EideticEngine

Adaptive LLM Agent Orchestration with Multi-Level Memory and Performance-Driven Meta-Cognition

Jeffrey Emanuel

Originally written 13 April 2025

Revised April 18, 2025

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, hierarchical goal management, and adaptive self-management capabilities inspired by cognitive science.

EideticEngine integrates two key components:

1. **Unified Memory System (UMS)** – a persistent, multi-level cognitive workspace implemented on an asynchronous SQLite backend, featuring distinct memory types (`WORKING`, `EPISODIC`, `SEMANTIC`, `PROCEDURAL`), rich metadata, explicit typed linking, hybrid search, and integrated workflow tracking.
2. **Agent Master Loop (AML)** – an adaptive orchestrator that directs an LLM (specifically *Claude 3.7 Sonnet*) using the UMS. The AML enforces dependency-aware plans, supports hierarchical task decomposition via a Goal Stack, and orchestrates agent-driven meta-cognition for reflection, consolidation, and self-regulation.

Empirical analysis demonstrates that EideticEngine enables agents to navigate analytical and creative tasks with greater autonomy, robustness, and learning capacity than existing frameworks.

1 Introduction: Towards Cognitive Autonomy in LLM Agents

The remarkable generative and reasoning abilities of Large Language Models (LLMs) [1, 2] have catalysed the development of autonomous agents aimed at complex problem-solving.

Yet, the transition from impressive demonstrations to *robust*, *reliable* systems capable of sustained, adaptive operation across diverse, long-horizon tasks remains formidable [3]. Most current frameworks grapple with five structural limitations:

1. **Memory Persistence & Structure.** Reliance on ephemeral prompt context or simplistic buffers hinders long-term learning and causal reasoning [4, 5]. EideticEngine addresses this via its persistent, multi-level UMS.
2. **Planning & Execution.** Ad-hoc, reactive plans struggle with complex inter-dependencies. Lack of explicit prerequisite tracking leads to brittle execution [6, 7]. EideticEngine employs validated `PlanStep` objects.
3. **Adaptation & Learning.** Few agents reflect on past actions or synthesise experiences into durable knowledge [8, 9]. Our AML integrates explicit meta-cognitive tools and performance-driven adaptation.
4. **Hierarchical Task Management.** Managing nested goals without losing focus is challenging [10]. A dedicated Goal Stack in EideticEngine provides disciplined decomposition.
5. **Cognitive Coherence.** Agents often lack a unified internal state that ties memory, goals, reasoning, and actions together. EideticEngine unifies these via the AML state and UMS schema.

To overcome these gaps we introduce **EideticEngine**, an architecture built on two tightly-coupled pillars:

Unified Memory System (UMS). Inspired by multi-store human memory models [11, 12], the UMS offers distinct yet interconnected layers—working, episodic, semantic, and procedural—each with explicit roles, metadata, and lifecycle policies. Hybrid retrieval blends vector similarity (semantic) with keyword search (lexical) and graph traversal (relational).

Agent Master Loop (AML). Acting as a central executive, the AML maintains an explicit plan, enforces dependency checks, manages a hierarchical Goal Stack, and equips the LLM with meta-cognitive tools to reflect, consolidate, and promote knowledge. Adaptive thresholds modulate reflection frequency based on runtime statistics, introducing a *mental-momentum* bias that avoids over-thinking when progress is smooth.

Our hypothesis is that this integration yields agents capable of sustained, goal-directed reasoning with measurable gains in success rate, resilience, and knowledge accumulation. The remainder of the paper details the architecture, situates it within related work, and analyses its behaviour on demanding analytical and creative tasks.

2 Related Work: Building on and Departing From Existing Paradigms

EideticEngine differentiates itself from several established lines of research, drawing inspiration while addressing key limitations. Architectures like those presented in [13] have demonstrated the power of memory and reflection for creating believable, long-term agent behaviour, validating the core principles EideticEngine aims to systematise.

Standard LLM Agent Frameworks (LangChain, LlamaIndex, etc.):
While providing valuable abstractions for tool use (akin to systems like [14]) and basic memory (often vector stores or simple buffers), these frameworks typically lack: (i)

a deeply integrated, multi-level cognitive memory model (UMS) with explicit typed linking (`LinkType`), provenance tracking, relevance dynamics (`_compute_memory_relevance`), and dynamic evolution (`promote_memory_level`, `consolidate_memories`); (ii) structured planning (`PlanStep`) with robust dependency checking (`_check_prerequisites`) enforced by the loop; (iii) explicit hierarchical goal management (Goal Stack); (iv) agent-driven meta-cognitive tools for reflection and knowledge synthesis integrated with operational history (`memory_operations`); (v) adaptive control mechanisms (`_adapt_thresholds`) adjusting agent behaviour based on runtime performance metrics (error rates, memory ratios) and incorporating biases like Mental Momentum. EideticEngine offers a more opinionated and comprehensive *cognitive architecture* rather than solely a flexible toolkit.

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 the persistent, structured UMS, explicit `PlanStep` objects, dependency checks, categorised error handling (`last_error_details` with `type`), Goal Stack management, and meta-cognition for coherence.

Memory-Augmented LLMs (MemGPT [4], RAG [15]):
These focus on enhancing LLM capabilities by providing access to external or specialised memory during generation. EideticEngine complements this by providing a persistent, structured *internal* memory system (UMS) that tracks the agent's *own* experiences, thoughts, actions, and synthesised knowledge, enabling longitudinal learning and self-understanding beyond immediate context retrieval. The UMS serves as the agent's evolving world model and operational history, integrating episodic, semantic, and procedural knowledge. Other related work includes memory network concepts [5] and systems aiming to augment human memory [16].

LLM Planning & Reasoning Techniques (ReAct [7], Chain-of-Thought [17]):
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 externalises the plan (`AgentState.current_plan`), memory (UMS), goals (`AgentState.goal_stack`), and workflow state (UMS `workflows` table) into persistent structures, allowing for much longer, more complex tasks, error recovery across loops (using `last_error_details` and forcing replans), and persistent learning (memory promotion/consolidation) that influences future reasoning cycles. EideticEngine's `thought_chains` provide a structured way to manage and persist complex reasoning paths potentially generated using these techniques. Research on structured plan representation is also relevant [10].

Classical Cognitive Architectures (SOAR [18], ACT-R [Ref2, 19]):
These offer rich, theoretically grounded models of cognition, often based on symbolic rule systems or specialised 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 inspired by [**Ref19**, 11], specific integrations of episodic [**Ref7**, 20] and semantic memory, relevance decay, meta-cognition, goal decomposition [18]) but implements them within a practi-

cal, LLM-native framework (AML orchestrating UMS tools) built for autonomous task execution and tool use, leveraging the LLM itself for high-level reasoning and specific meta-cognitive tasks (`generate_reflection`, `consolidate_memories`). The need for such integrated cognitive architectures for language agents is increasingly recognised [21].

Meta-Reasoning and Reflection Research [Ref17, 8]:
While the importance of meta-cognition is recognised [8], few practical LLM agent systems incorporate explicit, agent-driven reflection and knowledge consolidation loops tied to performance metrics. Reflexion [9] demonstrated the power of self-reflection for iterative improvement. EideticEngine operationalises this through dedicated UMS tools (`generate_reflection`, `consolidate_memories`) triggered by the AML based on success counters or error conditions. Significantly, the *frequency* of these operations is made adaptive via `_adapt_thresholds`, which analyses runtime UMS statistics and internal agent state (`tool_usage_stats`), creating a dynamic feedback loop for self-regulation and 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 [**Ref19**, 11], the UMS implements distinct but interconnected memory levels (`MemoryLevel` enum: `WORKING`, `EPISODIC`, `SEMANTIC`, `PROCEDURAL`). The latter three are stored within the main `memories` table and differentiated by the `memory_level` column, while working memory is managed separately. This structure dictates default behaviours 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 the AML using UMS tools like `optimize_working_memory`. This tool uses relevance scoring (`compute_memory_relevance` custom SQL function) and strategies ('balanced', 'importance', 'recency', 'diversity') to maintain a focused attentional set. The `auto_update_focus` tool further refines this by identifying the most salient item (`focal_memory_id` in `cognitive_states`) within this active set based on complex heuristics (`_calculate_focus_score`). This concept aligns with models of working memory as an active workspace [22].

Episodic Memory:
Directly captures agent experiences. Records associated with specific `actions` (via `action_id` FK in `memories`), `thoughts` (`thought_id` FK, using deferred constraints), or `arti`

facts (artifact_id FK) default to this level. They often have shorter default ttl values (defined in DEFAULT_TTL), reflecting their time-bound nature. UMS tools like record_action_start and record_artifact automatically create linked episodic memories (memory_type = ACTION_LOG or ARTIFACT_CREATION). The importance of episodic memory in cognitive architectures has been explored previously [**Ref7**, 20]. See also [12].

Semantic Memory:
Represents generalised knowledge, facts, insights, or summaries. 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. Important thoughts (memory_type=REASONING_STEP) are often stored at this level. See [12].

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 (validated within 1.0–10.0 and 0.0–1.0 ranges respectively), critical for prioritisation and belief revision.

Temporal Dynamics:
created_at, updated_at, last_accessed (Unix timestamps) combined with access_count and ttl enable relevance calculations (via the custom compute_memory_relevance SQL function, incorporating MEMORY_DECAY_RATE) and automatic expiration (managed by the delete_expired_memories tool). This gives the memory system temporal dynamics often missing in static knowledge bases.

Provenance & Context:
Foreign keys (action_id, thought_id, artifact_id, using deferred constraints for the thought link) 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 Categorisation:
Besides memory_level and memory_type, memories have a JSON tags field, automatically populated with level and type, and allowing additional user-defined tags. This enables multi-dimensional categorisation and retrieval using the custom json_contains_all SQLite function within query_memories. A separate normalised tags table and junction tables (workflow_tags, etc.) manage the tag taxonomy.

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, including a `strength` score (0.0–1.0).

Automated Linking:
The `store_memory` tool can optionally trigger background auto-linking (parameter `suggest_links=True`). This uses semantic similarity (`_find_similar_memories`) to identify candidate links above a `link_suggestion_threshold` and creates them using a default `link_type` (typically `RELATED`) or contextually inferred types, bootstrapping the knowledge graph. The AML manages this via `_start_background_task` calling the internal `_run_auto_linking` helper.

Graph Traversal:
The `get_linked_memories` tool enables navigation of this graph structure, retrieving neighbours 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 recorded via `record_action_start` automatically generate corresponding `Episodic` memories (`memory_type=ACTION_LOG`). `record_action_completion` updates this linked memory. Memories can be explicitly linked back to actions (`action_id` FK).

Thought–Memory Coupling:
Thoughts recorded via `record_thought` can be directly linked to relevant memories (`relevant_memory_id` FK in `thoughts`). Important thoughts (goals, decisions, summaries, etc.) automatically generate linked `Semantic` memories (`memory_type=REASONING_STEP`) referencing the thought via the `thought_id` FK in `memories`. Deferred constraints handle the circular dependency.

Artifact–Memory Coupling:
Recording artifacts via `record_artifact` creates linked `Episodic` memories (`memory_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`, `dependencies`, `cognitive_states`, `reflections`, `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 stored in the `embeddings` table (using models like `text-embedding-3-small`, dimension tracked) and calculated via the external `EmbeddingService`. Finds conceptually related information using cosine similarity (`_find_similar_memories`), filtered by core metadata (workflow, level, type, TTL). Candidate fetching is optimised.

Keyword & Attribute Search (`query_memories`):
Utilises SQLite's FTS5 virtual table (`memory_fts`, indexing content, description, reasoning, tags) for fast keyword matching (`MATCH` operator), 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` (via `compute_memory_relevance`).

Hybrid Search (`hybrid_search_memories`):
Powerfully combines semantic similarity scores (from `_find_similar_memories`) 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. Scores are normalised before weighting.

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`, `get_action_dependencies` 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 an LLM (configured via `provider` and `model` arguments, using `get_provider`) to synthesise multiple source memories (selected via IDs, filters, or defaults) into more abstract `Semantic` forms (summaries, insights via `_generate_consolidation_prompt`) or `Procedural` forms. The results are stored as new memories (with derived importance/confidence) and linked back to the sources (`LinkType.GENERALIZES`), 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 generalisation. Promotion to Procedural is constrained by `memory_type` (e.g. must be `PROCEDURE` or `SKILL`). Thresholds (`min_access_count_*`, `min_confidence_*`) are configurable per promotion step.

Reflection Integration (`generate_reflection`):
While the reflection content is stored in the `reflections` table, the process analyses `me`

`mory_operations` logs (using `_generate_reflection_prompt` and an LLM), 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` and the singleton `DBConnection` manager ensures the UMS doesn't block the main agent loop during database I/O. Background tasks for linking/promotion are orchestrated by the AML (`_start_background_task`).

Optimised SQL:
Leverages SQLite features like WAL mode, comprehensive indexing (>30 indices), FTS5, memory mapping (`PRAGMA` settings specified in `SQLITE_PRAGMAS`), and custom functions (`compute_memory_relevance`, `json_contains_*`). Uses deferred constraints for circular references.

Structured Data Handling:
Consistent use of Enums (`MemoryLevel`, `MemoryType`, `LinkType`, `ActionStatus`, etc.) ensures data integrity. Careful serialisation/deserialisation (`MemoryUtils.serialize/deserialize`) handles complex data types and prevents errors, including handling potential `MAX_TEXT_LENGTH` overflows gracefully with structured error reporting. SQL identifiers are validated (`_validate_sql_identifier`).

Comprehensive Auditing:
The `memory_operations` table (`_log_memory_operation`) 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, hierarchical goal management, sophisticated context management, robust error handling, and, critically, adaptive meta-cognitive control, leveraging the UMS and an LLM reasoning core (`claude-3-7-sonnet-20250219`).

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` – aligned with `ActionStatus`), `assigned_tool` and `tool_args` (optional), `result_summary`, and crucially, a `depends_on` list containing the a

`ction_ids` of prerequisite steps that have been successfully recorded via `record_action_completion`.

LLM-Driven Plan Updates:
The AML enables the LLM (`_call_agent_llm`) to propose complete plan revisions by invoking the internal `AGENT_TOOL_UPDATE_PLAN` tool. This tool accepts a new list of `PlanStep` objects, validates them (including checking for cycles using `_detect_plan_cycle`), and replaces the `AgentState.current_plan`. This allows the LLM to dynamically modify the entire strategy based on new information or errors.

Dependency Enforcement (`_check_prerequisites`):
Before executing any `PlanStep` that involves a tool call, the `_execute_tool_call_internal` function extracts the `depends_on` list from the *current* plan step. It then calls `_check_prerequisites`, which 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 (`type=DependencyNotMetError`) is logged in `state.last_error_details`, and the `state.needs_replan` flag is set, forcing the LLM to reconsider the plan. This mechanism prevents cascading failures.

