self, goal: str, context: Dict[str, Any]) ->
    List[Dict[str, Any]]:
        """Constructs the prompt for the LLM, including Tier 1, 2 & 3 tools
        and instructions."""
        system_prompt = f"""You are '{AGENT_NAME}', an AI agent orchestrator
        using a Unified Memory System. Achieve the Overall Goal by strategically
        using the provided memory tools.

Overall Goal: {goal}

Available Unified Memory Tools (Use ONLY these):
"""
        # Add tool descriptions to prompt
        if not self.tool_schemas: system_prompt += "- CRITICAL WARNING: No tools
        loaded.\n"
        else:
            for schema in self.tool_schemas:
```

```
                sanitized = schema['name']; original = self.mcp_client.server_
                manager.sanitized_to_original.get(sanitized,
                'Unknown')
                # Highlight Tier 1, 2 & 3 tools
                is_new_or_essential = original in [
                    TOOL_ADD_ACTION_DEPENDENCY, TOOL_GET_ACTION_DEPENDENCIES,
                    TOOL_RECORD_ARTIFACT, TOOL_GET_ARTIFACTS,
                    TOOL_GET_ARTIFACT_BY_ID,
                    TOOL_CREATE_LINK, TOOL_RECORD_ACTION_START,
                    TOOL_RECORD_ACTION_COMPLETION,
                    TOOL_RECORD_THOUGHT, TOOL_GET_ACTION_DETAILS,
                    TOOL_HYBRID_SEARCH, TOOL_STORE_MEMORY, TOOL_UPDATE_MEMORY,
                    TOOL_GET_WORKING_MEMORY,
                    TOOL_CREATE_THOUGHT_CHAIN, TOOL_GET_THOUGHT_CHAIN, # Tier
                    3
                    TOOL_COMPUTE_STATS, TOOL_DELETE_EXPIRED_MEMORIES # Tier 3
                ]
                prefix = "**" if is_new_or_essential else ""
                system_prompt += f"\n- {prefix}Name: `{sanitized}`
                (Represents: `{original}`){prefix}\n"
                system_prompt += f"  Desc: {schema.get('description',
                'N/A')}\n"; system_prompt += f"  Schema:
                {json.dumps(schema['input_schema'])}\n"

        # Add Tier 1, 2 & 3 Instructions
        system_prompt += """
Your Process:
1.  Context Analysis: Deeply analyze 'Current Context'. Note workflow status,
errors (`last_error_details`), recent actions, memories (`core_context`,
`proactive_memories`), thoughts, `current_plan`, `relevant_procedures`,
`current_working_memory` (most active memories), `current_thought_chain_id`, and
`meta_feedback`. Pay attention to memory `importance`/`confidence`.
2.  Error Handling: If `last_error_details` exists, **FIRST** reason about the
error and propose a recovery strategy in your Reasoning & Planning step. Check
if it was a dependency failure.
3.  Reasoning & Planning:
    a. State step-by-step reasoning towards the Goal/Sub-goal, integrating
    context and feedback. Consider `current_working_memory` for immediate
    context. Record thoughts using `record_thought` and specify the
    `thought_chain_id` if different from `current_thought_chain_id`.
    b. Evaluate `current_plan`. Is it valid? Does it address errors? Are
    dependencies (`depends_on`) likely met?
    c. **Action Dependencies:** If planning Step B requires output from Step A
    (action ID 'a123'), include `"depends_on": ["a123"]` in Step B's plan
    object.
    d. **Artifact Tracking:** If planning to use a tool that creates a
    file/data, plan a subsequent step to call `record_artifact`. If needing a
    previously created artifact, plan to use `get_artifacts` or
    `get_artifact_by_id` first.
```

```
    e. **Direct Memory Management:** If you synthesize a critical new fact,
    insight, or piece of knowledge, plan to use `store_memory` to explicitly
    save it. If you find strong evidence contradicting a stored memory, plan to
    use `update_memory` to correct it. Provide clear `content`, `memory_type`,
    `importance`, and `confidence`.
    f. **Custom Thought Chains:** If tackling a distinct sub-problem or
    exploring a complex tangent, consider creating a new reasoning thread using
    `create_thought_chain`. Provide a clear `title`. Subsequent related thoughts
    should specify the new `thought_chain_id`. The loop will automatically track
    the `current_thought_chain_id`.
    g. **Linking:** Identify potential memory relationships (causal, supportive,
    contradictory). Plan to use `create_memory_link` with specific `link_type`s.
    h. **Search:** Prefer `hybrid_search_memories` for mixed queries. Use
    `search_semantic_memories` for pure conceptual similarity.
    i. Propose an **Updated Plan** (1-3 structured `PlanStep` JSON objects).
    Explain reasoning for changes. Use `record_thought(thought_type='plan')` for
    complex planning.
4.  Action Decision: Choose **ONE** action based on the *first planned step* in
your Updated Plan:
    *   Call Memory Tool: Select the most precise `unified_memory:*` tool (or
    other available tool). Provide args per schema. **Mandatory:** Call
    `create_workflow` if context shows 'No Active Workflow'.
    *   Record Thought: Use `record_thought` for logging reasoning, questions,
    etc. Specify `thought_chain_id` if not the default/current one.
    *   Signal Completion: If Overall Goal is MET, respond ONLY with "Goal
    Achieved:" and summary.
5.  Output Format: Respond **ONLY** with the valid JSON for the chosen tool call
OR "Goal Achieved:" text. Include the updated plan JSON within your reasoning
text using the format `Updated Plan:\n```json\n[...plan steps...]\n````.

Key Considerations:
*   Use memory confidence. Update memories via `update_memory` if needed.
*   Store important learned info using `store_memory`.
*   Use `current_working_memory` for immediate relevance.
*   Dependencies: Ensure `depends_on` actions are likely complete. Use
`get_action_details`.
*   Artifacts: Track outputs (`record_artifact`), retrieve inputs
(`get_artifacts`/`get_artifact_by_id`).
*   Thought Chains: Use `create_thought_chain` for complex sub-problems. Record
subsequent thoughts using the correct `thought_chain_id`.
*   Linking: Use specific `link_type`s.
"""
        # Prepare context string
        context_str = json.dumps(context, indent=2, default=str,
        ensure_ascii=False); max_context_len = 25000
        if len(context_str) > max_context_len: context_str =
        context_str[:max_context_len] + "\n... (Context Truncated)\n}";
        self.logger.warning("Truncated context string sent to LLM.")

        user_prompt = f"Current Context:\n```json\n{context_str}\n```\n\n"
```

```
        user_prompt += f"My Current Plan
        (Structured):\n```json\n{json.dumps([s.model_dump() for s in
        self.state.current_plan], indent=2)}\n```\n\n"
        user_prompt += f"Last Action
        Summary:\n{self.state.last_action_summary}\n\n"
        if self.state.last_error_details: user_prompt += f"**CRITICAL: Address
        Last Error:**\n```json\n{json.dumps(self.state.last_error_details,
        indent=2)}\n```\n\n"
        if self.state.last_meta_feedback: user_prompt += f"**Meta-Cognitive
        Feedback:**\n{self.state.last_meta_feedback}\n\n"
        user_prompt += f"Overall Goal: {goal}\n\n"
        user_prompt += "**Instruction:** Analyze context & errors. Reason
        step-by-step. Update plan (output structured JSON plan steps in
        reasoning text). Decide ONE action based on the *first* planned step
        (Tool JSON or 'Goal Achieved:'). Focus on dependencies, artifacts,
        explicit memory storage/updates, custom thought chains, linking, and
        using working memory context."
        return [{"role": "user", "content": system_prompt + "\n---\n" +
        user_prompt}]

    async def _call_agent_llm(self, goal: str, context: Dict[str, Any]) ->
    Dict[str, Any]:
        """Calls Claude 3.7 Sonnet, includes structured plan parsing."""
        self.logger.info("Calling Agent LLM (Claude 3.7 Sonnet) for
        decision/plan...", emoji_key="robot_face")
        if not self.anthropic_client: return {"decision": "error", "message":
        "Anthropic client unavailable."}
        messages = self._construct_agent_prompt(goal, context)
        api_tools = self.tool_schemas

        try:
            response: Message = await self.anthropic_client.messages.create(
                model="claude-3-5-sonnet-20240620", max_tokens=4000,
                messages=messages, tools=api_tools, tool_choice={"type":
                "auto"}, temperature=0.4
            )
            self.logger.debug(f"LLM Raw Response Stop Reason:
            {response.stop_reason}")

            decision = {"decision": "error", "message": "LLM provided no
            actionable output."}
            text_parts = []
            tool_call = None
            updated_plan_steps = None

            for block in response.content:
                if block.type == "text": text_parts.append(block.text)
                elif block.type == "tool_use": tool_call = block

            full_text = "".join(text_parts).strip()
```

```
        # Parse Updated Plan from Text
        plan_match = re.search(r"Updated
        Plan:\s*```json\s*([\s\S]+?)\s*```", full_text, re.IGNORECASE)
        if plan_match:
            plan_json_str = plan_match.group(1).strip()
            try:
                plan_data = json.loads(plan_json_str)
                if isinstance(plan_data, list):
                    validated_plan = [PlanStep(**step_data) for step_data in
                    plan_data]
                    updated_plan_steps = validated_plan
                    self.logger.info(f"LLM proposed updated plan with
                    {len(updated_plan_steps)} steps.")
                else: self.logger.warning("LLM plan update was not a list.")
            except (json.JSONDecodeError, ValidationError, TypeError) as e:
                self.logger.warning(f"Failed to parse structured plan from
                LLM response: {e}")
        else: self.logger.debug("No structured 'Updated Plan:' block found
        in LLM text.")

        # Determine Final Decision
        if tool_call:
            tool_name_sanitized = tool_call.name; tool_input =
            tool_call.input or {}
            original_tool_name = self.mcp_client.server_manager.sanitized_t
            o_original.get(tool_name_sanitized,
            tool_name_sanitized)
            self.logger.info(f"LLM chose tool: {original_tool_name}",
            emoji_key="hammer_and_wrench")
            decision = {"decision": "call_tool", "tool_name":
            original_tool_name, "arguments": tool_input}
        elif full_text.startswith("Goal Achieved:"):
            decision = {"decision": "complete", "summary":
            full_text.replace("Goal Achieved:", "").strip()}
        elif full_text: # No tool, no completion -> treat as
        reasoning/thought
             decision = {"decision": "thought_process", "content":
             full_text}
             self.logger.info("LLM provided text reasoning/thought.")
        # else: decision remains default error

        # Attach parsed plan to the decision object
        if updated_plan_steps: decision["updated_plan_steps"] =
        updated_plan_steps

        self.logger.debug(f"Agent Decision Parsed: {decision}")
        return decision

    except anthropic.APIConnectionError as e: msg = f"API Connection Error:
    {e}"; self.logger.error(msg, exc_info=True)
```

```
    except anthropic.RateLimitError: msg = "Rate limit exceeded.";
    self.logger.error(msg, exc_info=True); await
    asyncio.sleep(random.uniform(5, 10))
    except anthropic.APIStatusError as e: msg = f"API Error
    {e.status_code}: {e.message}"; self.logger.error(f"Anthropic API status
    error: {e.status_code} - {e.response}", exc_info=True)
    except Exception as e: msg = f"Unexpected LLM interaction error: {e}";
    self.logger.error(msg, exc_info=True)
    return {"decision": "error", "message": msg}

async def _run_auto_linking(self, memory_id: str):
    """Background task to automatically link a new memory using richer
    link types."""
    try:
        if not memory_id or not self.state.workflow_id: return
        await asyncio.sleep(random.uniform(*AUTO_LINKING_DELAY_SECS))
        self.logger.debug(f"Attempting auto-linking for memory
        {memory_id[:8]}...")

        source_mem_details_result = await
        self._execute_tool_call_internal(TOOL_GET_MEMORY_BY_ID,
        {"memory_id": memory_id, "include_links": False},
        record_action=False)
        if not source_mem_details_result.get("success"):
        self.logger.warning(f"Auto-linking failed: couldn't retrieve source
        memory {memory_id}"); return
        source_mem = source_mem_details_result

        query_text = source_mem.get("description", "") or
        source_mem.get("content", "")[:200]
        if not query_text: return

        search_tool = TOOL_SEMANTIC_SEARCH
        if not self._find_tool_server(search_tool):
        self.logger.warning(f"Skipping auto-linking: Tool {search_tool}
        unavailable."); return

        similar_results = await
        self._execute_tool_call_internal(search_tool, {"workflow_id":
        self.state.workflow_id, "query": query_text, "limit":
        self.auto_linking_max_links + 1, "threshold":
        self.auto_linking_threshold }, record_action=False)
        if not similar_results.get("success"): return

        link_count = 0
        for similar_mem_summary in similar_results.get("memories", []):
            target_id = similar_mem_summary.get("memory_id")
            if not target_id or target_id == memory_id: continue
```

```
            target_mem_details_result = await
            self._execute_tool_call_internal(TOOL_GET_MEMORY_BY_ID,
            {"memory_id": target_id, "include_links": False},
            record_action=False)
            if not target_mem_details_result.get("success"): continue
            target_mem = target_mem_details_result

            inferred_link_type = LinkType.RELATED.value
            source_type = source_mem.get("memory_type"); target_type =
            target_mem.get("memory_type")
            if source_type == MemoryType.INSIGHT.value and target_type ==
            MemoryType.FACT.value: inferred_link_type =
            LinkType.SUPPORTS.value
            elif source_type == MemoryType.FACT.value and target_type ==
            MemoryType.INSIGHT.value: inferred_link_type =
            LinkType.SUPPORTS.value
            elif source_type == MemoryType.EVIDENCE.value and target_type
            == MemoryType.HYPOTHESIS.value: inferred_link_type =
            LinkType.SUPPORTS.value
            elif source_type == MemoryType.HYPOTHESIS.value and target_type
            == MemoryType.EVIDENCE.value: inferred_link_type =
            LinkType.SUPPORTS.value

            if not self._find_tool_server(TOOL_CREATE_LINK):
            self.logger.warning(f"Cannot create link: Tool
            {TOOL_CREATE_LINK} unavailable."); break

            await self._execute_tool_call_internal(TOOL_CREATE_LINK,
            {"source_memory_id": memory_id, "target_memory_id": target_id,
            "link_type": inferred_link_type, "strength":
            similar_mem_summary.get("similarity", 0.7), "description":
            f"Auto-link ({inferred_link_type}) based on similarity"},
            record_action=False)
            link_count += 1; self.logger.debug(f"Auto-linked memory
            {memory_id[:8]} to {target_id[:8]} ({inferred_link_type},
            similarity: {similar_mem_summary.get('similarity', 0):.2f})")
            if link_count >= self.auto_linking_max_links: break
    except Exception as e: self.logger.warning(f"Error in auto-linking task
    for {memory_id}: {e}", exc_info=False)

async def _check_prerequisites(self, dependency_ids: List[str]) ->
Tuple[bool, str]:
    """Check if all prerequisite actions are completed using
    get_action_details."""
    if not dependency_ids: return True, "No dependencies"
    self.logger.debug(f"Checking prerequisites: {dependency_ids}")
    if not self._find_tool_server(TOOL_GET_ACTION_DETAILS): return False,
    f"Cannot check: Tool {TOOL_GET_ACTION_DETAILS} unavailable."
    try:
```

```python
            dep_details_result = await
            self._execute_tool_call_internal(TOOL_GET_ACTION_DETAILS,
            {"action_ids": dependency_ids, "include_dependencies": False},
            record_action=False)
            if not dep_details_result.get("success"): return False, f"Failed to
            check dependencies: {dep_details_result.get('error', 'Unknown
            error')}"
            actions = dep_details_result.get("actions", []); found_ids =
            {a.get("action_id") for a in actions}; missing =
            list(set(dependency_ids) - found_ids)
            if missing: return False, f"Dependency actions not found:
            {missing}"
            incomplete = [a.get("action_id") for a in actions if
            a.get("status") != ActionStatus.COMPLETED.value]
            if incomplete: incomplete_titles = [f"'{a.get('title',
            a.get('action_id')[:8])}' ({a.get('status')})" for a in actions if
            a.get('action_id') in incomplete]; return False, f"Dependencies not
            completed: {', '.join(incomplete_titles)}"
            return True, "All dependencies completed"
        except Exception as e: self.logger.error(f"Error checking
        prerequisites: {e}", exc_info=False); return False, f"Error checking
        prerequisites: {str(e)}"

    async def _execute_tool_call_internal(
        self, tool_name: str, arguments: Dict[str, Any],
        record_action: bool = True,
        planned_dependencies: Optional[List[str]] = None
        ) -> Dict[str, Any]:
        """Handles server lookup, dependency checks, execution, results,
        optional action recording, dependency recording, and triggers."""
        action_id = None
        tool_result_content = {"success": False, "error": "Execution error."}
        start_time = time.time()

        # 1. Find Server
        target_server = self._find_tool_server(tool_name)
        if not target_server: err_msg = f"Tool/server unavailable:
        {tool_name}"; self.logger.error(err_msg); self.state.last_error_details
        = {"tool": tool_name, "error": err_msg}; return {"success": False,
        "error": err_msg, "status_code": 503}

        # 2. Dependency Check
        if planned_dependencies:
            met, reason = await self._check_prerequisites(planned_dependencies)
            if not met: err_msg = f"Prerequisites not met for {tool_name}:
            {reason}"; self.logger.warning(err_msg);
            self.state.last_error_details = {"tool": tool_name, "error":
            err_msg, "type": "dependency_failure", "dependencies":
            planned_dependencies}; self.state.needs_replan = True; return
            {"success": False, "error": err_msg, "status_code": 412}
```

```python
    self.logger.info(f"Prerequisites {planned_dependencies} met for
    {tool_name}.")