Heuristic Plan Update (`_apply_heuristic_plan_update`):
If the LLM *doesn't* call `AGENT_TOOL_UPDATE_PLAN`, this fallback mechanism provides basic plan progression. Based on the success or failure of the last executed tool call or thought recording, it marks the current step as 'completed' or 'failed', records a summary, removes completed steps, inserts an analysis step after failures, and adjusts meta-cognitive counters. This ensures the loop doesn't stall but prioritises explicit LLM-driven replanning when needed (`state.needs_replan=True`).

4.2 Hierarchical Goal Stack Management

EideticEngine introduces explicit management of hierarchical goals, allowing the agent to decompose complex objectives:
State Representation:
The `AgentState` maintains a `goal_stack` (list of goal dictionaries containing `goal_id`, `description`, `status`) and a `current_goal_id` pointing to the active goal (top of the stack).

Context Integration:
The `_gather_context` method retrieves details of the `current_goal` (using UMS tool `get_goal_details`) and provides a summary of the `goal_stack` to the LLM, ensuring decisions are goal-aware.

Agent-Driven Decomposition:
The LLM can push new sub-goals onto the stack using the UMS tool `push_sub_goal`. The AML (`_handle_workflow_and_goal_side_effects`) updates the `AgentState` stack and shifts the `current_goal_id` focus.

Status Tracking & Popping:
The LLM signals goal completion or failure using the UMS tool `mark_goal_status`. The AML updates the goal's status in the state stack and pops the finished goal, returning focus to the parent goal.

Workflow Integration:
Completion/failure of the root goal (when the stack becomes empty) sets the `state.goal_achieved_flag`, signalling the end of the main loop. Sub-workflow completion automatically triggers marking the corresponding parent goal's status. This distinguishes the goal stack (objective decomposition within a workflow) from the workflow stack (context switching).

4.3 Multi-Faceted Context Assembly (`_gather_context`)

The AML recognises 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` (with error `type`), the active `workflow_id`, `context_id`, `workflow_stack`, `current_thought_chain_id`, `current_plan`, and `needs_replan` flag.

Goal Context:
Includes details of the `current_goal` and a summary of the `goal_stack`.

Working Memory:
Queries `get_working_memory` to retrieve the IDs and summaries of memories currently in the agent's attentional focus, including the `focal_memory_id`. Results are timestamped (`retrieved_at`).

Proactive Goal-Relevant Memory:
Performs a `hybrid_search_memories` query using the description of the current plan step or goal to proactively fetch memories semantically or lexically related to the immediate task. Results limited by `CONTEXT_PROACTIVE_MEMORIES_FETCH_LIMIT` and timestamped.

Procedural Knowledge:
Executes another `hybrid_search_memories` query specifically filtered for `memory_level=procedural` using the plan step/goal description to find relevant "how-to" knowledge. Limited by `CONTEXT_PROCEDURAL_MEMORIES_FETCH_LIMIT` and timestamped.

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. Limited by respective `FETCH_LIMIT` constants and timestamped.

Relational Context:
Optionally uses `get_linked_memories` starting from a highly relevant memory (prioritising focal, then working, then important) to provide insight into the local knowledge-graph structure. Limited by `FETCH_LIMIT` and timestamped.

Meta-Cognitive Feedback:
Includes the summary (`state.last_meta_feedback`) from the last reflection or consolidation cycle.

Context Freshness:
Crucially, each major context component fetched from the UMS is tagged with a `retrieved_at` timestamp, allowing the LLM to assess the recency of information.

Context Compression:
Monitors the estimated token count (_estimate_tokens_anthropic) of the assembled context. If it exceeds CONTEXT_MAX_TOKENS_COMPRESS_THRESHOLD, it uses the specialised summarize_context_block UMS tool to compress less critical parts (e.g. detailed recent action logs) using context-type-specific prompts, aiming for CONTEXT_COMPRESSION_TARGET_TOKENS while preserving key information.

4.4 Adaptive Meta-Cognitive Control (_run_periodic_tasks, _adapt_thresholds)

This is a key adaptive aspect of the AML. It doesn't just execute tasks; it monitors and regulates its own cognitive processes, inspired by work like Reflexion [9]:
Triggering Meta-Cognition:
_run_periodic_tasks is executed each loop. It checks:
[label=()]

Dynamic action counts – comparing state.successful_actions_since_reflection vs. state.current_reflection_threshold and state.successful_actions_since_consolidation vs. state.current_consolidation_threshold;

state flags such as state.needs_replan;

periodic counters state.loops_since_* against constants (OPTIMIZATION_LOOP_INTERVAL, MEMORY_PROMOTION_LOOP_INTERVAL, STATS_ADAPTATION_INTERVAL, MAINTENANCE_INTERVAL).

Executing Meta-Cognitive Tools:
When triggered, it schedules and runs 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 _trigger_promotion_checks), and delete_expired_memories.

Feedback Loop:
Output from reflection/consolidation is stored in state.last_meta_feedback and fed into the *next* context assembly, letting the LLM react to its own self-assessment. Successful reflection often sets state.needs_replan=True.

Adaptive Threshold Adjustment (_adapt_thresholds):
Triggered periodically after compute_memory_statistics, it analyses UMS stats (e.g. episodic-memory ratio) and internal failure rates (tool_usage_stats). Heuristics then **modify** state.current_reflection_threshold and state.current_consolidation_threshold within configured bounds, dampened by THRESHOLD_ADAPTATION_DAMPENING.

Mental Momentum Bias:
If the agent is making stable progress (low failure rate, zero consecutive errors), MOMENTUM_THRESHOLD_BIAS_FACTOR biases *increases* in the reflection threshold, reducing interruptions and simulating a cognitive "flow state".

4.5 Robust Execution and Error Handling (`_execute_tool_call_internal`)

Tool Server Discovery:
Uses `_find_tool_server` to locate active servers providing requested tools via the `MCPClient`'s server manager. Handles internal tools (`AGENT_TOOL_UPDATE_PLAN`) separately.

Action Recording:
Wraps significant tool calls with UMS tools `record_action_start` and `record_action_completion`, ensuring operational history is captured and dependencies linked.

Dependency Recording:
After an action starts, `_record_action_start_internal` invokes `add_action_dependency` for all prerequisites listed in the relevant `PlanStep`.

Categorised Error Handling:
Catches execution errors, classifies them (e.g. `DependencyNotMetError`, `InvalidInputError`, `ServerUnavailable`), logs context in `state.last_error_details`, increments `state.consecutive_error_count`, and may set `state.needs_replan=True`. Exceeding `MAX_CONSECUTIVE_ERRORS` halts the loop.

Retry Mechanism (`_with_retries`):
Automatic retries with exponential back-off and jitter for configured transient exceptions, limited for non-idempotent operations and sensitive to shutdown signals.

Background Task Management:
`_start_background_task` launches non-blocking jobs (`_run_auto_linking`, `_check_and_trigger_promotion`) under concurrency limits, timeouts, and guaranteed cleanup via completion callbacks; shutdown calls `_cleanup_background_tasks`.

4.6 Thought Chain Management

Tracking:
The AML stores `state.current_thought_chain_id`; `_set_default_thought_chain_id` resolves a default on initialisation if absent.

Switching Chains:
When the LLM calls `create_thought_chain`, `_handle_workflow_and_goal_side_effects` updates `current_thought_chain_id`.

Automatic Injection:
The AML auto-injects `current_thought_chain_id` into `record_thought` if the LLM omits it, letting the model switch reasoning threads simply by choosing a chain.

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 external tools. The **Ultimate MCP Client** (`mcp_clie` `nt.py`) provides this critical "glue," offering a feature-rich environment specifically designed to support the complex needs of advanced cognitive agents like EideticEngine.

5.1 Unified Access to Distributed Capabilities

EideticEngine's power comes from leveraging diverse tools hosted across different servers. The MCP Client abstracts this complexity:

Server Management (`ServerManager`, `ServerConfig`):
Discovers (`discover_servers` via filesystem, registry, mDNS, and active port scanning), configures, connects (`connect_to_server`), monitors (`ServerMonitor`), and manages multiple MCP servers (STDIO and SSE via `RobustStdioSession` and `sse_client`). The AML thus accesses tools without caring about their location.

Centralised Tool/Resource Registry:
`ServerManager` aggregates tools, resources, and prompts from all servers into unified dictionaries, letting the AML present a single capability list to the LLM. Uses `format_tools_for_anthropic` to sanitise names.

Intelligent Routing:
When the AML executes a tool, the client routes the request to the correct server via `_find_tool_server` using the tool's registered `server_name`.

5.2 Robust Communication and Error Handling

Interacting with unreliable external processes demands resilience:

Asynchronous Architecture (`asyncio`, `httpx`, `aiosqlite`):
Fully async to prevent blocking the AML.

Specialised STDIO Handling (`RobustStdioSession`):
Custom session filters noisy stdout and resolves futures immediately.

STDIO Safety Wrappers:
`safe_stdout`, `get_safe_console`, `StdioProtectionWrapper` guard the JSON-RPC channel.

Retry Logic & Circuit Breaking:
`retry_with_circuit_breaker` decorator gives exponential backoff plus a simple breaker based on `ServerMetrics.error_rate`.

Graceful Shutdown:
Signal handlers and `close` routines terminate servers, flush caches, and persist state, complementing AML shutdown.

5.3 Enabling Advanced Agent Features

Features that directly enhance EideticEngine:

Streaming Support:
`process_streaming_query` and WebSocket `/ws/chat` deliver real-time LLM output and tool status.

Tool Result Caching (`ToolCache`):
In-memory and disk caching (via `diskcache`) with TTL mapping and dependency invalidation.

Conversation Management (`ConversationGraph`):
Branching chat history lets users or agents "fork" reasoning paths; async save/load to JSON.

Context Optimisation (`cmd_optimize`, `auto_prune_context`):
Summarises long histories with an LLM to stay within context windows.

Dynamic Prompting:
`cmd_prompt` and `apply_prompt_to_conversation` fetch template prompts from servers.

Observability (OpenTelemetry):
Tracing and metrics for performance insight.

Configuration Flexibility (`Config`):
YAML + env-var loading for keys, model prefs, discovery, caches.

Enhanced Discovery (mDNS & Port Scanning):
Zeroconf plus optional active scanning finds local servers automatically.

Platform Adaptation:
`adapt_path_for_platform` translates Windows paths to Linux/WSL, easing cross-platform use.

5.4 Developer Experience and Usability

Interactive CLI & Web UI:
Rich-powered CLI and a reactive FastAPI Web UI.

API Server (`FastAPI`):
REST and WebSocket endpoints expose full client and agent control.

Clear Status & Monitoring:
Live dashboards, progress bars, and detailed server-status commands via `rich`.

In essence, the Ultimate MCP Client provides the sophisticated, resilient, and observable infrastructure necessary for the AML to effectively harness the capabilities of the UMS and other tools within the Ultimate MCP Server ecosystem.

6 The Ultimate MCP 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 the **Ultimate MCP Server** instance. This server hosts the UMS tools (implemented in the `unified_memory_system_technical_analysis.md` codebase) alongside a powerful suite of complementary tools, expanding the agent's operational repertoire and enabling complex real-world workflows.

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

UMS as Tools:
The **UMS is implemented as a collection of tools within the broader Ultimate MCP Server**. The AML interacts with the UMS by invoking `unified_memory:*` tools (e.g. `store_memory`, `query_memories`) rather than direct DB calls.

Decoupling:
AML logic is decoupled from backend specifics (e.g. SQLite).

Extensibility:
New memory features or capabilities can be added as tools without touching AML code.

Standardised Interaction:
All memory, file access, web, and LLM calls flow through the unified MCP interface.

6.2 Core Ultimate MCP Server Capabilities (Beyond UMS)

Multi-Provider LLM Access:
Standardised `generate_completion`, `chat_completion`, `stream_completion` interfaces (OpenAI, Anthropic, Gemini), including key management, error handling, and token tracking.

Embedding Service:
`create_embeddings` with local caching; used heavily by UMS `store_memory` and search helpers.

Vector Database Service:
Manages vector collections; underpins semantic search (the current UMS does its own similarity maths via `sklearn`).

Caching Service:
Disk-backed caching for tool results via `diskcache`.

Prompt Management:
`PromptRepository` + `PromptTemplate` (Jinja2) for reusable prompts.

6.3 Synergistic Tools Enhancing EideticEngine's Capabilities

Advanced Extraction Tools:
- `extract_json` – structured JSON extraction with schema validation.
- `extract_table` – table parsing to JSON/Markdown.
- `extract_key_value_pairs` – key-value extraction for semantic memories.
- `extract_code_from_response` – cleans LLM code before storage.

Document Processing Tools:
- `chunk_document` – multiple chunking strategies.
- `summarize_document` – summaries stored back as `MemoryType.SUMMARY`.
- `extract_entities`, `generate_qa_pairs` – create new factual or question memories.

Secure Filesystem Tools:
- Safe `read_file`, `write_file`, directory listing, and artifact linking; path validation prevents escapes.

Local Text Processing Tools:
- Offline text manipulation via `rg`, `awk`, `sed`, `jq` with argument validation.

Web Browser Automation Tools:
- Playwright-based `browser_*` actions for live web interaction and artifact capture.

Optimisation & Meta Tools:
- `estimate_cost`, `compare_models`, `recommend_model` – cost reasoning before heavy LLM calls.
- `get_tool_info`, `get_llm_instructions` – introspect Gateway capabilities.

This rich ecosystem, accessed through the MCP Client, transforms EideticEngine from a system with sophisticated internal memory into one that interacts fluidly with external data, tools, and the web—while maintaining its cognitive state and history in the UMS.

7 Evaluation & Case Studies: Demonstrating Cognitive Capabilities

We evaluated EideticEngine's architecture through detailed simulations and analysis of its behavior on complex, multi-step tasks, tracing the agent's internal state (AML `AgentState`) and UMS interactions.

Case Study 1: Financial Market Analysis (Conceptual Walkthrough):
This task requires the agent to:
- **Structure & Goal Decomposition:** Create a workflow (`create_workflow`). Define the main goal. Use `push_sub_goal` to decompose analysis into sub-goals (e.g., "Analyze Interest Rates", "Analyze Equity Trends"). Potentially create separate thought chains (`create_thought_chain`) for each sub-goal.
- **Plan & Depend:** Generate a plan (`AGENT_TOOL_UPDATE_PLAN`) with `PlanStep` objects ... `_check_prerequisites` enforces this order.
- **Remember & Retrieve:** Store key economic facts (`store_memory`, `level=semantic`) ... assemble context via `_gather_context`.
- **Link:** Explicitly link related concepts (e.g., CPI data memory to market summary memory via `create_memory_link`). Background auto-linking (`_run_auto_linking`) connects related stored facts semantically.
- **Reflect & Adapt:** Based on performance ... prompts the LLM to revise the plan using `AGENT_TOOL_UPDATE_PLAN`.
- **Synthesize:** `consolidate_memories` generates a high-level insight (`type=insight`) ... storing it as a new `Semantic` memory.
- **Goal Completion:** Mark sub-goals complete (`mark_goal_status`) ... `state.goal_achieved_flag` is set.