# 3. Record Action Start (Optional)
# Determine if this tool call represents a significant agent action
should_record_start = record_action and tool_name not in [
    TOOL_RECORD_ACTION_START, TOOL_RECORD_ACTION_COMPLETION, #
    Meta-actions
    TOOL_GET_CONTEXT, TOOL_GET_WORKING_MEMORY, # Context gathering
    TOOL_SEMANTIC_SEARCH, TOOL_HYBRID_SEARCH, TOOL_QUERY_MEMORIES, #
    Searches
    TOOL_GET_MEMORY_BY_ID, TOOL_GET_LINKED_MEMORIES, # Retrievals
    TOOL_GET_ACTION_DETAILS, TOOL_GET_ARTIFACTS,
    TOOL_GET_ARTIFACT_BY_ID, # Retrievals
    TOOL_ADD_ACTION_DEPENDENCY, TOOL_CREATE_LINK, # Internal
    linking/dep mgmt
    TOOL_LIST_WORKFLOWS, TOOL_COMPUTE_STATS, TOOL_SUMMARIZE_TEXT, #
    Meta/utility
    TOOL_OPTIMIZE_WM, TOOL_AUTO_FOCUS, TOOL_PROMOTE_MEM, #
    Periodic/internal cognitive
    TOOL_REFLECTION, TOOL_CONSOLIDATION, TOOL_DELETE_EXPIRED_MEMORIES #
    Periodic/meta
]
if should_record_start:
    action_id = await self._record_action_start_internal(tool_name,
    arguments, target_server)
    # 3.5 Record Dependencies AFTER starting the action
    if action_id and planned_dependencies:
        await self._record_action_dependencies_internal(action_id,
        planned_dependencies)

# 4. Execute Tool
try:
    current_wf_id = self.state.workflow_stack[-1] if
    self.state.workflow_stack else self.state.workflow_id
    # Inject workflow_id automatically
    if 'workflow_id' not in arguments and current_wf_id and tool_name
    not in [TOOL_CREATE_WORKFLOW, TOOL_LIST_WORKFLOWS,
    'core:list_servers', 'core:get_tool_schema']:
    arguments['workflow_id'] = current_wf_id
    # Inject context_id if needed
    if 'context_id' not in arguments and self.state.context_id and
    tool_name in [TOOL_GET_WORKING_MEMORY, TOOL_OPTIMIZE_WM,
    TOOL_AUTO_FOCUS]: arguments['context_id'] = self.state.context_id
    # Inject current thought chain ID if needed and available
    if 'thought_chain_id' not in arguments and
    self.state.current_thought_chain_id and tool_name ==
    TOOL_RECORD_THOUGHT: arguments['thought_chain_id'] =
    self.state.current_thought_chain_id

    clean_args = {k: v for k, v in arguments.items() if v is not None}
```

```
call_tool_result = await self.mcp_client.execute_tool(target_server,
tool_name, clean_args)
latency_ms = (time.time() - start_time) * 1000
self.state.tool_usage_stats[tool_name]["latency_ms_total"] +=
latency_ms

# Process result
if isinstance(call_tool_result, dict):
    is_error = call_tool_result.get("isError", False); content =
    call_tool_result.get("content")
    if is_error or (content is None and "success" not in
    call_tool_result): error_msg = str(content or
    call_tool_result.get("error", "Unknown tool error."));
    tool_result_content = {"success": False, "error": error_msg}
    elif isinstance(content, dict) and "success" in content:
    tool_result_content = content
    else: tool_result_content = {"success": True, "data": content}
else: tool_result_content = {"success": False, "error":
f"Unexpected result type: {type(call_tool_result)}"}

log_msg = f"Tool {tool_name} executed. Success:
{tool_result_content.get('success')} ({latency_ms:.0f}ms)"
self.logger.info(log_msg, emoji_key="checkered_flag" if
tool_result_content.get('success') else "warning")
self.state.last_action_summary = log_msg
if not tool_result_content.get('success'):
     err_detail = str(tool_result_content.get('error',
     'Unknown'))[:150]; self.state.last_action_summary += f" Error:
     {err_detail}"; self.state.last_error_details = {"tool":
     tool_name, "args": arguments, "error": err_detail, "result":
     tool_result_content};
     self.state.tool_usage_stats[tool_name]["failure"] += 1
else:
     self.state.last_error_details = None;
     self.state.consecutive_error_count = 0;
     self.state.tool_usage_stats[tool_name]["success"] += 1
     # Trigger Post-Success Actions
     if tool_name in [TOOL_STORE_MEMORY, TOOL_UPDATE_MEMORY] and
     tool_result_content.get("memory_id"): self._start_background_t⌟
     ask(self._run_auto_linking(tool_result_content["memory_id"]))
     if tool_name == TOOL_RECORD_ARTIFACT and
     tool_result_content.get("linked_memory_id"):
     self._start_background_task(self._run_auto_linking(tool_result⌟
     _content["linked_memory_id"]))
     if tool_name in [TOOL_GET_MEMORY_BY_ID, TOOL_QUERY_MEMORIES,
     TOOL_HYBRID_SEARCH]:
        mem_ids_to_check = []
        if tool_name == TOOL_GET_MEMORY_BY_ID: mem_ids_to_check =
        [arguments.get("memory_id")]
```

```
                else: memories = tool_result_content.get("memories", []) if
                isinstance(tool_result_content, dict) else [];
                mem_ids_to_check = [m.get("memory_id") for m in
                memories[:3]]
                for mem_id in filter(None, mem_ids_to_check): self._start_
                background_task(self._check_and_trigger_promotion(mem_id))
             # Update current thought chain ID if a new one was created
             (Tier 3)
             if tool_name == TOOL_CREATE_THOUGHT_CHAIN and
             tool_result_content.get("success"):
                  new_chain_id = tool_result_content.get("thought_chain_id")
                  if new_chain_id:
                       self.state.current_thought_chain_id = new_chain_id
                       self.logger.info(f"Switched current thought chain to
                       newly created: {new_chain_id}")


     except (ToolError, ToolInputError) as e:
          err_str = str(e); status_code = getattr(e, 'status_code', None);
          self.logger.error(f"Tool Error executing {tool_name}: {e}",
          exc_info=False); tool_result_content = {"success": False, "error":
          err_str, "status_code": status_code};
          self.state.last_action_summary = f"Tool {tool_name} Error:
          {err_str[:100]}"; self.state.last_error_details = {"tool":
          tool_name, "args": arguments, "error": err_str, "type":
          type(e).__name__};
          self.state.tool_usage_stats[tool_name]["failure"] += 1
          if status_code == 412: self.state.last_error_details["type"] =
          "dependency_failure"; self.state.needs_replan = True
     except Exception as e: err_str = str(e); self.logger.error(f"Unexpected
     Error executing {tool_name}: {e}", exc_info=True); tool_result_content
     = {"success": False, "error": f"Unexpected error: {err_str}"};
     self.state.last_action_summary = f"Execution failed: Unexpected error.";
     self.state.last_error_details = {"tool": tool_name, "args": arguments,
     "error": err_str, "type": "Unexpected"};
     self.state.tool_usage_stats[tool_name]["failure"] += 1

     # 5. Record Action Completion (Optional)
     if should_record_start and action_id: # Only record completion if start
     was recorded
          await self._record_action_completion_internal(action_id,
          tool_result_content)

     # 6. Handle Workflow Side Effects
     await self._handle_workflow_side_effects(tool_name, arguments,
     tool_result_content)

     return tool_result_content

async def _record_action_dependencies_internal(self, source_action_id: str,
target_action_ids: List[str]):
```

```
        """Records dependencies using the add_action_dependency tool."""
        if not source_action_id or not target_action_ids: return
        self.logger.debug(f"Recording dependencies for action
        {source_action_id[:8]}: depends on {target_action_ids}")
        dep_tool_name = TOOL_ADD_ACTION_DEPENDENCY
        if not self._find_tool_server(dep_tool_name): self.logger.error(f"Cannot
        record dependency: Tool '{dep_tool_name}' unavailable."); return

        dep_tasks = []; unique_target_ids = set(target_action_ids)
        for target_id in unique_target_ids:
            if target_id == source_action_id: self.logger.warning(f"Skipping
            self-dependency for action {source_action_id}"); continue
            args = {"source_action_id": source_action_id, "target_action_id":
            target_id, "dependency_type": "requires"}
            task =
            asyncio.create_task(self._execute_tool_call_internal(dep_tool_name,
            args, record_action=False, planned_dependencies=None))
            dep_tasks.append(task)
        results = await asyncio.gather(*dep_tasks, return_exceptions=True)
        valid_target_ids = [tid for tid in unique_target_ids if tid !=
        source_action_id]
        for i, res in enumerate(results):
            if i >= len(valid_target_ids): break
            target_id = valid_target_ids[i]
            if isinstance(res, Exception): self.logger.error(f"Error recording
            dependency {source_action_id[:8]} -> {target_id[:8]}: {res}",
            exc_info=False)
            elif isinstance(res, dict) and not res.get("success"):
            self.logger.warning(f"Failed recording dependency
            {source_action_id[:8]} -> {target_id[:8]}: {res.get('error')}")

    async def _record_action_start_internal(self, primary_tool_name: str,
    primary_tool_args: Dict[str, Any], primary_target_server: str) ->
    Optional[str]:
         """Internal helper to record action start."""
         action_id = None; start_title = f"Exec:
         {primary_tool_name.split(':')[-1]}"; start_reasoning = f"Agent
         initiated tool: {primary_tool_name}"; current_wf_id =
         self.state.workflow_stack[-1] if self.state.workflow_stack else
         self.state.workflow_id
         if not current_wf_id: self.logger.warning("Cannot record start: No
         active workflow"); return None
         start_tool_name = TOOL_RECORD_ACTION_START
         if not self._find_tool_server(start_tool_name):
         self.logger.error(f"Cannot record start: Tool '{start_tool_name}'
         unavailable."); return None
         try:
             safe_tool_args = json.loads(json.dumps(primary_tool_args,
             default=str))
```

```
            start_args = {"workflow_id": current_wf_id, "action_type":
            ActionType.TOOL_USE.value, "title": start_title, "reasoning":
            start_reasoning, "tool_name": primary_tool_name, "tool_args":
            safe_tool_args}
            start_result_content = await
            self._execute_tool_call_internal(start_tool_name, start_args,
            record_action=False)
            if start_result_content.get("success"):
                action_id = start_result_content.get("action_id")
                if action_id: self.logger.debug(f"Action {action_id} started
                for {primary_tool_name}.")
                else: self.logger.warning(f"Record action start succeeded but
                returned no action ID.")
            else: error_msg = start_result_content.get('error', 'Unknown');
            self.logger.warning(f"Failed recording start for
            {primary_tool_name}: {error_msg}")
        except Exception as e: self.logger.error(f"Exception recording start
        for {primary_tool_name}: {e}", exc_info=True)
        return action_id

    async def _record_action_completion_internal(self, action_id: str,
    tool_result_content: Dict):
        """Internal helper to record action completion."""
        status = ActionStatus.COMPLETED.value if
        tool_result_content.get("success") else ActionStatus.FAILED.value
        comp_tool_name = TOOL_RECORD_ACTION_COMPLETION
        if not self._find_tool_server(comp_tool_name):
        self.logger.error(f"Cannot record completion: Tool '{comp_tool_name}'
        unavailable."); return
        try:
            safe_result = json.loads(json.dumps(tool_result_content,
            default=str))
            completion_args = {"action_id": action_id, "status": status,
            "tool_result": safe_result}
            comp_result_content = await
            self._execute_tool_call_internal(comp_tool_name, completion_args,
            record_action=False)
            if not comp_result_content.get("success"): error_msg =
            comp_result_content.get('error', 'Unknown');
            self.logger.warning(f"Failed recording completion for {action_id}:
            {error_msg}")
            else: self.logger.debug(f"Action {action_id} completion recorded
            ({status})")
        except Exception as e: self.logger.error(f"Error recording completion
        for {action_id}: {e}", exc_info=True)

    async def _handle_workflow_side_effects(self, tool_name: str, arguments:
    Dict, result_content: Dict):
        """Handles state changes after specific tool calls."""
        if tool_name == TOOL_CREATE_WORKFLOW and result_content.get("success"):
```

```
            new_wf_id = result_content.get("workflow_id"); parent_id =
            arguments.get("parent_workflow_id")
            if new_wf_id:
                self.state.workflow_id = new_wf_id; self.state.context_id =
                new_wf_id
                if parent_id: self.state.workflow_stack.append(new_wf_id)
                else: self.state.workflow_stack = [new_wf_id]
                # --- Set current_thought_chain_id for new workflow (Tier 3)
                ---
                self.state.current_thought_chain_id =
                result_content.get("primary_thought_chain_id")
                self.logger.info(f"Switched to {'sub-' if parent_id else
                'new'} workflow: {new_wf_id}. Current chain:
                {self.state.current_thought_chain_id}", emoji_key="label")
                self.state.current_plan = [PlanStep(description=f"Start new
                workflow: {result_content.get('title', 'Untitled')}. Goal:
                {result_content.get('goal', 'Not specified')}.")];
                self.state.consecutive_error_count = 0; self.state.needs_replan
                = False
        elif tool_name == TOOL_UPDATE_WORKFLOW_STATUS and
        result_content.get("success"):
            status = arguments.get("status"); wf_id =
            arguments.get("workflow_id")
            if status in [s.value for s in [WorkflowStatus.COMPLETED,
            WorkflowStatus.FAILED, WorkflowStatus.ABANDONED]] and
            self.state.workflow_stack and wf_id ==
            self.state.workflow_stack[-1]:
                 finished_wf = self.state.workflow_stack.pop()
                 if self.state.workflow_stack:
                      self.state.workflow_id = self.state.workflow_stack[-1];
                      self.state.context_id = self.state.workflow_id
                      # Fetch parent's primary thought chain ID
                      await self._set_default_thought_chain_id()
                      self.logger.info(f"Sub-workflow {finished_wf} finished.
                      Returning to parent {self.state.workflow_id}. Current
                      chain: {self.state.current_thought_chain_id}",
                      emoji_key="arrow_left")
                      self.state.needs_replan = True; self.state.current_plan =
                      [PlanStep(description=f"Returned from sub-workflow
                      {finished_wf} (status: {status}). Re-assess parent
                      goal.")]
                 else: self.state.workflow_id = None; self.state.context_id =
                 None; self.state.current_thought_chain_id = None;
                 self.logger.info(f"Root workflow {finished_wf} finished.");
                 self.state.goal_achieved_flag = True

    async def _update_plan(self, context: Dict[str, Any], last_decision:
    Dict[str, Any], last_tool_result_content: Optional[Dict[str, Any]] = None):
        """Updates the plan based on LLM proposal or heuristics."""
        self.logger.info("Updating agent plan...", emoji_key="clipboard")
        llm_proposed_plan = last_decision.get("updated_plan_steps")
```

```
    if llm_proposed_plan and isinstance(llm_proposed_plan, list):
        try:
            validated_plan = [PlanStep(**step) if isinstance(step, dict)
            else step for step in llm_proposed_plan]
            if validated_plan and all(isinstance(step, PlanStep) for step
            in validated_plan):
                self.state.current_plan = validated_plan;
                self.logger.info(f"Plan updated by LLM with
                {len(validated_plan)} steps. First step:
                '{validated_plan[0].description[:50]}...'");
                self.state.needs_replan = False
                if self.state.last_error_details:
                self.state.consecutive_error_count = 0
                if last_decision.get("decision") == "call_tool" and
                isinstance(last_tool_result_content, dict) and
                last_tool_result_content.get("success"):
                self.state.successful_actions_since_reflection += 1;
                self.state.successful_actions_since_consolidation += 1
                return
            else: self.logger.warning("LLM provided invalid or empty plan
            structure. Falling back to heuristic.")
        except (ValidationError, TypeError) as e:
        self.logger.warning(f"Failed to validate LLM plan: {e}. Falling
        back.")

    # --- Fallback to Heuristic Plan Update ---
    if not self.state.current_plan: self.logger.warning("Plan is empty,
    adding default re-evaluation step."); self.state.current_plan =
    [PlanStep(description="Fallback: Re-evaluate situation.")];
    self.state.needs_replan = True; return
    current_step = self.state.current_plan[0]

    if last_decision.get("decision") == "call_tool":
        tool_success = isinstance(last_tool_result_content, dict) and
        last_tool_result_content.get("success", False)
        if tool_success:
            current_step.status = "completed"; action_id_from_result = None
            if isinstance(last_tool_result_content, dict):
            action_id_from_result =
            last_tool_result_content.get('action_id') or
            (last_tool_result_content.get('data') or {}).get('action_id')
            summary_text = f"Success:
            {str(last_tool_result_content)[:100]}..."
            if action_id_from_result and last_decision.get("tool_name") ==
            TOOL_RECORD_ACTION_START : summary_text += f" (ActionID:
            {action_id_from_result[:8]})"
            current_step.result_summary = summary_text;
            self.state.current_plan.pop(0)
```