Case Study 2: Creative Concept Development (Conceptual Walkthrough):
This task requires the agent to:
- **Ideate & Structure:** Brainstorm concepts (`record_thought`) ... manage development phases using the Goal Stack.
- **Develop & Persist:** Create character profiles ... retrieve these using `query_memories` when needed.
- **Iterate & Track:** Generate pilot script scenes ... dependencies ensure scenes precede draft updates.

- **Utilize Context:** `_gather_context` retrieves character profiles ... when the current plan step is ”Write Scene 2 featuring Alice”.
- **Finalize:** Retrieve the full draft artifact ... mark the final goal complete (`mark_goal_status`).

Analysis: Across both conceptual studies, the EideticEngine architecture facilitates successful completion of complex, multi-phase tasks. The UMS provides the necessary persistence, structure, and retrieval flexibility ... The Mental Momentum bias allows for more focused execution during productive phases.

8 Discussion: Implications of the EideticEngine Architecture

Beyond Reactive Agents:
EideticEngine moves agents from simple stimulus-response loops towards goal-directed, reflective, and adaptive behavior based on persistent internal state (UMS) and hierarchical objectives (Goal Stack).

Scalability for Complex Tasks:
Structured planning (`PlanStep`), explicit dependency management (`_check_prerequisites`), Goal Stack decomposition, and modular thought chains enable tackling problems that overwhelm simpler architectures.

Structured Learning and Adaptation:
Reflection, consolidation, memory promotion, and adaptive thresholds refine knowledge and strategy over time.

Introspection and Explainability:
Detailed UMS logging and visualisation tools provide unprecedented insight into the agent’s operational history and reasoning.

Foundation for General Capabilities:
Robust multi-level memory and adaptive control lay groundwork for future, more powerful reasoning cores.

Limitations: EideticEngine still relies heavily on the core LLM’s reasoning quality ... Error recovery depends on the LLM’s ability to interpret categorised errors and replan effectively.

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, hierarchical goal management via a Goal Stack, agent-driven meta-cognition (reflection, consolidation, promotion), and adaptive self-regulation of cognitive processes (including Mental Momentum bias), EideticEngine demonstrates a significant advance over existing agent paradigms. Its design supports sustained, goal-directed, adaptive, 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 ... quantify improvements in success, robustness, efficiency.

Advanced Adaptation & Learning:
Explore RL ... refine Mental Momentum heuristics.

Multi-Agent Systems:
Extend to collaborative tasks ... distributed UMS.

Real-Time Interaction:
Adaptations for tighter perception-action loops.

Theoretical Grounding:
Formalise loops and memory models against cognitive-science theory.

Hybrid Reasoning:
Integrate symbolic planners / knowledge-graph engines with UMS.

Advanced Memory Management:
Priority-based GC and compression for long-term storage.

Enhanced Introspection & Self-Modeling:
Deeper reasoning about UMS contents and biases.

Advanced Cognitive Simulation:
Counterfactual reasoning, basic emotional modelling.

Multimodal UMS:
Extend schema/tools to handle image, audio, video data.

Addendum: Implementation Details

This addendum provides supplementary technical details on the EideticEngine implementation.

10.1 Low-Level Implementation Considerations

Transaction Management:
UMS operations via tools such as `create_workflow`, `record_action_start`, `store_memory` use the `DBConnection.transaction` async context manager, ensuring atomic multi-table operations. Deferred FK constraints resolve the circular dependency between `thoughts` and `memories`.

Memory Compression/Summarisation:
Compression happens in `_gather_context` when token estimates exceed a threshold, invoking `summarize_context_block`. Direct pre-storage compression is not standard but could be added.

Embedding Caching:
`_store_embedding` leverages the Gateway's `EmbeddingService` cache to avoid redundant API calls.

Retry Logic Patterns:
External calls use `_with_retries`: configurable max retries, exponential back-off, jitter, and exception filtering; retries limited for non-idempotent ops.

Memory Expiration:
TTL-based removal via `delete_expired_memories`; no priority GC implemented yet.

10.2 Operational Statistics and Telemetry

Performance Metrics:
`with_tool_metrics` logs per-tool latency; AML tracks `AgentState.current_loop`.

Tool Usage Statistics:
`AgentState.tool_usage_stats` feeds `_adapt_thresholds`.

Memory Statistics:
`compute_memory_statistics` supplies aggregates for threshold adaptation.

Logging:
Rich logs + `memory_operations` audit trail.

Current Limitations:
No fine-grained telemetry yet (token heatmaps, reflection KPIs), though OpenTelemetry hooks exist.

10.3 Micro-Level Decision Handling

LLM Output Handling:
`_call_agent_llm` parses responses, using `AGENT_TOOL_UPDATE_PLAN` for plan changes.

Tool Selection:
Executes only the first requested tool; cost-based selection not yet implemented.

Error Classification System:
`_execute_tool_call_internal` maps exceptions to typed categories stored in `state.last_error_details`.

Conversation Management:
Branching history handled by the client's `ConversationGraph`, not AML/UMS.

10.4 Meta-Cognitive Mechanisms

Reflection:
`generate_reflection` on thresholds/errors.

Consolidation:
`consolidate_memories` synthesises knowledge.

Memory Promotion:
Background `promote_memory_level` via `_trigger_promotion_checks`.

Working Memory Optimisation:
`optimize_working_memory` + `auto_update_focus`.

Adaptive Thresholds:
Dynamic via `_adapt_thresholds`.

Mental Momentum:
Bias reduces reflection during stable progress.

10.5 Micro-Task Case Study Insights

10.5.1 Knowledge Integration Challenge

1. **Contradiction Detection:** Relies on LLM noticing conflicts; explicit detection tool not yet present.
2. **Authority Assessment:** Driven by LLM using metadata.
3. **Reconciliation Strategy:** LLM must formulate and store rule via `store_memory`.
4. **Knowledge Structure Update:** Explicit links via `create_memory_link`.

10.5.2 Dynamic Planning Adaptation

1. **Impact Analysis:** LLM reasons over `current_plan`; no dedicated traversal tool.
2. **Resource Reallocation:** Use artifacts/memories + Gateway tools to update outputs.
3. **Graceful Constraint Handling:** Searches procedural memories for prior fixes.
4. **Meta-Cognitive Efficiency:** Reflection frequency adjusts only via success/error counters.

10.5.3 Long-Duration Task Management

1. **Hibernate/Resume Capability:** State save/load; background tasks not persisted.
2. **Re-Contextualisation:** First `_gather_context` rebuilds context after resume.
3. **Time-Aware Reasoning:** Time decay in relevance + freshness indicators.
4. **Continuity Verification:** No explicit consistency check beyond successful reload.

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.

Annotation: The 2020 paper "Language models are few-shot learners" by Tom B. Brown and colleagues at OpenAI introduced GPT-3 and demonstrated its remarkable ability to perform a wide range of natural language tasks with few or even zero task-specific training examples, a capability termed "few-shot learning." This was a novel and important finding, suggesting that very large language models could generalize based solely on natural language prompts providing examples or instructions. The paper showcased GPT-3's performance across various tasks (text generation, QA, translation, basic reasoning), often rivaling fine-tuned models, highlighting emergent capabilities from scaling.

While acknowledging limitations, the primary contribution was demonstrating the unprecedented few-shot abilities of massive LLMs, fundamentally changing the NLP landscape. The relevance to EideticEngine is paramount, as it is designed to orchestrate an advanced LLM (`claude-3-5-sonnet-20240620`). The few-shot (and zero-shot) learning capabilities demonstrated by GPT-3 and expected in models like Claude are precisely what enable EideticEngine to guide the LLM to perform complex reasoning, planning, and meta-cognitive tasks through carefully designed prompts and interactions with its memory system, without requiring extensive task-specific fine-tuning of the agent itself.

. [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.

Annotation: The 2017 NeurIPS paper "Attention Is All You Need" introduced the **Transformer architecture**, revolutionizing sequence-to-sequence tasks and becoming the foundation for most modern large language models, including `claude-3-5-sonnet-20240620` used in EideticEngine. The key innovation was relying entirely on **attention mechanisms**, particularly **self-attention**, eliminating the need for recurrent or convolutional layers typical in NLP at the time. Self-attention allows the model to weigh the importance of different input parts when processing each position, enabling effective capture of long-range dependencies and significant parallelization during training. The Transformer consists of encoder and decoder stacks using multi-head self-attention and feed-forward networks. The paper demonstrated state-of-the-art machine translation results with faster training. The relevance to EideticEngine is fundamental: the underlying LLM's ability to process information, generate text, reason, plan, and engage in meta-cognition is directly enabled by the Transformer architecture introduced in this paper. Understanding the Transformer is crucial for comprehending the core capabilities that EideticEngine orchestrates and augments with its memory and control systems.

. [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.

Annotation: The 2023 arXiv preprint "A Survey on Large Language Model based Autonomous Agents" by Wang et al. provides a comprehensive overview of the burgeoning field of autonomous agents powered by LLMs. It likely covers architectures, capabilities (planning, memory, tool use), applications, challenges, and future directions. These agents aim to perform complex tasks without continuous human intervention by leveraging LLM reasoning combined with mechanisms for interacting with environments and tools. The survey likely discusses different approaches (memory systems, planning frameworks, control loops) and task domains (web browsing, content creation, research). It probably addresses open challenges like reliability, long-horizon task handling, safety, and ethics. EideticEngine falls squarely within the scope of this survey, presented as an architecture for orchestrating LLM agents with robust memory, structured planning, and meta-cognition. The survey likely highlights the importance of aspects EideticEngine focuses on (memory persistence/structure, planning, adaptation, hierarchical management) as key areas for advancing LLM agents. This survey provides valuable context, positioning EideticEngine within the broader landscape and affirming the significance of the problems it addresses.

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

Annotation: The 2023 arXiv preprint "MemGPT: Towards LLMs as Operating Systems" introduces MemGPT, a system designed to overcome the finite context window limitation of LLMs by providing them with external long-term memory management. MemGPT treats the LLM as a "kernel" and itself as a memory management unit, enabling interaction with a persistent external memory store. It uses techniques like in-context learning to teach the LLM memory management functions (e.g., recall, paging). The goal is to enable more complex, long-horizon tasks and coherent interactions over extended periods. MemGPT's novelty lies in its practical approach to extending LLM context. EideticEngine also addresses persistent memory for LLM agents but does so through a more structured, multi-level *internal* memory system (UMS) inspired by cognitive science, rather than primarily focusing on virtual context management. However, the underlying goal is similar: enabling LLM agents to possess persistent, accessible memory supporting complex cognitive processes. Concepts explored in MemGPT, such as memory management strategies and the LLM-memory interaction, are relevant to the design and operation of the UMS in EideticEngine, representing an alternative approach to the same fundamental challenge.

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

Annotation: The 2014 arXiv preprint "Memory Networks" by Weston, Chopra, and Bordes introduced a class of neural network architectures designed specifically for tasks requiring reasoning over large memories. Memory Networks consist of a memory component (storing facts/episodes, often as vectors) and inference components that can read from and write to this memory, often iteratively (multiple hops) to perform complex reasoning. The key idea was to allow the network to explicitly access and process relevant information from an external memory store when making predictions, moving beyond the implicit memory stored in network weights. They demonstrated effectiveness on tasks like QA and language modeling. The relevance to EideticEngine lies in the shared principle of augmenting a computational system (neural network or LLM) with an **explicit, accessible memory component** to enhance its capabilities. EideticEngine's Unified Memory System (UMS) serves this purpose for the LLM agent, providing a structured, persistent store. While Memory Networks are a specific neural architecture, the underlying concept of using external memory to support complex cognitive tasks is shared. EideticEngine's mechanisms for reading/writing to the UMS and retrieving relevant information are analogous to Memory Network operations, though implemented within a different LLM-centric framework.

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

Annotation: The 2023 GitHub repository "AutoGPT: An autonomous GPT-4 experiment" by Toran Bruce Richards introduced an influential early example of an autonomous agent driven by an LLM (GPT-4). AutoGPT aimed to perform complex tasks by having the LLM iteratively set goals, plan actions, and execute them (often using external tools like web search), without continuous human intervention. It maintained a short-term memory of recent thoughts/actions fed back to the LLM. AutoGPT gained attention for showcasing the potential of LLMs for autonomous behavior but also highlighted challenges like planning reliability, robust memory management, coherence, and error recovery over long runs. AutoGPT represents a pioneering effort in autonomous LLM agents, shaping subsequent research. EideticEngine builds upon lessons from projects like AutoGPT. While AutoGPT used simpler memory and planning, EideticEngine introduces a more structured and persistent Unified Memory System, explicit dependency-aware planning, and mechanisms for meta-cognition and adaptive control, aiming to address challenges faced by earlier agents. The core concept of an LLM in a continuous loop of thinking, planning, and acting is central to both AutoGPT and EideticEngine's Agent Master Loop.

. [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.

Annotation: The 2023 ICLR paper "ReAct: Synergizing Reasoning and Acting in Language Models" by Yao et al. introduced the **ReAct framework**, which enhances LLM capabilities on interactive tasks by **interleaving reasoning steps (thoughts) and action steps** within the LLM's generation process. Instead of generating only actions or only reasoning, ReAct prompts the LLM to produce a sequence like Thought -> Action -> Observation -> Thought -> Action -> ... . The LLM generates textual reasoning about the current state and plan, decides on an action (e.g., API call, environment interaction), receives an observation (result of the action), and uses that observation to inform the next reasoning step and subsequent action. This iterative synergy allows the agent to dynamically plan, gather information, and adapt its strategy based on real-time feedback. ReAct significantly outperformed prior methods on tasks requiring both reasoning and interaction (e.g., tool use QA, text games). ReAct is highly relevant prior art for EideticEngine, as its Agent Master Loop inherently orchestrates the LLM's reasoning and its interactions with the environment/tools (via PlanSteps). The ReAct paradigm of **interleaving reasoning and action** is fundamental to how EideticEngine operates. EideticEngine's structured plan execution, where the LLM reasons about the next step based on the current goal, context, and past results (observations stored in episodic memory), directly implements the ReAct principle within a broader cognitive architecture featuring persistent memory and more complex planning structures. Citing ReAct anchors EideticEngine's core operational loop in proven techniques for LLM-based agency and highlights how EideticEngine extends this concept with long-term memory and sophisticated plan management.

. [8] 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.

Annotation: The 2023 arXiv preprint "Self-adaptive agents using Large Language Models" explores developing autonomous LLM-based agents capable of adapting their behavior and strategies based on environmental feedback or performance analysis. The paper likely discusses using the LLM's reasoning to analyze outcomes, identify errors, and generate revised plans or strategies, possibly guided by external frameworks or specific prompting techniques for reflection. Self-adaptation is crucial for robust agents operating in dynamic environments. EideticEngine strongly emphasizes self-adaptation through its adaptive control layer, which dynamically adjusts meta-cognitive parameters based on operational statistics and internal state, enabling self-regulation. This paper's research on self-adaptive LLM agents is directly relevant to EideticEngine's design principles, particularly its focus on enabling the agent to reflect on performance and adjust behavior for optimal effectiveness. Mechanisms for achieving adaptation (prompting, external control) are important considerations in developing autonomous LLM agents like EideticEngine.