```
            if not self.state.current_plan:
            self.state.current_plan.append(PlanStep(description="Analyze
            successful tool output and plan next steps."))
            self.state.consecutive_error_count = 0; self.state.needs_replan
            = False; self.state.successful_actions_since_reflection += 1;
            self.state.successful_actions_since_consolidation += 1
        else:
            current_step.status = "failed"; error_msg =
            str(last_tool_result_content.get('error', 'Unknown
            failure'))[:150]; current_step.result_summary = f"Failure:
            {error_msg}"
            self.state.current_plan = [current_step] +
            self.state.current_plan[1:]
            if len(self.state.current_plan) < 2 or not
            self.state.current_plan[1].description.startswith("Analyze
            failure"): self.state.current_plan.insert(1,
            PlanStep(description=f"Analyze failure of step
            '{current_step.description[:30]}...' and replan."))
            self.state.consecutive_error_count += 1;
            self.state.needs_replan = True;
            self.state.successful_actions_since_reflection =
            self.state.current_reflection_threshold # Use dynamic
            threshold
    elif last_decision.get("decision") == "thought_process":
        current_step.status = "completed"; current_step.result_summary =
        f"Thought Recorded: {last_decision.get('content','')[:50]}..."
        self.state.current_plan.pop(0)
        if not self.state.current_plan:
        self.state.current_plan.append(PlanStep(description="Decide next
        action based on recorded thought."))
        self.state.consecutive_error_count = 0; self.state.needs_replan =
        False
    elif last_decision.get("decision") == "complete":
    self.state.current_plan = [PlanStep(description="Goal Achieved.
    Finalizing.", status="completed")]; self.state.consecutive_error_count =
    0; self.state.needs_replan = False
    else:
        current_step.status = "failed"; current_step.result_summary =
        f"Agent/Tool Error: {self.state.last_action_summary[:100]}..."
        self.state.current_plan = [current_step] +
        self.state.current_plan[1:]
        if len(self.state.current_plan) < 2 or not
        self.state.current_plan[1].description.startswith("Re-evaluate
        due"): self.state.current_plan.insert(1,
        PlanStep(description="Re-evaluate due to agent error or unclear
        decision."))
        self.state.consecutive_error_count += 1; self.state.needs_replan =
        True
    log_plan = f"Plan updated (Heuristic). Steps:
    {len(self.state.current_plan)}. Next:
    '{self.state.current_plan[0].description[:60]}...'"
```

```
        self.logger.info(log_plan)

    # --- Periodic Tasks (Enhanced for Tier 3) ---
    async def _run_periodic_tasks(self):
        """Runs meta-cognition and maintenance tasks, including adaptive
        adjustments."""
        if not self.state.workflow_id or not self.state.context_id or
        self._shutdown_event.is_set(): return

        tasks_to_run: List[Tuple[str, Dict]] = []; trigger_reason = []
        reflection_tool_available = self._find_tool_server(TOOL_REFLECTION) is
        not None
        consolidation_tool_available =
        self._find_tool_server(TOOL_CONSOLIDATION) is not None
        optimize_wm_tool_available = self._find_tool_server(TOOL_OPTIMIZE_WM) is
        not None
        auto_focus_tool_available = self._find_tool_server(TOOL_AUTO_FOCUS) is
        not None
        promote_mem_tool_available = self._find_tool_server(TOOL_PROMOTE_MEM) is
        not None
        stats_tool_available = self._find_tool_server(TOOL_COMPUTE_STATS) is not
        None
        maintenance_tool_available =
        self._find_tool_server(TOOL_DELETE_EXPIRED_MEMORIES) is not None

        # --- Tier 3: Stats Check & Adaptation ---
        self.state.loops_since_stats_adaptation += 1
        if self.state.loops_since_stats_adaptation >= STATS_ADAPTATION_INTERVAL:
            if stats_tool_available:
                trigger_reason.append("StatsInterval")
                try:
                    stats = await self._execute_tool_call_internal(
                        TOOL_COMPUTE_STATS, {"workflow_id":
                        self.state.workflow_id}, record_action=False
                    )
                    if stats.get("success"):
                        self._adapt_thresholds(stats)
                        # Trigger consolidation if episodic memories are high
                        episodic_count = stats.get("by_level",
                        {}).get(MemoryLevel.EPISODIC.value, 0)
                        # Example: Trigger if > 2x the consolidation
                        threshold, regardless of success count
                        if episodic_count >
                        (self.state.current_consolidation_threshold * 2) and
                        consolidation_tool_available:
                            if not any(task[0] == TOOL_CONSOLIDATION for task
                            in tasks_to_run): # Avoid duplicate scheduling
```

```
                        tasks_to_run.append((TOOL_CONSOLIDATION,
                        {"workflow_id": self.state.workflow_id,
                        "consolidation_type": "summary",
                        "query_filter": {"memory_level":
                        MemoryLevel.EPISODIC.value},
                        "max_source_memories":
                        self.consolidation_max_sources}))
                        trigger_reason.append(f"HighEpisodic({episodic
                        _count})")
                        self.state.successful_actions_since_consolidat
                        ion = 0 # Reset counter as we're consolidating
                        now
                else: self.logger.warning(f"Failed to compute stats for
                adaptation: {stats.get('error')}")
            except Exception as e: self.logger.error(f"Error during stats
            computation/adaptation: {e}", exc_info=False)
            self.state.loops_since_stats_adaptation = 0 # Reset interval
            counter
        else: self.logger.warning(f"Skipping stats/adaptation: Tool
        {TOOL_COMPUTE_STATS} not available")


    # --- Tier 3: Maintenance Check ---
    self.state.loops_since_maintenance += 1
    if self.state.loops_since_maintenance >= MAINTENANCE_INTERVAL:
        if maintenance_tool_available:
            tasks_to_run.append((TOOL_DELETE_EXPIRED_MEMORIES, {}));
            trigger_reason.append("MaintenanceInterval")
            self.state.loops_since_maintenance = 0 # Reset interval
            counter
        else: self.logger.warning(f"Skipping maintenance: Tool
        {TOOL_DELETE_EXPIRED_MEMORIES} not available")

    # --- Existing Triggers (Now use dynamic thresholds) ---
    # Reflection Trigger
    if self.state.needs_replan or
    self.state.successful_actions_since_reflection >=
    self.state.current_reflection_threshold:
        if reflection_tool_available:
            if not any(task[0] == TOOL_REFLECTION for task in
            tasks_to_run): # Avoid duplicates if scheduled by stats
```

```
reflection_type = self.reflection_type_sequence[self.state
.reflection_cycle_index %
len(self.reflection_type_sequence)];
tasks_to_run.append((TOOL_REFLECTION, {"workflow_id":
self.state.workflow_id, "reflection_type":
reflection_type})); trigger_reason.append(f"ReplanNeeded({
self.state.needs_replan})" if self.state.needs_replan else
f"SuccessCount({self.state.successful_actions_since_reflec
tion}>={self.state.current_reflection_threshold})");
self.state.successful_actions_since_reflection = 0;
self.state.reflection_cycle_index += 1
    else: self.logger.warning(f"Skipping reflection: Tool
    {TOOL_REFLECTION} not available")
# Consolidation Trigger
if self.state.successful_actions_since_consolidation >=
self.state.current_consolidation_threshold:
    if consolidation_tool_available:
        if not any(task[0] == TOOL_CONSOLIDATION for task in
        tasks_to_run): # Avoid duplicates if scheduled by stats
            tasks_to_run.append((TOOL_CONSOLIDATION, {"workflow_id":
            self.state.workflow_id, "consolidation_type": "summary",
            "query_filter": {"memory_level":
            MemoryLevel.EPISODIC.value}, "max_source_memories":
            self.consolidation_max_sources}));
            trigger_reason.append(f"ConsolidateThreshold({self.state.
            successful_actions_since_consolidation}>={self.state.curr
            ent_consolidation_threshold})");
            self.state.successful_actions_since_consolidation = 0
    else: self.logger.warning(f"Skipping consolidation: Tool
    {TOOL_CONSOLIDATION} not available")
# Optimization Trigger
self.state.loops_since_optimization += 1
if self.state.loops_since_optimization >= OPTIMIZATION_LOOP_INTERVAL: #
Use constant interval
    if optimize_wm_tool_available:
    tasks_to_run.append((TOOL_OPTIMIZE_WM, {"context_id":
    self.state.context_id})); trigger_reason.append("OptimizeInterval")
    else: self.logger.warning(f"Skipping optimization: Tool
    {TOOL_OPTIMIZE_WM} not available")
    if auto_focus_tool_available: tasks_to_run.append((TOOL_AUTO_FOCUS,
    {"context_id": self.state.context_id}));
    trigger_reason.append("FocusUpdate")
    else: self.logger.warning(f"Skipping auto-focus: Tool
    {TOOL_AUTO_FOCUS} not available")
    self.state.loops_since_optimization = 0
# Promotion Check Trigger
self.state.loops_since_promotion_check += 1
if self.state.loops_since_promotion_check >=
MEMORY_PROMOTION_LOOP_INTERVAL: # Use constant interval
```

```
            if promote_mem_tool_available:
            tasks_to_run.append(("CHECK_PROMOTIONS", {}));
            trigger_reason.append("PromotionInterval")
            else: self.logger.warning(f"Skipping promotion check: Tool
            {TOOL_PROMOTE_MEM} not available")
            self.state.loops_since_promotion_check = 0

        # Execute Scheduled Tasks
        if tasks_to_run:
            unique_reasons = sorted(set(trigger_reason)) # Deduplicate reasons
            for logging
            self.logger.info(f"Running {len(tasks_to_run)} periodic tasks
            (Triggers: {', '.join(unique_reasons)})...", emoji_key="brain")
            # Prioritize maintenance and stats if scheduled
            tasks_to_run.sort(key=lambda x: 0 if x[0] ==
            TOOL_DELETE_EXPIRED_MEMORIES else 1 if x[0] == TOOL_COMPUTE_STATS
            else 2)
            for tool_name, args in tasks_to_run:
                if self._shutdown_event.is_set(): break
                try:
                    if tool_name == "CHECK_PROMOTIONS": await
                    self._trigger_promotion_checks(); continue
                    self.logger.debug(f"Executing periodic task: {tool_name}
                    with args: {args}")
                    result_content = await
                    self._execute_tool_call_internal(tool_name, args,
                    record_action=False)
                    # Meta-Cognition Feedback Loop
                    if tool_name in [TOOL_REFLECTION, TOOL_CONSOLIDATION] and
                    result_content.get('success'):
                        content_key = "reflection_content" if tool_name ==
                        TOOL_REFLECTION else "consolidated_content"; feedback
                        = result_content.get(content_key, "") or
                        result_content.get("data", "")
                        if feedback: feedback_summary =
                        str(feedback).split('\n')[0][:150];
                        self.state.last_meta_feedback = f"Feedback from
                        {tool_name.split(':')[-1]}: {feedback_summary}...";
                        self.logger.info(self.state.last_meta_feedback);
                        self.state.needs_replan = True
                except Exception as e: self.logger.warning(f"Periodic task
                {tool_name} failed: {e}", exc_info=False)
                await asyncio.sleep(0.1) # Small delay

    # --- Tier 3: Adaptive Threshold Logic ---
    def _adapt_thresholds(self, stats: Dict[str, Any]):
        """Adjusts reflection and consolidation thresholds based on memory
        stats."""
        self.logger.debug(f"Adapting thresholds based on stats: {stats}")
        adjustment_factor = 0.1 # How much to adjust thresholds by each time
```

```
# Example Heuristic 1: High episodic count -> Lower consolidation
threshold (consolidate sooner)
episodic_count = stats.get("by_level",
{}).get(MemoryLevel.EPISODIC.value, 0)
target_episodic = BASE_CONSOLIDATION_THRESHOLD * 1.5 # Target range
if episodic_count > target_episodic * 1.5: # Significantly over target
    new_threshold = max(MIN_CONSOLIDATION_THRESHOLD,
    self.state.current_consolidation_threshold -
    math.ceil(self.state.current_consolidation_threshold *
    adjustment_factor))
    if new_threshold != self.state.current_consolidation_threshold:
        self.logger.info(f"High episodic count ({episodic_count}).
        Lowering consolidation threshold:
        {self.state.current_consolidation_threshold} ->
        {new_threshold}")
        self.state.current_consolidation_threshold = new_threshold
elif episodic_count < target_episodic * 0.75: # Well below target
     new_threshold = min(MAX_CONSOLIDATION_THRESHOLD,
     self.state.current_consolidation_threshold +
     math.ceil(self.state.current_consolidation_threshold *
     adjustment_factor))
     if new_threshold != self.state.current_consolidation_threshold:
         self.logger.info(f"Low episodic count ({episodic_count}).
         Raising consolidation threshold:
         {self.state.current_consolidation_threshold} ->
         {new_threshold}")
         self.state.current_consolidation_threshold = new_threshold


# Example Heuristic 2: High tool failure rate -> Lower reflection
threshold (reflect sooner)
total_calls = sum(sum(v.values()) for k, v in
self.state.tool_usage_stats.items() if isinstance(v, dict) and k !=
'latency_ms_total')
total_failures = sum(v.get("failure", 0) for v in
self.state.tool_usage_stats.values())
failure_rate = (total_failures / total_calls) if total_calls > 5 else
0.0 # Require minimum calls

if failure_rate > 0.25: # High failure rate
     new_threshold = max(MIN_REFLECTION_THRESHOLD,
     self.state.current_reflection_threshold -
     math.ceil(self.state.current_reflection_threshold *
     adjustment_factor))
     if new_threshold != self.state.current_reflection_threshold:
         self.logger.info(f"High tool failure rate ({failure_rate:.1%}).
         Lowering reflection threshold:
         {self.state.current_reflection_threshold} -> {new_threshold}")
         self.state.current_reflection_threshold = new_threshold
elif failure_rate < 0.05 and total_calls > 10: # Very low failure rate
```

```
            new_threshold = min(MAX_REFLECTION_THRESHOLD,
            self.state.current_reflection_threshold +
            math.ceil(self.state.current_reflection_threshold *
            adjustment_factor))
            if new_threshold != self.state.current_reflection_threshold:
                self.logger.info(f"Low tool failure rate ({failure_rate:.1%}).
                Raising reflection threshold:
                {self.state.current_reflection_threshold} -> {new_threshold}")
                self.state.current_reflection_threshold = new_threshold


    async def _trigger_promotion_checks(self):
        """Checks promotion criteria for recently accessed, eligible
        memories."""
        self.logger.debug("Running periodic promotion check...")
        tool_name_query = TOOL_QUERY_MEMORIES
        if not self._find_tool_server(tool_name_query):
        self.logger.warning(f"Skipping promotion check: Tool {tool_name_query}
        unavailable."); return

        candidate_memory_ids = set()
        try:
            episodic_results = await
            self._execute_tool_call_internal(tool_name_query, {"workflow_id":
            self.state.workflow_id, "memory_level": MemoryLevel.EPISODIC.value,
            "sort_by": "last_accessed", "sort_order": "DESC", "limit": 5,
            "include_content": False}, record_action=False)
            if episodic_results.get("success"):
            candidate_memory_ids.update(m.get('memory_id') for m in
            episodic_results.get("memories", []) if m.get('memory_id'))
            semantic_results = await
            self._execute_tool_call_internal(tool_name_query, {"workflow_id":
            self.state.workflow_id, "memory_level": MemoryLevel.SEMANTIC.value,
            "sort_by": "last_accessed", "sort_order": "DESC", "limit": 5,
            "include_content": False}, record_action=False)
            if semantic_results.get("success"):
            candidate_memory_ids.update(m.get('memory_id') for m in
            semantic_results.get("memories", []) if m.get('memory_id'))
            if candidate_memory_ids: self.logger.debug(f"Checking
            {len(candidate_memory_ids)} memories for promotion"); promo_tasks =
            [self._check_and_trigger_promotion(mem_id) for mem_id in
            candidate_memory_ids]; await asyncio.gather(*promo_tasks,
            return_exceptions=True)
            else: self.logger.debug("No recent eligible memories found for
            promotion check.")
        except Exception as e: self.logger.error(f"Error during periodic
        promotion check query: {e}", exc_info=False)

    async def _check_and_trigger_promotion(self, memory_id: str):
        """Checks a single memory for promotion and triggers it."""
```

```
    if not memory_id or not self._find_tool_server(TOOL_PROMOTE_MEM):
    return
    try:
        await asyncio.sleep(random.uniform(0.1, 0.5))
        promotion_result = await
        self._execute_tool_call_internal(TOOL_PROMOTE_MEM, {"memory_id":
        memory_id}, record_action=False)
        if promotion_result.get("success") and
        promotion_result.get("promoted"): self.logger.info(f"Memory
        {memory_id[:8]} promoted from
        {promotion_result.get('previous_level')} to
        {promotion_result.get('new_level')}", emoji_key="arrow_up")
    except Exception as e: self.logger.warning(f"Error in memory promotion
    check for {memory_id}: {e}", exc_info=False)

# --- Run Method (Main Loop - Incorporates Tier 1, 2 & 3) ---
async def run(self, goal: str, max_loops: int = 50):
    """Main agent execution loop."""
    if not await self.initialize(): self.logger.critical("Agent
    initialization failed."); return

    self.logger.info(f"Starting main loop. Goal: '{goal}' Max Loops:
    {max_loops}", emoji_key="arrow_forward")
    self.state.goal_achieved_flag = False

    while not self.state.goal_achieved_flag and self.state.current_loop <
    max_loops:
        if self._shutdown_event.is_set(): self.logger.info("Shutdown signal
        detected, exiting loop."); break
        self.state.current_loop += 1
        self.logger.info(f"--- Agent Loop
        {self.state.current_loop}/{max_loops} (RefThresh:
        {self.state.current_reflection_threshold}, ConThresh:
        {self.state.current_consolidation_threshold}) ---",
        emoji_key="arrows_counterclockwise")

        # Error Check
        if self.state.consecutive_error_count >= MAX_CONSECUTIVE_ERRORS:
            self.logger.error(f"Max consecutive errors
            ({MAX_CONSECUTIVE_ERRORS}) reached. Aborting.",
            emoji_key="stop_sign")
            if self.state.workflow_id: await
            self._update_workflow_status_internal("failed", "Agent failed
            due to repeated errors.")
            break

        # 1. Gather Context
        context = await self._gather_context()
        if context.get("status") == "No Active Workflow":
            self.logger.warning("No active workflow. Agent must create
            one.")
```

```
            self.state.current_plan = [PlanStep(description=f"Create the
            primary workflow for goal: {goal}")]
            self.state.needs_replan = False
        elif "errors" in context and context.get("errors"):
            self.logger.warning(f"Context gathering encountered errors:
            {context['errors']}. Proceeding cautiously.")