. [9] 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.

Annotation: The 2023 NeurIPS paper "Reflexion: Language Agents with Verbal Reinforcement Learning" introduced the **Reflexion** framework, enabling LLM agents to learn from mistakes through self-reflection and iterative refinement using verbal feedback as reinforcement. After a task attempt (especially failure), the agent generates a natural language **reflection** analyzing what went wrong. This reflection is stored (e.g., in an episodic memory or added to future prompts) and used to augment context in subsequent attempts, helping the agent avoid repeating errors. Reflexion demonstrated improved performance on challenging tasks by enabling agents to learn from their own experiences guided by simple reward signals (success/failure) and self-generated textual feedback. This is directly relevant prior art for EideticEngine's **adaptive meta-cognition**. EideticEngine incorporates agent-driven reflection (e.g., using the `generate_reflection` tool after PlanStep failures) inspired by this principle. While Reflexion focused on learning across multiple trials, EideticEngine integrates reflection into its continuous operational loop for analysis, knowledge consolidation (e.g., promoting insights from reflections to semantic memory), and strategy adaptation within a workflow. Reflexion provided a template for turning errors into lessons automatically using the LLM itself. By citing Shinn et al., we acknowledge that EideticEngine's mechanism for an agent learning from mistakes and updating its knowledge builds upon recently demonstrated successful strategies, validating its meta-cognitive loop and error handling capabilities.

. [10] 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.

Annotation: The 2025 arXiv preprint "Generating Structured Plan Representation of Procedures with LLMs" explores using large language models (LLMs) to generate structured representations of procedural plans, crucial for autonomous agents performing complex tasks. This paper focuses on prompting LLMs to output plans in formats explicitly capturing steps, order, and dependencies, possibly using templates or formal languages. The research likely evaluates the quality, correctness, and usability of these generated plans. The ability of LLMs to understand and generate structured plans is highly relevant to EideticEngine, which utilizes structured `PlanStep` objects with explicit dependency tracking. This paper's insights on eliciting structured procedural knowledge from LLMs could directly inform EideticEngine's plan generation and updating mechanisms. The focus on representing procedures structurally, including steps and dependencies, aligns well with EideticEngine's approach to managing complex tasks through well-defined plans.

. [11] 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.

Annotation: The 1968 chapter by Richard C. Atkinson and Richard M. Shiffrin, "Human memory: A proposed system and its control processes," presented the foundational multi-store model of memory, a landmark framework in cognitive psychology. This model proposed a **structural separation** of human memory into distinct stores: brief sensory memory, short-term memory (STM) with limited capacity, and long-term memory (LTM) with vast capacity. Critically, the model emphasized the **dynamic control processes** used by individuals to manage the flow of information between these stores, including attention, rehearsal (to maintain information in STM), encoding (transferring from STM to LTM), and retrieval (accessing information from LTM). The Atkinson-Shiffrin model was novel and important because it provided a clear, structured framework treating memory as an organized system, highlighting both distinct storage mechanisms and the active cognitive role in manipulating information. The relevance of this work to EideticEngine is significant, as its Unified Memory System (UMS) explicitly incorporates a multi-level memory structure inspired by such models. The distinctions within EideticEngine between Working Memory (analogous to STM), Episodic Memory, Semantic Memory, and Procedural Memory (all related to LTM) **directly echo the conceptual separation** proposed by Atkinson and Shiffrin. Furthermore, the control processes they described find parallels in EideticEngine's mechanisms for managing and utilizing its different memory levels, such as encoding new information, retrieving relevant context for the LLM, and processes like **consolidation** where experiences might be transformed into more durable long-term knowledge. This foundational work validates the principle that intelligent agents benefit from distinct memory mechanisms for recent versus long-term information, a core principle underpinning EideticEngine's unified architecture.

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

Annotation: In his seminal 1972 chapter "Episodic and semantic memory," Endel Tulving introduced the fundamental distinction between two major types of long-term memory: **episodic memory** and **semantic memory**. Episodic memory stores personally experienced events tied to specific times and places (context-dependent), allowing for "mental time travel." Semantic memory stores general world knowledge, facts, concepts, and vocabulary, independent of learning context. Tulving argued these systems serve different functions and have different properties. This distinction was highly influential, providing a core framework for memory research. Its relevance to EideticEngine is direct and significant. The architecture of EideticEngine's Unified Memory System (UMS) explicitly incorporates both **episodic memory** (storing the agent's interaction history, observations, PlanStep executions) and **semantic memory** (storing distilled facts, learned rules, general knowledge) as distinct levels. This design choice is directly inspired by Tulving's conceptualization, acknowledging prior art that multi-component memory architectures mirroring human cognition are beneficial. EideticEngine's episodic store enables recalling specific past events (important for reflection, avoiding repeated mistakes), while its semantic store allows applying general knowledge. Tulving's work also implies potentially different encoding and retrieval processes for each, informing EideticEngine's memory management operations like consolidation (potentially turning episodic details into semantic facts) and context-specific retrieval.

. [13] Joon Sung Park et al. "Generative Agents: Interactive Simulacra of Human Behavior". In: *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23)*. 2023. DOI: 10.1145/3586183.3606763. URL: https://doi.org/10.1145/3586183.3606763.

Annotation: The 2023 UIST paper "Generative Agents: Interactive Simulacra of Human Behavior" by Park et al. presented a novel architecture for creating LLM-powered agents capable of simulating believable, long-term human behavior in a sandbox environment.

The key innovation was an architecture enabling agents to remember experiences, retrieve relevant memories, reflect, and plan over extended periods (simulated days). Each agent had a **memory stream** (chronological record of experiences), mechanisms for **dynamic memory retrieval** based on relevance, recency, and importance, and periodic **reflection** to synthesize experiences into higher-level insights stored back into memory. This allowed agents to exhibit coherent behavior, maintain relationships, and pursue goals informed by their history. This work is highly relevant prior art for EideticEngine's Unified Memory System and planning capabilities. EideticEngine's UMS, storing episodic records and retrieving relevant context for the LLM, closely mirrors the memory architecture of Generative Agents. Techniques demonstrated by Park et al. for managing large memory streams and scoring relevance (e.g., using embeddings, time decay, importance scores) inform EideticEngine's **dynamic context assembly**. Furthermore, the **reflection and consolidation** mechanism in Generative Agents directly inspires EideticEngine's **meta-cognitive loop**, where the agent analyzes recent events to update its semantic memory or adjust strategies. Citing Park et al. emphasizes that EideticEngine's architectural choices for memory and planning have been validated in complex interactive settings, supporting the feasibility and importance of its multi-memory, reflection-capable, long-horizon approach for LLM-driven agents.

. [14] 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`.

Annotation: The 2023 arXiv preprint "HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in HuggingFace" presented a system using ChatGPT as a task planner and orchestrator for various AI models from the HuggingFace Hub. HuggingGPT decomposes complex user requests into sub-tasks solvable by specialized models (e.g., image generation, object detection). ChatGPT understands intent, plans the sequence, selects appropriate models, and coordinates their execution, passing outputs between models. This enables solving complex, multi-modal tasks via natural language by leveraging diverse AI tools. While EideticEngine orchestrates an LLM (`claude-3-5-sonnet-20240620`), its focus is more on the agent's internal cognitive architecture (memory, planning, metacognition). However, the high-level concept of using an LLM as a central controller to manage and utilize different tools/capabilities is a shared theme. In EideticEngine, the tools are primarily internal (UMS operations, reflection) or external tools accessed via an MCPClient, with the LLM deciding tool use based on its plan and goals. HuggingGPT highlights the power of LLMs in coordinating AI resources, a related aspect of building intelligent autonomous systems.

. [15] 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`.

Annotation: The 2020 NeurIPS paper "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks" by Patrick Lewis et al. introduced the Retrieval-Augmented Generation (RAG) framework. RAG combines pre-trained generative language models with an external knowledge retriever. When given an input, RAG first retrieves relevant documents from a knowledge source (e.g., Wikipedia), then feeds these documents along with the input query to the language model to generate the final output. This allows the model to leverage specific, up-to-date information beyond its training data, proving effective for knowledge-intensive tasks like QA and fact verification. RAG's novelty lies in enabling LLMs to flexibly incorporate external knowledge in an end-to-end manner. While EideticEngine focuses on an agent's internal cognitive architecture, the core concept of augmenting an LLM with access to a knowledge source is related. EideticEngine's Unified Memory System (UMS) serves as the *internal* knowledge source the LLM agent queries. The mechanisms within EideticEngine for retrieving relevant information from the UMS (episodic, semantic memories) and incorporating it into the LLM's context share conceptual similarities with RAG's retrieval step, though RAG typically uses external static corpora, whereas EideticEngine emphasizes the agent's dynamic, internally stored experiences and learned knowledge.

. [16] 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`.

Annotation: The 2025 arXiv preprint "Second Me: An AI-Native Memory Offload System" proposes a system designed to augment human memory by acting as a personal AI assistant that continuously records, stores, and retrieves life experiences, knowledge, and interactions. The core idea is an "offloaded" memory extension using AI for capture, processing, indexing, and retrieval. While focused on human memory augmentation, the underlying concepts of building a comprehensive, searchable AI-powered memory system are relevant to designing memory for autonomous agents. EideticEngine's Unified Memory System (UMS) shares the goal of creating a robust, structured repository for storing and retrieving information, albeit for an AI agent's experiences, reasoning, and knowledge. The challenges discussed in "Second Me" (efficient capture, effective indexing/retrieval, privacy) are pertinent to developing advanced AI memory systems like EideticEngine's UMS. The central theme of an AI acting as a memory extension, recalling relevant information on demand, connects "Second Me" to the design of sophisticated autonomous agents with persistent memory.

. [17] 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%5C_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html`.

Annotation: The 2022 NeurIPS paper "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models" by Wei et al. introduced a simple yet highly effective prompting technique called **chain-of-thought (CoT)**. CoT involves providing LLMs with few-shot examples that include not just the input and final answer but also the intermediate reasoning steps connecting them. This demonstration of a "chain of thought" enables LLMs to generate similar step-by-step reasoning for new problems, significantly improving performance on complex tasks requiring arithmetic, commonsense, or symbolic reasoning. The novelty was showing that this latent reasoning ability in LLMs could be unlocked through prompting alone. The paper explored CoT's effectiveness across models and tasks, showing substantial gains over standard prompting. Its relevance to EideticEngine lies in the use of an LLM (`claude-3-5-sonnet-20240620`) as its core reasoning engine. CoT prompting principles could potentially be employed within EideticEngine's prompts (e.g., for planning, reflection, or memory consolidation) to elicit more explicit, coherent, and effective multi-step reasoning from the LLM, potentially improving the overall performance and reliability of the autonomous agent.

. [18] 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.

Annotation: The 1987 article "SOAR: An architecture for general intelligence" by Laird, Newell, and Rosenbloom introduced the SOAR (State, Operator, And Result) cognitive architecture, a significant effort towards a unified theory of cognition and AGI. SOAR models problem-solving as selecting and applying operators in problem spaces. Its core mechanisms include a **production system** (if-then rules) for all decision-making stored in long-term memory, a **working memory** holding the current state and goals, and a learning mechanism called **chunking**, which creates new production rules summarizing successful resolutions of impasses. A key feature is **universal subgoaling**: when SOAR encounters an impasse (cannot select an operator), it automatically creates a subgoal to resolve it, naturally forming a **goal stack**. SOAR was novel for demonstrating how a single architecture using these principles could learn and handle diverse tasks, continuously acquiring knowledge. Its relevance to EideticEngine is significant. EideticEngine's *procedural memory* (storing PlanStep templates) and its learning mechanisms (e.g., creating generalized plans from successful executions) mirror aspects of Soar's production memory and chunking. The **hierarchical goal stack** in EideticEngine is a direct conceptual descendant of Soar's universal subgoaling. Soar's success in domains from **robotics to simulations** provides prior validation for unified approaches integrating memory and goal management. By citing Soar, we credit prior art for the concepts of a persistent **cognitive loop (decision cycle)**, integrated memory/goal structures, and learning from impasses, which heavily inform EideticEngine's Agent Master Loop and overall design.

. [19] 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.

Annotation: The 1996 paper by John R. Anderson titled "ACT: A simple theory of complex cognition" presents the ACT-R (Adaptive Control of Thought-Rational) theory, which stands as a prominent and influential unified theory of human cognition. This work aimed to provide a single framework capable of explaining a wide range of cognitive phenomena, from basic perceptual and motor skills to high-level reasoning and problem-solving. The core of ACT-R lies in its postulation of a cognitive architecture comprising several independent modules that interact to produce behavior. These modules include a declarative memory system for factual knowledge, a procedural memory system for skill-based knowledge represented as production rules (if-then statements), a goal module for maintaining current objectives, and several perceptual-motor modules for interacting with the environment. A central concept within ACT-R is the idea that cognition arises from the interaction and coordination of these modules, mediated by a central pattern matcher that selects and executes production rules based on the current state of declarative memory and the goal module. Learning in ACT-R occurs through two primary mechanisms: declarative learning, where new factual knowledge is encoded into memory, and procedural learning, where the strength and utility of production rules are adjusted based on their successful application. This theory introduced the novel idea of a computational framework that could simulate human cognitive processes with a high degree of detail, allowing researchers to develop and test specific hypotheses about how different cognitive functions operate and interact. The importance of ACT-R lies in its ability to provide a coherent and comprehensive account of cognition, influencing research across various fields of psychology, including memory, learning, problem-solving, and human-computer interaction. The distinction between declarative and procedural knowledge, the role of production rules in guiding behavior, and the concept of a modular cognitive architecture are all ideas that resonate with the design principles of EideticEngine, particularly in its separation of memory functions (semantic/episodic vs. procedural) and the control loop that orchestrates the agent's actions based on its internal state and knowledge.

. [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.

Annotation: The 2007 AAAI paper "Extending Cognitive Architecture with Episodic Memory" by Nuxoll and Laird discusses the importance and implementation of integrating episodic memory (memory for specific, personally experienced events) into the SOAR cognitive architecture. They argued that episodic memory is crucial for human-like learning, prediction, and handling novel situations. The paper presents their approach to adding an episodic memory module to SOAR, which automatically records snapshots of working memory content during decision cycles. The novelty lies in integrating retrieval into cognitive processing: the agent could cue episodic memory with the current situation to retrieve similar past experiences and use that information to guide current decisions. This is relevant prior art for EideticEngine's **Unified Memory System**, which explicitly includes an episodic component logging the agent's experiences (e.g., PlanStep execution history, observations). This paper demonstrates that providing AI agents with the ability to recall and leverage specific past events is beneficial for problem-solving and learning, allowing reflection on past actions or detection of changes – analogous to how EideticEngine's agent might review past attempts upon failure. It highlights technical considerations like memory indexing and retrieval, addressed by EideticEngine's implementation. Referencing this work shows a parallel: just as Soar's capabilities expanded with episodic memory, EideticEngine uses its stored episodes for **meta-cognitive reflection** and adaptation, a key part of its design validated by this prior research.