        # 2. Decide
        agent_decision = await self._call_agent_llm(goal, context)

        # 3. Act
        decision_type = agent_decision.get("decision")
        last_tool_result_content = None

        # Get Current Plan Step and Dependencies
        current_plan_step: Optional[PlanStep] = self.state.current_plan[0]
        if self.state.current_plan else None
        planned_dependencies_for_step: Optional[List[str]] =
        current_plan_step.depends_on if current_plan_step else None

        # Update Plan based on LLM suggestion FIRST
        if agent_decision.get("updated_plan_steps"):
            proposed_plan = agent_decision["updated_plan_steps"]
            if isinstance(proposed_plan, list) and all(isinstance(step,
            PlanStep) for step in proposed_plan):
                self.state.current_plan = proposed_plan;
                self.logger.info(f"Plan updated by LLM with
                {len(self.state.current_plan)} steps.");
                self.state.needs_replan = False
                current_plan_step = self.state.current_plan[0] if
                self.state.current_plan else None;
                planned_dependencies_for_step =
                current_plan_step.depends_on if current_plan_step else
                None
            else: self.logger.warning("LLM provided updated_plan_steps in
            unexpected format, ignoring.")

        # Execute Action
        if decision_type == "call_tool":
            tool_name = agent_decision.get("tool_name"); arguments =
            agent_decision.get("arguments", {})
            if not tool_name: self.logger.error("LLM requested tool call
            but provided no tool name."); self.state.last_action_summary =
            "Agent error: Missing tool name.";
            self.state.last_error_details = {"agent_decision_error":
            "Missing tool name"}; self.state.consecutive_error_count += 1;
            self.state.needs_replan = True
            else:
                self.logger.info(f"Agent requests tool: {tool_name} with
                args: {arguments}", emoji_key="wrench")
```

```
            last_tool_result_content = await
            self._execute_tool_call_internal(tool_name, arguments,
            True, planned_dependencies_for_step)
            if isinstance(last_tool_result_content, dict) and not
            last_tool_result_content.get("success"):
                self.state.needs_replan = True
                if last_tool_result_content.get("status_code") == 412:
                self.logger.warning(f"Tool execution failed due to
                unmet prerequisites:
                {last_tool_result_content.get('error')}")
                else: self.logger.warning(f"Tool execution failed:
                {last_tool_result_content.get('error')}")

    elif decision_type == "thought_process":
        thought_content = agent_decision.get("content", "No thought
        content provided.")
        self.logger.info(f"Agent reasoning:
        '{thought_content[:100]}...'. Recording.",
        emoji_key="thought_balloon")
        if self.state.workflow_id:
            # Use current_thought_chain_id if available
            thought_args = {"workflow_id": self.state.workflow_id,
            "content": thought_content, "thought_type":
            ThoughtType.INFERENCE.value}
            if self.state.current_thought_chain_id:
            thought_args["thought_chain_id"] =
            self.state.current_thought_chain_id
            try: thought_result = await
            self._execute_tool_call_internal(TOOL_RECORD_THOUGHT,
            thought_args, True); assert thought_result.get("success")
            except Exception as e: self.logger.error(f"Failed to
            record thought: {e}", exc_info=False);
            self.state.consecutive_error_count += 1;
            self.state.last_action_summary = f"Error recording
            thought: {str(e)[:100]}"; self.state.needs_replan = True;
            self.state.last_error_details = {"tool":
            TOOL_RECORD_THOUGHT, "error": str(e)}
        else: self.logger.warning("No workflow to record thought.");
        self.state.last_action_summary = "Agent provided reasoning,
        but no workflow active."

    elif decision_type == "complete":
        summary = agent_decision.get("summary", "Goal achieved.");
        self.logger.info(f"Agent signals completion: {summary}",
        emoji_key="tada"); self.state.goal_achieved_flag = True;
        self.state.needs_replan = False
        if self.state.workflow_id: await
        self._update_workflow_status_internal("completed", summary)
        break

    elif decision_type == "error":
```

```
                error_msg = agent_decision.get("message", "Unknown agent
                error"); self.logger.error(f"Agent decision error:
                {error_msg}", emoji_key="x"); self.state.last_action_summary =
                f"Agent decision error: {error_msg[:100]}";
                self.state.last_error_details = {"agent_decision_error":
                error_msg}; self.state.consecutive_error_count += 1;
                self.state.needs_replan = True
                if self.state.workflow_id:
                    try:
                      await
                      self._execute_tool_call_internal(TOOL_RECORD_THOUGHT,
                      {"workflow_id": self.state.workflow_id, "content":
                      f"Agent Error: {error_msg}", "thought_type":
                      ThoughtType.CRITIQUE.value}, False)
                    except Exception: pass

            else: self.logger.warning(f"Unhandled decision: {decision_type}");
            self.state.last_action_summary = "Unknown agent decision.";
            self.state.consecutive_error_count += 1; self.state.needs_replan =
            True; self.state.last_error_details = {"agent_decision_error":
            f"Unknown type: {decision_type}"}

            # 4. Update Plan (Fallback if LLM didn't provide one)
            if not agent_decision.get("updated_plan_steps"):
                await self._update_plan(context, agent_decision,
                last_tool_result_content)

            # 5. Periodic Tasks (Enhanced for Tier 3)
            await self._run_periodic_tasks()

            # 6. Save State Periodically
            if self.state.current_loop % 5 == 0: await
            self._save_agent_state()

            # 7. Loop Delay
            await asyncio.sleep(random.uniform(0.8, 1.2))

        # --- End of Loop ---
        self.logger.info("--- Agent Loop Finished ---", emoji_key="stopwatch")
        await self._cleanup_background_tasks(); await self._save_agent_state()
        if self.state.workflow_id and not self._shutdown_event.is_set():
            final_status = "completed" if self.state.goal_achieved_flag else
            "incomplete"
            self.logger.info(f"Workflow ended with status: {final_status}")
            await self._generate_final_report()
        elif not self.state.workflow_id: self.logger.info("Loop finished with
        no active workflow.")

    def _start_background_task(self, coro):
        """Creates and tracks a background task."""
```

```python
        task = asyncio.create_task(coro); self.state.background_tasks.add(task);
        task.add_done_callback(self.state.background_tasks.discard)

    async def _cleanup_background_tasks(self):
        """Cancels and awaits completion of any running background tasks."""
        if self.state.background_tasks:
            self.logger.info(f"Cleaning up {len(self.state.background_tasks)}
            background tasks...")
            cancelled_tasks = []; [task.cancel() for task in
            list(self.state.background_tasks) if not task.done() and
            cancelled_tasks.append(task)]
            if cancelled_tasks: await asyncio.gather(*cancelled_tasks,
            return_exceptions=True); self.logger.debug(f"Cancelled
            {len(cancelled_tasks)} background tasks.")
            self.logger.info("Background tasks cleaned up.");
            self.state.background_tasks.clear()

    async def signal_shutdown(self):
        """Initiates graceful shutdown."""
        self.logger.info("Graceful shutdown signal received.",
        emoji_key="wave"); self._shutdown_event.set(); await
        self._cleanup_background_tasks()

    async def shutdown(self):
        """Performs final cleanup and state saving."""
        self.logger.info("Shutting down agent loop...",
        emoji_key="power_button"); self._shutdown_event.set(); await
        self._cleanup_background_tasks(); await self._save_agent_state();
        self.logger.info("Agent loop shutdown complete.",
        emoji_key="checkered_flag")

    async def _update_workflow_status_internal(self, status: str, message:
    Optional[str] = None):
        """Internal helper to update workflow status via tool call."""
        if not self.state.workflow_id: return
        try: status_value = WorkflowStatus(status.lower()).value
        except ValueError: self.logger.warning(f"Invalid workflow status
        '{status}'. Using 'failed'."); status_value =
        WorkflowStatus.FAILED.value
        tool_name = TOOL_UPDATE_WORKFLOW_STATUS
        if not self._find_tool_server(tool_name): self.logger.error(f"Cannot
        update status: Tool {tool_name} unavailable."); return
        try: await self._execute_tool_call_internal(tool_name, {"workflow_id":
        self.state.workflow_id, "status": status_value, "completion_message":
        message}, record_action=False)
        except Exception as e: self.logger.error(f"Error marking workflow
        {self.state.workflow_id} as {status_value}: {e}", exc_info=False)

    async def _generate_final_report(self):
        """Generates and logs a final report using the memory tool."""
        if not self.state.workflow_id: return
```