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

Annotation: The 2023 arXiv preprint "Cognitive Architectures for Language Agents" by Sumers et al. discusses the importance of designing structured architectural frameworks for agents powered by LLMs to achieve robust and general intelligence. Drawing inspiration from traditional cognitive architectures, the authors argue that simply relying on LLMs is insufficient and explore key components potentially needed for LLM-based agents, such as memory systems, planning mechanisms, goal management, and meta-cognitive abilities. The paper likely reviews different approaches to integrating these components with LLMs, emphasizing the need to move beyond simple reactive loops towards systems enabling reasoning, learning, and effective action over extended periods in complex environments. The development of cognitive architectures for language agents is framed as crucial for realizing the full potential of LLMs in autonomous systems. EideticEngine directly aligns with this paper's central theme, being explicitly presented as a cognitive architecture for orchestrating LLM agents. The paper's discussion of memory systems (cf. EideticEngine's UMS), planning (cf. EideticEngine's structured plans), goal management (cf. EideticEngine's Goal Stack), and meta-cognition (cf. EideticEngine's reflection tools) highlights the relevance of EideticEngine to broader research efforts. This paper provides a conceptual framework validating the role and importance of architectures like EideticEngine in advancing LLM-based autonomous agents.

. [22] 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.

Annotation: The 1974 chapter "Working memory" by Alan D. Baddeley and Graham J. Hitch presented a groundbreaking multi-component model that redefined short-term memory not as a passive buffer, but as an **active workspace** crucial for ongoing cognitive tasks. They proposed that working memory consists of a **central executive** (an attentional control system) coordinating subsidiary "slave" systems: the **phonological loop** (for verbal/auditory information) and the **visuospatial sketchpad** (for visual/spatial information). This model was novel in emphasizing that working memory is actively involved in **manipulating and maintaining information in real time** to support complex cognition like reasoning, comprehension, and problem-solving, holding goals, intermediate results, and perceptual inputs simultaneously. Subsequent work added the episodic buffer to integrate information. The relevance to EideticEngine is clear: its architecture must manage the LLM's immediate context (prompt, recent interactions, current plan step) as a form of working memory. EideticEngine's *Unified Memory System* implicitly distinguishes this temporary, actively managed context (akin to the workspace) from its long-term stores. The concept of a central executive maps well to EideticEngine's **Agent Master Loop**, which governs the agent's focus (what information from memory is brought into the LLM's context) and action selection. Baddeley & Hitch's work serves as prior art justifying the need for a dedicated, limited-capacity system for on-the-fly computation and control in any robust cognitive system. EideticEngine builds on this by implementing a dynamic working context for the LLM and mechanisms to assemble that context from long-term memory, effectively bridging the working memory concept with modern LLM prompt engineering.

A Appendix A: Unified Memory System (UMS) Technical Analysis

Technical Analysis of Unified Agent Memory and Cognitive System

System Overview and Architecture

The provided code implements a sophisticated ‘Unified Agent Memory and Cognitive System‘ designed for LLM agents. This system combines a structured memory hierarchy with process tracking, reasoning capabilities, and knowledge management. It’s built as an asynchronous Python module using SQLite for persistence with sophisticated memory organization patterns.

Core Architecture

The system implements a cognitive architecture with four distinct memory levels:

1. **Working Memory**: Temporarily active information (30-minute default TTL)
2. **Episodic Memory**: Experiences and event records (7-day default TTL)
3. **Semantic Memory**: Knowledge, facts, and insights (30-day default TTL)
4. **Procedural Memory**: Skills and procedures (90-day default TTL)

These are implemented through a SQLite database using ‘aiosqlite‘ for asynchronous operations, with optimized configuration:

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

# Memory management parameters
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
    "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
```

The system uses various SQLite optimizations through pragmas:

```
SQLITE_PRAGMAS = [
    "PRAGMA journal_mode=WAL",  # Write-Ahead Logging
    "PRAGMA synchronous=NORMAL",  # Balance durability and performance
    "PRAGMA foreign_keys=ON",
    "PRAGMA temp_store=MEMORY",
    "PRAGMA cache_size=-32000",  # ~32MB cache
    "PRAGMA mmap_size=2147483647",  # Memory-mapped I/O
    "PRAGMA busy_timeout=30000"  # 30-second timeout
]
```

Type System and Enumerations

The code defines comprehensive type hierarchies through enumerations:

Workflow and Action Status

```
class WorkflowStatus(str, Enum):
    ACTIVE = "active"
    PAUSED = "paused"
    COMPLETED = "completed"
    FAILED = "failed"
```

```python
    ABANDONED = "abandoned"

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

Content Classification

```python
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

```python
class MemoryLevel(str, Enum):
    WORKING = "working"
    EPISODIC = "episodic"
    SEMANTIC = "semantic"
    PROCEDURAL = "procedural"

class MemoryType(str, Enum):
    OBSERVATION = "observation"
    ACTION_LOG = "action_log"
    TOOL_OUTPUT = "tool_output"
    ARTIFACT_CREATION = "artifact_creation"
    REASONING_STEP = "reasoning_step"
    FACT = "fact"
    INSIGHT = "insight"
    PLAN = "plan"
    QUESTION = "question"
    SUMMARY = "summary"
    REFLECTION = "reflection"
    SKILL = "skill"
    PROCEDURE = "procedure"
    PATTERN = "pattern"
    CODE = "code"
    JSON = "json"
    URL = "url"
    TEXT = "text"

class LinkType(str, Enum):
    RELATED = "related"
    CAUSAL = "causal"
    SEQUENTIAL = "sequential"
    HIERARCHICAL = "hierarchical"
    CONTRADICTS = "contradicts"
    SUPPORTS = "supports"
    GENERALIZES = "generalizes"
    SPECIALIZES = "specializes"
    FOLLOWS = "follows"
    PRECEDES = "precedes"
    TASK = "task"
    REFERENCES = "references"
```

Database Schema

The system uses a sophisticated relational database schema with 15+ tables and numerous indices:

1. `workflows`: Tracks high-level workflow containers
2. `actions`: Records agent actions and tool executions
3. `artifacts`: Stores outputs and files created during workflows
4. `thought_chains`: Groups related thoughts (reasoning processes)
5. `thoughts`: Individual reasoning steps and insights
6. `memories`: Core memory storage with metadata and classification
7. `memory_links`: Associative connections between memories
8. `embeddings`: Vector embeddings for semantic search
9. `cognitive_states`: Snapshots of agent cognitive state
10. `reflections`: Meta-cognitive analysis outputs
11. `memory_operations`: Audit log of memory system operations
12. `tags, workflow_tags, action_tags, artifact_tags`: Tagging system
13. `dependencies`: Tracks dependencies between actions
14. `memory_fts`: Virtual FTS5 table for full-text search

Each table has appropriate foreign key constraints and indexes for performance optimization. The schema includes circular references between memories and thoughts, implemented with deferred constraints.

Connection Management

The database connection is managed through a sophisticated singleton pattern:

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

    _instance: Optional[aiosqlite.Connection] = None
    _lock = asyncio.Lock()
    _db_path_used: Optional[str] = None
    _init_lock_timeout = 15.0  # seconds

    # Methods for connection management, initialization, transaction handling, etc.
```

Key features include:

- Asynchronous context manager pattern with `__aenter__` and `__aexit__`
- Lock-protected singleton initialization with timeout
- Transaction context manager with automatic commit/rollback
- Schema initialization on first connection
- Custom SQLite function registration

Utility Functions

The system includes several utility classes and functions:

```python
def to_iso_z(ts: float) -> str:
    """Converts Unix timestamps to ISO-8601 with Z suffix."""
    # Implementation

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())

    # Methods for serialization, validation, sequence generation, etc.
```

Additional utility methods include:

- JSON serialization with robust error handling and truncation
- SQL identifier validation to prevent injection
- Tag processing to maintain taxonomies
- Access tracking to update statistics
- Operation logging for audit trails

Vector Embeddings and Semantic Search

The system integrates with an external embedding service:

```python
# Embedding configuration
DEFAULT_EMBEDDING_MODEL = "text-embedding-3-small"
EMBEDDING_DIMENSION = 384  # For the default model
SIMILARITY_THRESHOLD = 0.75
```

Implementation includes:

- `_store_embedding()`: Generates and stores vector embeddings with error handling
- `_find_similar_memories()`: Performs semantic search with cosine similarity and filtering
- Integration with scikit-learn for similarity calculations

Memory Relevance Calculation

The system implements a sophisticated memory relevance scoring algorithm:

```python
def _compute_memory_relevance(importance, confidence, created_at, access_count, last_accessed):
    """Computes a relevance score based on multiple factors."""
    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)
    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
    relevance = (decayed_importance * usage_boost * confidence * recency_factor)
    return min(max(relevance, 0.0), 10.0)
```

This function factors in:

- Base importance score (1-10 scale)
- Time-based decay of importance
- Usage frequency boost
- Confidence weighting
- Recency bias

Core Memory Operations

The system implements a comprehensive set of operations for memory management through tool functions, each designed with standardized error handling and metrics tracking via decorators (`@with_tool_metri`
`@with_error_handling`).

Memory Creation and Storage

The primary function for creating memories is `store_memory()`:

```python
async def store_memory(
    workflow_id: str,
    content: str,
    memory_type: str,
    memory_level: str = MemoryLevel.EPISODIC.value,
    importance: float = 5.0,
    confidence: float = 1.0,
    description: Optional[str] = None,
    reasoning: Optional[str] = None,
    source: Optional[str] = None,
    tags: Optional[List[str]] = None,
    ttl: Optional[int] = None,
    context_data: Optional[Dict[str, Any]] = None,
    generate_embedding: bool = True,
    suggest_links: bool = True,
    link_suggestion_threshold: float = SIMILARITY_THRESHOLD,
    max_suggested_links: int = 3,
    action_id: Optional[str] = None,
    thought_id: Optional[str] = None,
    artifact_id: Optional[str] = None,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function:

1. Validates input parameters (checking enum values, numeric ranges)
2. Generates a UUID for the memory
3. Records a timestamp
4. Establishes database connections
5. Performs existence checks for foreign keys
6. Inserts the memory record with all metadata
7. Optionally generates and stores vector embeddings for semantic search
8. Identifies and suggests semantic links to related memories
9. Updates workflow timestamps and logs the operation
10. Returns a structured result with memory details and suggested links

Key parameters include:

- `workflow_id`: Required container for the memory
- `content`: The actual memory content text
- `memory_type`: Classification (e.g., ”observation”, ”fact”, ”insight”)
- `memory_level`: Cognitive level (e.g., ”episodic”, ”semantic”)
- `importance/confidence`: Scoring for relevance calculations (1.0-10.0/0.0-1.0)
- `generate_embedding`: Whether to create vector embeddings for semantic search
- `suggest_links`: Whether to automatically find related memories

Memory creation automatically handles:

- Tag normalization and storage
- TTL determination (using defaults if not specified)
- Importance and confidence validation
- Creation of bidirectional links to similar memories

Memory Retrieval and Search

The system offers multiple retrieval mechanisms:

Direct Retrieval by ID

```
async def get_memory_by_id(
    memory_id: str,
    include_links: bool = True,
    include_context: bool = True,
    context_limit: int = 5,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function:

1. Fetches specific memory by ID
2. Updates access statistics
3. Optionally includes outgoing and incoming links
4. Optionally includes semantically similar memories as context
5. Checks TTL expiration

Keyword/Criteria-Based Search

```
async def query_memories(
    workflow_id: Optional[str] = None,
    memory_level: Optional[str] = None,
    memory_type: Optional[str] = None,
    search_text: Optional[str] = None,
    tags: Optional[List[str]] = None,
    min_importance: Optional[float] = None,
    max_importance: Optional[float] = None,
    min_confidence: Optional[float] = None,
    min_created_at_unix: Optional[int] = None,
    max_created_at_unix: Optional[int] = None,
    sort_by: str = "relevance",
    sort_order: str = "DESC",
    include_content: bool = True,
    include_links: bool = False,
    link_direction: str = "outgoing",
    limit: int = 10,
    offset: int = 0,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function provides powerful filtering capabilities:

- Workflow, level, type filters
- Full-text search via SQLite FTS5
- Tag filtering with array containment
- Importance/confidence ranges
- Creation time ranges
- Custom sorting options (`relevance`, `importance`, `created_at`, `updated_at`, etc.)
- Pagination via limit/offset
- Link inclusion options

Semantic/Vector Search

```
async def search_semantic_memories(
    query: str,
    workflow_id: Optional[str] = None,
    limit: int = 5,
    threshold: float = SIMILARITY_THRESHOLD,
    memory_level: Optional[str] = None,
    memory_type: Optional[str] = None,
    include_content: bool = True,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This implements vector similarity search:

1. Generates embeddings for the query
2. Finds memories with similar embeddings using cosine similarity
3. Applies threshold and filters
4. Updates access statistics for retrieved memories

Hybrid Search (Keyword + Vector)

```python
async def hybrid_search_memories(
    query: str,
    workflow_id: Optional[str] = None,
    limit: int = 10,
    offset: int = 0,
    semantic_weight: float = 0.6,
    keyword_weight: float = 0.4,
    memory_level: Optional[str] = None,
    memory_type: Optional[str] = None,
    tags: Optional[List[str]] = None,
    min_importance: Optional[float] = None,
    max_importance: Optional[float] = None,
    min_confidence: Optional[float] = None,
    min_created_at_unix: Optional[int] = None,
    max_created_at_unix: Optional[int] = None,
    include_content: bool = True,
    include_links: bool = False,
    link_direction: str = "outgoing",
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This sophisticated search function:

1. Combines semantic and keyword search results
2. Normalizes and weights scores from both approaches
3. Applies comprehensive filtering options
4. Performs efficient batched database operations for large result sets
5. Returns hybrid-scored results with detailed metadata

Memory Updating and Maintenance

```python
async def update_memory(
    memory_id: str,
    content: Optional[str] = None,
    importance: Optional[float] = None,
    confidence: Optional[float] = None,
    description: Optional[str] = None,
    reasoning: Optional[str] = None,
    tags: Optional[List[str]] = None,
    ttl: Optional[int] = None,
    memory_level: Optional[str] = None,
    regenerate_embedding: bool = False,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function allows updating memory attributes:

1. Dynamically builds SQL UPDATE clauses for changed fields
2. Optionally regenerates embeddings when content changes
3. Maintains timestamps and history
4. Returns detailed update information

```python
async def delete_expired_memories(db_path: str = DEFAULT_DB_PATH) -> Dict[str, Any]:
```

This maintenance function:

1. Identifies memories that have reached their TTL
2. Removes them in efficient batches
3. Handles cascading deletions via foreign key constraints
4. Logs operations for each affected workflow

Memory Linking and Relationships

```python
async def create_memory_link(
    source_memory_id: str,
    target_memory_id: str,
    link_type: str,
    strength: float = 1.0,
    description: Optional[str] = None,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function creates directional associations between memories:

1. Prevents self-linking
2. Validates link types against `LinkType` enum
3. Ensures link strength is in valid range (0.0-1.0)
4. Uses UPSERT pattern for idempotency
5. Returns link details

```python
async def get_linked_memories(
    memory_id: str,
    direction: str = "both",
    link_type: Optional[str] = None,
    limit: int = 10,
    include_memory_details: bool = True,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This retrieval function:

1. Gets outgoing and/or incoming links
2. Optionally filters by link type
3. Includes detailed information about linked memories
4. Updates access statistics
5. Returns structured link information

Thought Chains and Reasoning

The system implements a sophisticated thought chain mechanism for tracking reasoning:

Thought Chain Creation and Management

```python
async def create_thought_chain(
    workflow_id: str,
    title: str,
    initial_thought: Optional[str] = None,
    initial_thought_type: str = "goal",
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function:

1. Creates a container for related thoughts
2. Optionally adds an initial thought (goal, hypothesis, etc.)
3. Ensures atomicity through transaction management
4. Returns chain details with ID and creation timestamp

```
async def record_thought(
    workflow_id: str,
    content: str,
    thought_type: str = "inference",
    thought_chain_id: Optional[str] = None,
    parent_thought_id: Optional[str] = None,
    relevant_action_id: Optional[str] = None,
    relevant_artifact_id: Optional[str] = None,
    relevant_memory_id: Optional[str] = None,
    db_path: str = DEFAULT_DB_PATH,
    conn: Optional[aiosqlite.Connection] = None
) -> Dict[str, Any]:
```

This function records individual reasoning steps:

1. Validates thought type against `ThoughtType` enum
2. Handles complex foreign key relationships
3. Automatically determines target thought chain if not specified
4. Manages parent-child relationships for hierarchical reasoning
5. Creates links to related actions, artifacts, and memories
6. Automatically creates semantic memory entries for important thoughts
7. Supports transaction nesting through optional connection parameter

```
async def get_thought_chain(
    thought_chain_id: str,
    include_thoughts: bool = True,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This retrieval function:

1. Fetches chain metadata
2. Optionally includes all thoughts in sequence
3. Returns formatted timestamps and structured data

Thought Chain Visualization

```
async def visualize_reasoning_chain(
    thought_chain_id: str,
    output_format: str = "mermaid",
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function generates visualizations:

1. Retrieves the complete thought chain
2. For Mermaid format:
    - Generates a directed graph representation
    - Creates node definitions with appropriate shapes based on thought types
    - Handles parent-child relationships with connections
    - Adds external links to related entities
    - Implements CSS styling for different thought types
3. For JSON format:
    - Creates a hierarchical tree structure
    - Maps parent-child relationships
    - Includes all metadata
4. Returns the visualization content in the requested format

The Mermaid generation happens through a helper function `_generate_thought_chain_mermaid()` that constructs a detailed graph with styling:

```python
async def _generate_thought_chain_mermaid(thought_chain: Dict[str, Any]) -> str:
    # Implementation creates a complex Mermaid diagram with:
    # - Header node for the chain
    # - Nodes for each thought with type-specific styling
    # - Parent-child connections
    # - External links to actions, artifacts, memories
    # - Comprehensive styling definitions
```

Working Memory Management

The system implements sophisticated working memory with capacity management:

Working Memory Operations

```python
async def get_working_memory(
    context_id: str,
    include_content: bool = True,
    include_links: bool = True,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function:

1. Retrieves the current active memory set for a context
2. Updates access statistics
3. Optionally includes memory content
4. Optionally includes links between memories
5. Returns a structured view of working memory

```python
async def focus_memory(
    memory_id: str,
    context_id: str,
    add_to_working: bool = True,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function:

1. Sets a specific memory as the current focus of attention
2. Optionally adds the memory to working memory if not present
3. Ensures memory and context workflow consistency
4. Updates cognitive state records
5. Returns focus update confirmation

```python
async def _add_to_active_memories(conn: aiosqlite.Connection, context_id: str, memory_id: str) -> bool:
```

This internal helper function implements working memory capacity management:

1. Checks if memory is already in working memory
2. Enforces the `MAX_WORKING_MEMORY_SIZE` limit
3. When capacity is reached, computes relevance scores for all memories
4. Removes least relevant memory to make space
5. Returns success/failure status

```python
async def optimize_working_memory(
    context_id: str,
    target_size: int = MAX_WORKING_MEMORY_SIZE,
    strategy: str = "balanced",
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function performs optimization:

1. Implements multiple strategies:
   - `balanced`: Considers all relevance factors
   - `importance`: Prioritizes importance scores
   - `recency`: Prioritizes recently accessed memories
   - `diversity`: Ensures variety of memory types
2. Scores memories based on strategy
3. Selects optimal subset to retain
4. Updates the cognitive state
5. Returns detailed optimization results

```python
async def auto_update_focus(
    context_id: str,
    recent_actions_count: int = 3,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function implements automatic attention shifting:

1. Analyzes memories currently in working memory
2. Scores them based on relevance and recent activity
3. Uses the `_calculate_focus_score()` helper with sophisticated heuristics
4. Updates focus to the highest-scoring memory
5. Returns details of the focus shift

The focus scoring implements multiple weight factors:

```python
def _calculate_focus_score(memory: Dict, recent_action_ids: List[str], now_unix: int) -> float:
    """Calculate focus priority score based on multiple factors."""
    score = 0.0

    # Base relevance (importance, confidence, recency, usage)
    relevance = _compute_memory_relevance(...)
    score += relevance * 0.6  # Heavily weighted

    # Boost for recent action relationship
    if memory.get("action_id") in recent_action_ids:
        score += 3.0  # Significant boost

    # Type-based boosts for attention-worthy types
    if memory.get("memory_type") in ["question", "plan", "insight"]:
        score += 1.5

    # Memory level boosts
    if memory.get("memory_level") == MemoryLevel.SEMANTIC.value:
        score += 0.5
    elif memory.get("memory_level") == MemoryLevel.PROCEDURAL.value:
        score += 0.7

    return max(0.0, score)
```

Cognitive State Management

The system implements cognitive state persistence for context restoration:

```python
async def save_cognitive_state(
    workflow_id: str,
    title: str,
    working_memory_ids: List[str],
    focus_area_ids: Optional[List[str]] = None,
    context_action_ids: Optional[List[str]] = None,
    current_goal_thought_ids: Optional[List[str]] = None,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function:

1. Validates that all provided IDs exist and belong to the workflow

2. Marks previous states as not latest
3. Serializes state components
4. Records a timestamped cognitive state snapshot
5. Returns confirmation with state ID

```
async def load_cognitive_state(
    workflow_id: str,
    state_id: Optional[str] = None,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function:

1. Loads either a specific state or the latest state
2. Deserializes state components
3. Logs the operation
4. Returns full state details

```
async def get_workflow_context(
    workflow_id: str,
    recent_actions_limit: int = 10,
    important_memories_limit: int = 5,
    key_thoughts_limit: int = 5,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function builds a comprehensive context summary:

1. Fetches workflow metadata (title, goal, status)
2. Gets latest cognitive state
3. Retrieves recent actions
4. Includes important memories
5. Adds key thoughts (goals, decisions, reflections)
6. Returns a structured context overview

Action and Artifact Tracking

The system tracks all agent actions and created artifacts:

Action Management

```
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]:
```

This function:

1. Validates action type against `ActionType` enum
2. Requires reasoning explanation
3. Validates references to workflow, parent action, and related thought
4. Auto-generates title if not provided
5. Creates a corresponding episodic memory entry
6. Returns action details with ID and start time

```python
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",
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function:

1. Validates completion status (completed, failed, skipped)
2. Records tool execution result
3. Updates the action record
4. Optionally adds a concluding thought
5. Updates the linked episodic memory with outcome
6. Returns completion confirmation

```python
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]:
```

This function:

1. Retrieves details for one or more actions
2. Deserializes tool args and results
3. Includes associated tags
4. Optionally includes dependency relationships
5. Returns comprehensive action information

```python
async def get_recent_actions(
    workflow_id: str,
    limit: int = 5,
    action_type: Optional[str] = None,
    status: Optional[str] = None,
    include_tool_results: bool = True,
    include_reasoning: bool = True,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function:

1. Gets the most recent actions for a workflow
2. Applies type and status filters
3. Controls inclusion of potentially large fields (tool results, reasoning)
4. Returns a time-ordered action list

Action Dependencies

```python
async def add_action_dependency(
    source_action_id: str,
    target_action_id: str,
    dependency_type: str = "requires",
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function:

1. Creates an explicit dependency relationship between actions
2. Ensures actions belong to the same workflow
3. Handles duplicate dependency declarations

4. Returns dependency details

```python
async def get_action_dependencies(
    action_id: str,
    direction: str = "downstream",
    dependency_type: Optional[str] = None,
    include_details: bool = False,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function:

1. Retrieves actions that depend on this one (downstream) or
2. Retrieves actions this one depends on (upstream)
3. Optionally filters by dependency type
4. Optionally includes full action details
5. Returns structured dependency information

Artifact Management

```python
async def record_artifact(
    workflow_id: str,
    name: str,
    artifact_type: str,
    action_id: Optional[str] = None,
    description: Optional[str] = None,
    path: Optional[str] = None,
    content: Optional[str] = None,
    metadata: Optional[Dict[str, Any]] = None,
    is_output: bool = False,
    tags: Optional[List[str]] = None,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function:

1. Validates artifact type against `ArtifactType` enum
2. Handles content truncation for large text artifacts
3. Creates a corresponding episodic memory entry
4. Records relationships to creating action
5. Applies tags and metadata
6. Returns artifact details with ID

```python
async def get_artifacts(
    workflow_id: str,
    artifact_type: Optional[str] = None,
    tag: Optional[str] = None,
    is_output: Optional[bool] = None,
    include_content: bool = False,
    limit: int = 10,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function:

1. Lists artifacts for a workflow with filtering
2. Controls inclusion of potentially large content
3. Deserializes metadata
4. Returns artifact list with details

```python
async def get_artifact_by_id(
    artifact_id: str,
    include_content: bool = True,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function:

1. Retrieves a specific artifact by ID
2. Updates access stats for related memory
3. Returns complete artifact details

Meta-Cognitive Capabilities

The system implements sophisticated meta-cognitive functions:

Memory Consolidation

```python
async def consolidate_memories(
    workflow_id: Optional[str] = None,
    target_memories: Optional[List[str]] = None,
    consolidation_type: str = "summary",
    query_filter: Optional[Dict[str, Any]] = None,
    max_source_memories: int = 20,
    prompt_override: Optional[str] = None,
    provider: str = LLMGatewayProvider.OPENAI.value,
    model: Optional[str] = None,
    store_result: bool = True,
    store_as_level: str = MemoryLevel.SEMANTIC.value,
    store_as_type: Optional[str] = None,
    max_tokens: int = 1000,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function implements memory consolidation:

1. Allows selecting source memories:
   - Explicit memory ID list or
   - Query-based filtering or
   - Recent important memories from workflow
2. Supports multiple consolidation types:
   - `summary`: Comprehensive integration of information
   - `insight`: Pattern recognition and implications
   - `procedural`: Generalized steps or methods
   - `question`: Key information gaps or uncertainties
3. Generates LLM prompts with detailed instructions
4. Makes external LLM API calls to process memories
5. Automatically stores the result as a new memory
6. Creates bidirectional links to source memories
7. Returns consolidated content and details

The consolidation prompt generation is handled by `_generate_consolidation_prompt()`:

```python
def _generate_consolidation_prompt(memories: List[Dict], consolidation_type: str) -> str:
    # Formats memory details with truncation
    # Adds type-specific instruction templates:
    # - summary: comprehensive integration
    # - insight: pattern identification
    # - procedural: generalized methods
    # - question: information gaps
```

Reflection Generation

```python
async def generate_reflection(
    workflow_id: str,
    reflection_type: str = "summary",
    recent_ops_limit: int = 30,
    provider: str = LLMGatewayProvider.OPENAI.value,
    model: Optional[str] = None,
    max_tokens: int = 1000,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This meta-cognitive function:

1. Analyzes recent memory operations (from the operation log)
2. Supports multiple reflection types:
   - `summary`: Overview of recent activity
   - `progress`: Analysis of goal advancement
   - `gaps`: Knowledge and understanding deficits
   - `strengths`: Effective patterns and insights
   - `plan`: Strategic next steps
3. Generates sophisticated prompts using `_generate_reflection_prompt()`
4. Makes external LLM calls to perform analysis
5. Stores the reflection in the reflection table
6. Returns reflection content and metadata

Memory Promotion and Evolution

```
async def promote_memory_level(
    memory_id: str,
    target_level: Optional[str] = None,
    min_access_count_episodic: int = 5,
    min_confidence_episodic: float = 0.8,
    min_access_count_semantic: int = 10,
    min_confidence_semantic: float = 0.9,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function implements memory evolution:

1. Checks if a memory meets criteria for promotion to a higher level
2. Implements promotion paths:
   - Episodic → Semantic (experiences to knowledge)
   - Semantic → Procedural (knowledge to skills, with type constraints)
3. Applies configurable criteria based on:
   - Access frequency (demonstrates importance)
   - Confidence level (demonstrates reliability)
   - Memory type (suitability for procedural level)
4. Updates the memory level if criteria are met
5. Returns promotion status with reason

Text Summarization

```
async def summarize_text(
    text_to_summarize: str,
    target_tokens: int = 500,
    prompt_template: Optional[str] = None,
    provider: str = "openai",
    model: Optional[str] = None,
    workflow_id: Optional[str] = None,
    record_summary: bool = False,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This utility function:

1. Summarizes text content using LLM
2. Uses configurable prompt templates
3. Controls summary length via token targeting
4. Optionally stores summary as memory
5. Returns summary text and metadata

Context Summarization

```python
async def summarize_context_block(
    text_to_summarize: str,
    target_tokens: int = 500,
    context_type: str = "actions",
    workflow_id: Optional[str] = None,
    provider: str = LLMGatewayProvider.ANTHROPIC.value,
    model: Optional[str] = "claude-3-5-haiku-20241022",
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This specialized function:

1. Summarizes specific types of context (actions, memories, thoughts)
2. Uses custom prompts optimized for each context type
3. Designed for agent context window management
4. Returns focused summaries with compression ratio