```
        self.logger.info(f"Generating final report for workflow
        {self.state.workflow_id}...", emoji_key="scroll")
        tool_name = TOOL_GENERATE_REPORT
        if not self._find_tool_server(tool_name): self.logger.error(f"Cannot
        generate report: Tool {tool_name} unavailable."); return
        try:
            report_result_content = await
            self._execute_tool_call_internal(tool_name, {"workflow_id":
            self.state.workflow_id, "report_format": "markdown", "style":
            "professional"}, record_action=False)
            if isinstance(report_result_content, dict) and
            report_result_content.get("success"): report_text =
            report_result_content.get("report", "Report content missing.");
            self.mcp_client.safe_print("\n--- FINAL WORKFLOW REPORT ---\n" +
            report_text + "\n--- END REPORT ---")
            else: self.logger.error(f"Failed to generate final report:
            {report_result_content.get('error', 'Unknown error')}")
        except Exception as e: self.logger.error(f"Exception generating final
        report: {e}", exc_info=True)

    def _find_tool_server(self, tool_name: str) -> Optional[str]:
        """Finds an active server providing the specified tool."""
        if self.mcp_client and self.mcp_client.server_manager:
            if tool_name in self.mcp_client.server_manager.tools:
                 server_name =
                 self.mcp_client.server_manager.tools[tool_name].server_name
                 if server_name in
                 self.mcp_client.server_manager.active_sessions: return
                 server_name
                 else: self.logger.debug(f"Server '{server_name}' for tool
                 '{tool_name}' is not active.")
            elif tool_name.startswith("core:") and "CORE" in
            self.mcp_client.server_manager.active_sessions: return "CORE"
        self.logger.debug(f"Tool '{tool_name}' not found on any active server.")
        return None

    async def _check_workflow_exists(self, workflow_id: str) -> bool:
        """Checks if a workflow ID exists using list_workflows tool."""
        self.logger.debug(f"Checking existence of workflow {workflow_id} using
        list_workflows (potentially inefficient).")
        tool_name = TOOL_LIST_WORKFLOWS
        if not self._find_tool_server(tool_name): self.logger.error(f"Cannot
        check workflow: Tool {tool_name} unavailable."); return False
        try:
             result = await self._execute_tool_call_internal(tool_name,
             {"limit": 500}, record_action=False)
             if isinstance(result, dict) and result.get("success"): wf_list =
             result.get("workflows", []); return any(wf.get("workflow_id") ==
             workflow_id for wf in wf_list)
             return False
```

```
        except Exception as e: self.logger.error(f"Error checking WF
        {workflow_id}: {e}"); return False

# --- Main Execution Block ---
async def run_agent_process(mcp_server_url: str, anthropic_key: str, goal: str,
max_loops: int, state_file: str, config_file: Optional[str]):
    """Sets up and runs the agent process."""
    if not MCP_CLIENT_AVAILABLE: print("|\\xmark| ERROR: MCPClient dependency
    not met."); sys.exit(1)
    mcp_client_instance = None; agent_loop_instance = None; exit_code = 0;
    printer = print

    try:
        printer("Instantiating MCP Client...")
        mcp_client_instance = MCPClient(config_path=config_file)
        if hasattr(mcp_client_instance, 'safe_print'): printer =
        mcp_client_instance.safe_print
        if not mcp_client_instance.config.api_key:
            if anthropic_key: printer("Using provided Anthropic API key.");
            mcp_client_instance.config.api_key = anthropic_key;
            mcp_client_instance.anthropic =
            AsyncAnthropic(api_key=anthropic_key)
            else: raise ValueError("Anthropic API key missing.")
        printer("Setting up MCP Client...")
        await mcp_client_instance.setup(interactive_mode=False)
        printer("Instantiating Agent Master Loop...")
        agent_loop_instance =
        AgentMasterLoop(mcp_client_instance=mcp_client_instance,
        agent_state_file=state_file)

        loop = asyncio.get_running_loop()
        def signal_handler_wrapper(signum, frame):
            log.warning(f"Signal {signal.Signals(signum).name} received.
            Initiating graceful shutdown.")
            if agent_loop_instance:
            asyncio.create_task(agent_loop_instance.signal_shutdown())
            else: loop.stop()
        for sig in [signal.SIGINT, signal.SIGTERM]:
            try: loop.add_signal_handler(sig, signal_handler_wrapper, sig,
            None)
            except ValueError: log.debug(f"Signal handler for {sig} already
            exists.")
            except NotImplementedError: log.warning(f"Signal handling for
            {sig} not supported on this platform.")

        printer("Running Agent Loop...")
        await agent_loop_instance.run(goal=goal, max_loops=max_loops)

    except KeyboardInterrupt: printer("\n[yellow]Agent loop interrupt handled
    by signal handler.[/yellow]"); exit_code = 130
```

```
    except Exception as main_err: printer(f"\n|\\xmark| Critical error:
    {main_err}"); log.critical("Top-level execution error", exc_info=True);
    exit_code = 1
    finally:
        printer("Initiating final shutdown sequence...")
        if agent_loop_instance: printer("Shutting down agent loop..."); await
        agent_loop_instance.shutdown()
        if mcp_client_instance: printer("Closing MCP client..."); await
        mcp_client_instance.close()
        printer("Agent execution finished.")
        if __name__ == "__main__": await asyncio.sleep(0.5);
        sys.exit(exit_code)

if __name__ == "__main__":
    MCP_SERVER_URL = os.environ.get("MCP_SERVER_URL", "http://localhost:8013")
    ANTHROPIC_API_KEY = os.environ.get("ANTHROPIC_API_KEY")
    # --- Updated Goal for Tier 3 Testing ---
    AGENT_GOAL = os.environ.get("AGENT_GOAL",
        "Create workflow 'Tier 3 Test'. Goal: Research 'Quantum Computing impact
        on Cryptography'. "
        "Plan: 1. Create a new thought chain for 'Cryptography Research'. 2.
        Search memory for existing info (hybrid search). 3. Perform simulated
        web search (store results as memory). 4. Consolidate findings. 5.
        Reflect on progress and potential gaps. 6. Mark workflow complete."
    )
    MAX_ITERATIONS = int(os.environ.get("MAX_ITERATIONS", "30")) # Increased
    slightly for more complex goal
    AGENT_STATE_FILENAME = os.environ.get("AGENT_STATE_FILE", AGENT_STATE_FILE)
    MCP_CLIENT_CONFIG_FILE = os.environ.get("MCP_CLIENT_CONFIG")

    if not ANTHROPIC_API_KEY: print("|\\xmark| ERROR: ANTHROPIC_API_KEY
    missing."); sys.exit(1)
    if not MCP_CLIENT_AVAILABLE: print("|\\xmark| ERROR: MCPClient dependency
    missing."); sys.exit(1)

    print(f"--- {AGENT_NAME} (Tier 1, 2 & 3) ---") # Updated name
    print(f"Memory System URL: {MCP_SERVER_URL}")
    print(f"Agent Goal: {AGENT_GOAL}")
    print(f"Max Iterations: {MAX_ITERATIONS}")
    print(f"State File: {AGENT_STATE_FILENAME}")
    print(f"Client Config: {MCP_CLIENT_CONFIG_FILE or 'Default'}")
    print(f"Log Level: {log.level}")
    print("Anthropic API Key: Found")
    print("-----------------------------------------")

    # --- Tool Simulation Setup ---
    async def simulate_web_search(query: str):
        log.info(f"[SIMULATED] Searching web for: {query}")
        await asyncio.sleep(0.5)
        # Simulate finding some relevant snippets
        results = [
```

```
            f"Quantum computers threaten RSA encryption due to Shor's algorithm.
            (Source: Tech Journal)",
            f"Post-quantum cryptography (PQC) standards are being developed by
            NIST. (Source: NIST website)",
            f"Lattice-based cryptography is a leading candidate for PQC.
            (Source: Crypto Conf paper)"
        ]
        return {"success": True, "search_results": results}

    async def setup_and_run():
        """Wrapper to setup client and potentially register simulated
        tools."""
        # Placeholder for tool registration (adapt to your MCPClient)
        # client = MCPClient(...)
        # await client.register_tool_function("simulate:web_search",
        simulate_web_search)
        # await client.setup(...)
        # await run_agent_process(...) using this client instance
        await run_agent_process(MCP_SERVER_URL, ANTHROPIC_API_KEY, AGENT_GOAL,
        MAX_ITERATIONS, AGENT_STATE_FILENAME, MCP_CLIENT_CONFIG_FILE)

        asyncio.run(setup_and_run())
```