Reporting and Visualization

The system implements sophisticated reporting capabilities:

```python
async def generate_workflow_report(
    workflow_id: str,
    report_format: str = "markdown",
    include_details: bool = True,
    include_thoughts: bool = True,
    include_artifacts: bool = True,
    style: Optional[str] = "professional",
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function creates comprehensive reports:

1. Fetches complete workflow details
2. Supports multiple formats:
   - `markdown`: Text-based structured report
   - `html`: Web-viewable report with CSS
   - `json`: Machine-readable structured data
   - `mermaid`: Diagrammatic representation
3. Implements multiple styling options:
   - `professional`: Formal business report style
   - `concise`: Brief summary focused on key points
   - `narrative`: Story-like descriptive format
   - `technical`: Data-oriented technical format
4. Uses helper functions for specific formats:
   - `_generate_professional_report()`
   - `_generate_concise_report()`
   - `_generate_narrative_report()`
   - `_generate_technical_report()`
   - `_generate_mermaid_diagram()`
5. Returns report content with metadata

Memory network visualization is implemented through:

```
async def visualize_memory_network(
    workflow_id: Optional[str] = None,
    center_memory_id: Optional[str] = None,
    depth: int = 1,
    max_nodes: int = 30,
    memory_level: Optional[str] = None,
    memory_type: Optional[str] = None,
    output_format: str = "mermaid",
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function:

1. Creates a visual representation of memory relationships
2. Supports workflow-wide view or centered on specific memory
3. Uses breadth-first search to explore links to depth limit
4. Applies memory type and level filters
5. Generates Mermaid diagram with:
    - Nodes styled by memory level
    - Links showing relationship types
    - Center node highlighting
6. Returns complete diagram code

Detailed Key Tool Functions (Additional Core Functionality)

Below I'll cover several more important tool functions in detail that implement key functionality:

LLM Integration

The system integrates with external LLM providers through the `llm_gateway` module:

```
from llm_gateway.constants import Provider as LLMGatewayProvider
from llm_gateway.core.providers.base import get_provider
```

This enables:

1. Dynamic provider selection (OpenAI, Anthropic, etc.)
2. Model specification
3. Standardized prompting
4. Response handling

Example LLM integration in consolidation:

```
provider_instance = await get_provider(provider)
llm_result = await provider_instance.generate_completion(
    prompt=prompt, model=model_to_use, max_tokens=max_tokens, temperature=0.7
)
reflection_content = llm_result.text.strip()
```

System Statistics and Metrics

```
async def compute_memory_statistics(
    workflow_id: Optional[str] = None,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function:

1. Computes comprehensive system statistics

2. Supports global or workflow-specific scope
3. Collects metrics on:
    - Total memory counts
    - Distribution by level and type
    - Confidence and importance averages
    - Temporal metrics (newest/oldest)
    - Link statistics by type
    - Tag frequencies
    - Workflow statuses
4. Returns structured statistical data

Workflow Listing and Management

```python
async def list_workflows(
    status: Optional[str] = None,
    tag: Optional[str] = None,
    after_date: Optional[str] = None,
    before_date: Optional[str] = None,
    limit: int = 10,
    offset: int = 0,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function:

1. Lists workflows with filtering options
2. Supports status, tag, and date range filters
3. Includes pagination
4. Returns workflow list with counts

```python
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]:
```

This function:

1. Creates a new workflow container
2. Creates default thought chain
3. Adds initial goal thought if provided
4. Supports workflow hierarchies via parent reference
5. Returns workflow details with IDs

```python
async def update_workflow_status(
    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]:
```

This function:

1. Updates workflow status (active, paused, completed, failed, abandoned)
2. Adds completion thought for terminal statuses
3. Updates tags
4. Returns status update confirmation

Database Schema Details and Implementation

The system's database schema represents a sophisticated cognitive architecture designed for tracking agent workflows, actions, thoughts, and memories. Let's examine its detailed structure:

Schema Creation and Initialization

The schema is defined in the `SCHEMA_SQL` constant, which contains all DDL statements. The system uses a transactional approach to schema initialization:

```python
# Initialize schema if needed
cursor = await conn.execute("SELECT name FROM sqlite_master WHERE type='table' AND name='workflows'")
table_exists = await cursor.fetchone()
await cursor.close()
if not table_exists:
    logger.info("Database schema not found. Initializing...", emoji_key="gear")
    await conn.execute("PRAGMA foreign_keys = ON;")
    await conn.executescript(SCHEMA_SQL)
    logger.success("Database schema initialized successfully.", emoji_key="white_check_mark")
```

The schema includes several critical components:

Base Tables

1. `workflows`: The top-level container

   ```sql
   CREATE TABLE IF NOT EXISTS workflows (
       workflow_id TEXT PRIMARY KEY,
       title TEXT NOT NULL,
       description TEXT,
       goal TEXT,
       status TEXT NOT NULL,
       created_at INTEGER NOT NULL,
       updated_at INTEGER NOT NULL,
       completed_at INTEGER,
       parent_workflow_id TEXT,
       metadata TEXT,
       last_active INTEGER
   );
   ```

2. `actions`: Records of agent activities

   ```sql
   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,
       title TEXT,
       reasoning TEXT,
       tool_name TEXT,
       tool_args TEXT,
       tool_result TEXT,
       status TEXT NOT NULL,
       started_at INTEGER NOT NULL,
       completed_at INTEGER,
       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
   );
   ```

3. `artifacts`: Outputs and files created during workflows

   ```sql
   CREATE TABLE IF NOT EXISTS artifacts (
       artifact_id TEXT PRIMARY KEY,
       workflow_id TEXT NOT NULL,
       action_id TEXT,
       artifact_type TEXT NOT NULL,
       name TEXT NOT NULL,
   ```

```
    description TEXT,
    path TEXT,
    content TEXT,
    metadata TEXT,
    created_at INTEGER 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
);
```

4. `memories`: Core memory storage

```
CREATE TABLE IF NOT EXISTS memories (
    memory_id TEXT PRIMARY KEY,
    workflow_id TEXT NOT NULL,
    content TEXT NOT NULL,
    memory_level TEXT NOT NULL,
    memory_type TEXT NOT NULL,
    importance REAL DEFAULT 5.0,
    confidence REAL DEFAULT 1.0,
    description TEXT,
    reasoning TEXT,
    source TEXT,
    context TEXT,
    tags TEXT,
    created_at INTEGER NOT NULL,
    updated_at INTEGER NOT NULL,
    last_accessed INTEGER,
    access_count INTEGER DEFAULT 0,
    ttl INTEGER DEFAULT 0,
    embedding_id TEXT,
    action_id TEXT,
    thought_id TEXT,
    artifact_id TEXT,
    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,
    FOREIGN KEY (artifact_id) REFERENCES artifacts(artifact_id) ON DELETE SET NULL
);
```

5. `thought_chains` and `thoughts`: Reasoning structure

```
CREATE TABLE IF NOT EXISTS thought_chains (
    thought_chain_id TEXT PRIMARY KEY,
    workflow_id TEXT NOT NULL,
    action_id TEXT,
    title TEXT NOT NULL,
    created_at INTEGER 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
);

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,
    content TEXT NOT NULL,
    sequence_number INTEGER NOT NULL,
    created_at INTEGER NOT NULL,
    relevant_action_id TEXT,
    relevant_artifact_id TEXT,
    relevant_memory_id TEXT,
    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
);
```

Advanced Features

1. **Circular Foreign Key Constraints**: The schema implements circular references between memories and thoughts using deferred constraints:

```
-- Deferrable Circular Foreign Key Constraints for thoughts <-> memories
BEGIN IMMEDIATE TRANSACTION;
PRAGMA defer_foreign_keys = ON;

ALTER TABLE thoughts ADD CONSTRAINT fk_thoughts_memory
    FOREIGN KEY (relevant_memory_id) REFERENCES memories(memory_id)
    ON DELETE SET NULL DEFERRABLE INITIALLY DEFERRED;

ALTER TABLE memories ADD CONSTRAINT fk_memories_thought
    FOREIGN KEY (thought_id) REFERENCES thoughts(thought_id)
    ON DELETE SET NULL DEFERRABLE INITIALLY DEFERRED;

COMMIT;
```

This pattern allows creating memories that reference thoughts and thoughts that reference memories, resolving the chicken-and-egg problem typically encountered with circular foreign keys.

2. **Full-Text Search**: The system implements sophisticated text search through SQLite's FTS5 virtual table:

```
CREATE VIRTUAL TABLE IF NOT EXISTS memory_fts USING fts5(
    content, description, reasoning, tags,
    workflow_id UNINDEXED,
    memory_id UNINDEXED,
    content='memories',
    content_rowid='rowid',
    tokenize='porter unicode61'
);
```

With synchronized triggers:

```
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;
```

3. **Vector Embeddings**: The schema includes an 'embeddings' table for storing vector representations:

```
CREATE TABLE IF NOT EXISTS embeddings (
    id TEXT PRIMARY KEY,
    memory_id TEXT UNIQUE,
    model TEXT NOT NULL,
    embedding BLOB NOT NULL,
    dimension INTEGER NOT NULL,
    created_at INTEGER NOT NULL
);
```

With a back-reference from embeddings to memories:

```
ALTER TABLE embeddings ADD CONSTRAINT fk_embeddings_memory FOREIGN KEY (memory_id) REFERENCES
memories(memory_id) ON DELETE CASCADE;
```

4. **Memory Links**: Associative connections between memories:

```
CREATE TABLE IF NOT EXISTS memory_links (
    link_id TEXT PRIMARY KEY,
    source_memory_id TEXT NOT NULL,
    target_memory_id TEXT NOT NULL,
    link_type TEXT NOT NULL,
    strength REAL DEFAULT 1.0,
    description TEXT,
    created_at INTEGER NOT NULL,
    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)
);
```

5. **Cognitive States**: Persistence of cognitive context:

```sql
CREATE TABLE IF NOT EXISTS cognitive_states (
    state_id TEXT PRIMARY KEY,
    workflow_id TEXT NOT NULL,
    title TEXT NOT NULL,
    working_memory TEXT,
    focus_areas TEXT,
    context_actions TEXT,
    current_goals TEXT,
    created_at INTEGER NOT NULL,
    is_latest BOOLEAN NOT NULL,
    FOREIGN KEY (workflow_id) REFERENCES workflows(workflow_id) ON DELETE CASCADE
);
```

6. **Meta-Cognitive Components**:

```sql
CREATE TABLE IF NOT EXISTS reflections (
    reflection_id TEXT PRIMARY KEY,
    workflow_id TEXT NOT NULL,
    title TEXT NOT NULL,
    content TEXT NOT NULL,
    reflection_type TEXT NOT NULL,
    created_at INTEGER NOT NULL,
    referenced_memories TEXT,
    FOREIGN KEY (workflow_id) REFERENCES workflows(workflow_id) ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS memory_operations (
    operation_log_id TEXT PRIMARY KEY,
    workflow_id TEXT NOT NULL,
    memory_id TEXT,
    action_id TEXT,
    operation TEXT NOT NULL,
    operation_data TEXT,
    timestamp INTEGER NOT NULL,
    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
);
```

7. **Tagging System**: Comprehensive tagging with junction tables:

```sql
CREATE TABLE IF NOT EXISTS tags (
    tag_id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL UNIQUE,
    description TEXT,
    category TEXT,
    created_at INTEGER NOT NULL
);

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
);
```

   With similar structures for `action_tags` and `artifact_tags`.

8. **Dependencies**: Structured action dependencies:

```sql
CREATE TABLE IF NOT EXISTS dependencies (
    dependency_id INTEGER PRIMARY KEY AUTOINCREMENT,
    source_action_id TEXT NOT NULL,
    target_action_id TEXT NOT NULL,
    dependency_type TEXT NOT NULL,
    created_at INTEGER NOT NULL,
    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)
);
```

Schema Optimization

The schema includes comprehensive indexing for performance optimization:

```sql
-- 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);
-- 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);
```

With over 25 carefully designed indices covering most query patterns. Foreign keys are indexed as well as search fields, and compound indices are used for common query patterns.

Custom SQLite Functions

The system extends SQLite with custom functions for advanced querying capabilities:

```python
await conn.create_function("json_contains", 2, _json_contains, deterministic=True)
await conn.create_function("json_contains_any", 2, _json_contains_any, deterministic=True)
await conn.create_function("json_contains_all", 2, _json_contains_all, deterministic=True)
await conn.create_function("compute_memory_relevance", 5, _compute_memory_relevance, deterministic=True)
```

These functions enable:

1. **JSON Array Operations**:

```python
def _json_contains(json_text, search_value):
    """Check if a JSON array contains a specific 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
```

   With similar functions for checking if any or all values from a list are present in a JSON array.

2. **Memory Relevance Calculation**:

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

   This function is central to memory prioritization, implementing:

   - Time-based decay of importance
   - Recency boost for recently accessed memories
   - Usage frequency boost
   - Confidence weighting
   - Bounded output range (0.0-10.0)

Error Handling and Decorators

The system implements consistent error handling through decorators:

```
@with_tool_metrics
@with_error_handling
async def function_name(...):
    # Implementation
```

These decorators provide:

1. **Error Standardization**:
   - `ToolInputError`: For invalid parameters
   - `ToolError`: For operational/system failures
   - Robust exception conversion and logging
2. **Performance Metrics**:
   - Timing for each operation
   - Success/failure tracking
   - Consistent result formatting
3. **Logging Integration**:
   - Standardized log format with emojis
   - Differentiated log levels (info, warning, error)
   - Performance timing included

The pattern ensures all tool functions have consistent behavior:

```
# Example decorator patterns:
def with_error_handling(func):
    """Wrapper for standardized error handling in tool functions."""
    @functools.wraps(func)
    async def wrapper(*args, **kwargs):
        try:
            return await func(*args, **kwargs)
        except ToolInputError:
            # Re-raise with input validation errors
            raise
        except Exception as e:
            # Convert other exceptions to ToolError
            logger.error(f"Error in {func.__name__}: {e}", exc_info=True)
            raise ToolError(f"Operation failed: {str(e)}") from e
    return wrapper

def with_tool_metrics(func):
    """Wrapper for tracking metrics and standardizing tool function results."""
    @functools.wraps(func)
    async def wrapper(*args, **kwargs):
        start_time = time.time()
        result = await func(*args, **kwargs)
        processing_time = time.time() - start_time

        # Add standardized fields if result is a dict
        if isinstance(result, dict):
            result["success"] = True
            result["processing_time"] = processing_time

        logger.info(f"{func.__name__} completed in {processing_time:.3f}s")
        return result
    return wrapper
```

Transaction Management

The system implements sophisticated transaction management through a context manager:

```
@contextlib.asynccontextmanager
async def transaction(self) -> AsyncIterator[aiosqlite.Connection]:
    """Provides an atomic transaction block using the singleton connection."""
    conn = await self.__aenter__()  # Acquire the connection instance
    try:
        await conn.execute("BEGIN DEFERRED TRANSACTION")
        logger.debug("DB Transaction Started.")
        yield conn  # Provide the connection to the 'async with' block
    except Exception as e:
        logger.error(f"Exception during transaction, rolling back: {e}", exc_info=True)
        await conn.rollback()
        logger.warning("DB Transaction Rolled Back.", emoji_key="rewind")
        raise  # Re-raise the exception after rollback
    else:
        await conn.commit()
        logger.debug("DB Transaction Committed.")
```

This allows operations to be grouped atomically:

```
# Usage example
db_manager = DBConnection(db_path)
async with db_manager.transaction() as conn:
    # Multiple operations that should succeed or fail together
    await conn.execute("INSERT INTO ...")
    await conn.execute("UPDATE ...")
    # Auto-commits on success, rolls back on exception
```

The transaction manager is used extensively throughout the codebase to ensure data integrity, particularly for:

- Creating workflow and initial thought chain
- Recording actions and linked memories
- Creating thoughts with associated memory entries
- Complex dependency operations

Vector Embedding and Semantic Search Implementation

Embedding Storage

The system implements vector embedding storage:

```
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."""
    try:
        embedding_service = get_embedding_service()  # Get singleton instance
        if not embedding_service.client:
            logger.warning("EmbeddingService client not available. Cannot generate embedding.")
            return None

        # Generate embedding using the service (handles caching internally)
        embedding_list = await embedding_service.create_embeddings(texts=[text])
        if not embedding_list or not embedding_list[0]:
            logger.warning(f"Failed to generate embedding for memory {memory_id}")
            return None
        embedding_array = np.array(embedding_list[0], dtype=np.float32)
        if embedding_array.size == 0:
            logger.warning(f"Generated embedding is empty for memory {memory_id}")
            return None

        # Get the embedding dimension
        embedding_dimension = embedding_array.shape[0]

        # Generate a unique ID for this embedding entry
        embedding_db_id = MemoryUtils.generate_id()
        embedding_bytes = embedding_array.tobytes()
        model_used = embedding_service.default_model

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

        # Update memory record to link to embedding
        await conn.execute(
            "UPDATE memories SET embedding_id = ? WHERE memory_id = ?",
            (embedding_db_id, memory_id)
        )

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

Key aspects:

1. Integration with external embedding service
2. Numpy array serialization to binary BLOB
3. Dimension tracking for compatibility
4. UPSERT pattern for idempotent updates
5. Error handling for service failures

Semantic Search Implementation

```
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
) -> List[Tuple[str, float]]:
    """Finds memories with similar semantic meaning using embeddings."""
    try:
        embedding_service = get_embedding_service()
        if not embedding_service.client:
            logger.warning("EmbeddingService client not available.")
            return []

        # 1. Generate query embedding
        query_embedding_list = await embedding_service.create_embeddings(texts=[query_text])
        if not query_embedding_list or not query_embedding_list[0]:
            logger.warning(f"Failed to generate query embedding")
            return []
        query_embedding = np.array(query_embedding_list[0], dtype=np.float32)
        query_dimension = query_embedding.shape[0]
        query_embedding_2d = query_embedding.reshape(1, -1)

        # 2. Build query for candidate embeddings with filters
        sql = """
        SELECT m.memory_id, e.embedding
        FROM memories m
        JOIN embeddings e ON m.embedding_id = e.id
        WHERE e.dimension = ?
        """
        params: List[Any] = [query_dimension]

        # Add filters
        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())
        if memory_type:
            sql += " AND m.memory_type = ?"
            params.append(memory_type.lower())

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

        # Optimize with pre-filtering and candidate limit
        candidate_limit = max(limit * 5, 50)
        sql += " ORDER BY m.last_accessed DESC NULLS LAST LIMIT ?"
        params.append(candidate_limit)

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

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

        # 4. Calculate similarities using scikit-learn
        similarities: List[Tuple[str, float]] = []
        for memory_id, embedding_bytes in candidates:
            try:
                # Deserialize embedding from bytes
                memory_embedding = np.frombuffer(embedding_bytes, dtype=np.float32)
                if memory_embedding.size == 0:
                    continue

                memory_embedding_2d = memory_embedding.reshape(1, -1)

                # Safety check for dimension mismatch
                if query_embedding_2d.shape[1] != memory_embedding_2d.shape[1]:
                    continue

                # Calculate cosine similarity
                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 (descending) and limit
        similarities.sort(key=lambda x: x[1], reverse=True)
        return similarities[:limit]

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

Key aspects:

1. Integration with embedding service API
2. Efficient querying with dimension matching
3. Candidate pre-filtering before similarity calculation
4. Serialized binary embedding handling
5. Scikit-learn integration for cosine similarity
6. Threshold filtering and result ranking
7. Comprehensive error handling for edge cases

Mermaid Diagram Generation

The system generates sophisticated visualization diagrams:

Workflow Diagram Generation

```python
async def _generate_mermaid_diagram(workflow: Dict[str, Any]) -> str:
    """Generates a detailed Mermaid flowchart representation of the workflow."""

    def sanitize_mermaid_id(uuid_str: Optional[str], prefix: str) -> str:
        """Creates a valid Mermaid node ID from a UUID, handling None."""
        if not uuid_str:
             return f"{prefix}_MISSING_{MemoryUtils.generate_id().replace('-', '_')}"
        sanitized = uuid_str.replace("-", "_")  # Hyphens cause issues in Mermaid
        return f"{prefix}_{sanitized}"

    diagram = ["```mermaid", "flowchart TD"]  # Top-Down flowchart

    # --- Generate Workflow Node ---
    wf_node_id = sanitize_mermaid_id(workflow.get('workflow_id'), "W")
    wf_title = _mermaid_escape(workflow.get('title', 'Workflow'))
    wf_status_class = f":::{workflow.get('status', 'active')}"
    diagram.append(f'    {wf_node_id}("{wf_title}"){wf_status_class}')

    # --- Generate Action Nodes ---
    action_nodes = {}  # Map action_id to mermaid_node_id
    parent_links = {}  # Map child_action_id to parent_action_id
    sequential_links = {}  # Map sequence_number to action_id

    for action in sorted(workflow.get("actions", []), key=lambda a: a.get("sequence_number", 0)):
        action_id = action.get("action_id")
        if not action_id:
            continue

        node_id = sanitize_mermaid_id(action_id, "A")
        action_nodes[action_id] = node_id

        # Create node label with type, title, and tool info
        action_type = action.get('action_type', 'Action').capitalize()
        action_title = _mermaid_escape(action.get('title', action_type))
        sequence_number = action.get("sequence_number", 0)
        label = f"<b>{action_type} #{sequence_number}</b><br/>{action_title}"
        if action.get('tool_name'):
            label += f"<br/><i>Tool: {_mermaid_escape(action['tool_name'])}</i>"

        # Style node based on status
        status = action.get('status', ActionStatus.PLANNED.value)
        node_style = f":::{status}"

        diagram.append(f'    {node_id}["{label}"]{node_style}')

        # Record parent relationship
        parent_action_id = action.get("parent_action_id")
        if parent_action_id:
            parent_links[action_id] = parent_action_id
        else:
            sequential_links[sequence_number] = action_id

    # --- Generate Action Links ---
    linked_actions = set()

    # Parent->Child links
    for child_id, parent_id in parent_links.items():
        if child_id in action_nodes and parent_id in action_nodes:
            child_node = action_nodes[child_id]
            parent_node = action_nodes[parent_id]
            diagram.append(f"    {parent_node} --> {child_node}")
            linked_actions.add(child_id)

    # Sequential links for actions without explicit parents
    last_sequential_node = wf_node_id
    for seq_num in sorted(sequential_links.keys()):
        action_id = sequential_links[seq_num]
        if action_id in action_nodes:
             node_id = action_nodes[action_id]
             diagram.append(f"    {last_sequential_node} --> {node_id}")
             last_sequential_node = node_id
             linked_actions.add(action_id)

    # --- Generate Artifact Nodes ---
    for artifact in workflow.get("artifacts", []):
```

```
        artifact_id = artifact.get("artifact_id")
        if not artifact_id:
            continue

        node_id = sanitize_mermaid_id(artifact_id, "F")
        artifact_name = _mermaid_escape(artifact.get('name', 'Artifact'))
        artifact_type = _mermaid_escape(artifact.get('artifact_type', 'file'))
        label = f" <br/><b>{artifact_name}</b><br/>({artifact_type})"

        node_shape_start, node_shape_end = "[(", ")]"  # Database/capsule shape
        node_style = ":::artifact"
        if artifact.get('is_output'):
            node_style = ":::artifact_output"  # Special style for outputs

        diagram.append(f'    {node_id}{node_shape_start}"{label}"{node_shape_end}{node_style}')

        # Link from creating action
        creator_action_id = artifact.get("action_id")
        if creator_action_id and creator_action_id in action_nodes:
            creator_node = action_nodes[creator_action_id]
            diagram.append(f"    {creator_node} -- Creates --> {node_id}")
        else:
            # Link to workflow if no specific action
            diagram.append(f"    {wf_node_id} -.-> {node_id}")

    # --- Add Class Definitions for Styling ---
    diagram.append("\n    %% Stylesheets")
    diagram.append("    classDef workflow fill:#e7f0fd,stroke:#0056b3,stroke-width:2px,color:#000")
    diagram.append("    classDef completed fill:#d4edda,stroke:#155724,stroke-width:1px,color:#155724")
    diagram.append("    classDef failed fill:#f8d7da,stroke:#721c24,stroke-width:1px,color:#721c24")
    # ... many more style definitions ...

    diagram.append("```")
    return "\n".join(diagram)
```

This intricate function:

1. Sanitizes UUIDs for Mermaid compatibility
2. Constructs a flowchart with workflow, actions, and artifacts
3. Creates hierarchical relationships
4. Handles parent-child and sequential relationships
5. Implements detailed styling based on status
6. Escapes special characters for Mermaid compatibility

Memory Network Diagram Generation

```
async def _generate_memory_network_mermaid(memories: List[Dict], links: List[Dict], center_memory_id: Optional[str]
= None) -> str:
    """Helper function to generate Mermaid graph syntax for a memory network."""

    def sanitize_mermaid_id(uuid_str: Optional[str], prefix: str) -> str:
        """Creates a valid Mermaid node ID from a UUID, handling None."""
        if not uuid_str:
             return f"{prefix}_MISSING_{MemoryUtils.generate_id().replace('-', '_')}"
        sanitized = uuid_str.replace("-", "_")
        return f"{prefix}_{sanitized}"

    diagram = ["```mermaid", "graph TD"]  # Top-Down graph direction

    # --- Memory Node Definitions ---
    memory_id_to_node_id = {}  # Map full memory ID to sanitized Mermaid node ID
    for memory in memories:
        mem_id = memory.get("memory_id")
        if not mem_id:
            continue

        node_id = sanitize_mermaid_id(mem_id, "M")
        memory_id_to_node_id[mem_id] = node_id

        # Create node label with type, description, importance
        mem_type = memory.get("memory_type", "memory").capitalize()
        desc = _mermaid_escape(memory.get("description", mem_id))
```

```
        if len(desc) > 40:
            desc = desc[:37] + "..."
        importance = memory.get('importance', 5.0)
        label = f"<b>{mem_type}</b><br/>{desc}<br/><i>(I: {importance:.1f})</i>"

        # Choose node shape based on memory level
        level = memory.get("memory_level", MemoryLevel.EPISODIC.value)
        shape_start, shape_end = "[", "]"  # Default rectangle (Semantic)
        if level == MemoryLevel.EPISODIC.value:
            shape_start, shape_end = "(", ")"  # Round (Episodic)
        elif level == MemoryLevel.PROCEDURAL.value:
            shape_start, shape_end = "[[", "]]"  # Subroutine (Procedural)
        elif level == MemoryLevel.WORKING.value:
             shape_start, shape_end = "([", "])"  # Capsule (Working)

        # Style node based on level + highlight center
        node_style = f":::level{level}"
        if mem_id == center_memory_id:
            node_style += " :::centerNode"  # Highlight center node

        diagram.append(f'    {node_id}{shape_start}"{label}"{shape_end}{node_style}')

    # --- Memory Link Definitions ---
    for link in links:
        source_mem_id = link.get("source_memory_id")
        target_mem_id = link.get("target_memory_id")
        link_type = link.get("link_type", "related")

        # Only draw links where both ends are in the visualization
        if source_mem_id in memory_id_to_node_id and target_mem_id in memory_id_to_node_id:
            source_node = memory_id_to_node_id[source_mem_id]
            target_node = memory_id_to_node_id[target_mem_id]
            diagram.append(f"    {source_node} -- {link_type} --> {target_node}")

    # --- Add Class Definitions for Styling ---
    diagram.append("\n    %% Stylesheets")
    diagram.append("    classDef levelworking fill:#e3f2fd,stroke:#2196f3,color:#1e88e5,stroke-width:1px;")
    diagram.append("    classDef levelepisodic fill:#e8f5e9,stroke:#4caf50,color:#388e3c,stroke-width:1px;")
    # ... additional style definitions ...
    diagram.append("    classDef centerNode stroke-width:3px,stroke:#0d47a1,font-weight:bold;")

    diagram.append("```")
    return "\n".join(diagram)
```

This visualization:

1. Displays memories with level-specific shapes
2. Shows relationship types on connection lines
3. Provides visual cues for importance and type
4. Highlights the center node when specified
5. Implements sophisticated styling based on memory levels

Character Escaping for Mermaid

The system implements robust character escaping for Mermaid compatibility:

```
def _mermaid_escape(text: str) -> str:
    """Escapes characters problematic for Mermaid node labels."""
    if not isinstance(text, str):
        text = str(text)
    # Replace quotes first, then other potentially problematic characters
    text = text.replace('"', '#quot;')
    text = text.replace('(', '#40;')
    text = text.replace(')', '#41;')
    text = text.replace('[', '#91;')
    text = text.replace(']', '#93;')
    text = text.replace('{', '#123;')
    text = text.replace('}', '#125;')
    text = text.replace(':', '#58;')
    text = text.replace(';', '#59;')
    text = text.replace('<', '#lt;')
    text = text.replace('>', '#gt;')
```

```
    # Replace newline with <br> for multiline labels
    text = text.replace('\n', '<br>')
    return text
```

This function handles all special characters that could break Mermaid diagram syntax.

Serialization and Data Handling

The system implements sophisticated serialization with robust error handling:

```
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

    json_str = None

    try:
        # Attempt direct JSON serialization
        json_str = json.dumps(obj, ensure_ascii=False, default=str)

    except TypeError as e:
        # Handle objects that are not directly serializable
        logger.debug(f"Direct JSON serialization failed for type {type(obj)}: {e}")
        try:
            # Fallback using string representation
            fallback_repr = str(obj)
            fallback_bytes = fallback_repr.encode('utf-8')

            if len(fallback_bytes) > MAX_TEXT_LENGTH:
                # Truncate if too large
                truncated_bytes = fallback_bytes[:MAX_TEXT_LENGTH]
                truncated_repr = truncated_bytes.decode('utf-8', errors='replace')

                # Advanced handling for multi-byte character truncation
                if truncated_repr.endswith('\ufffd') and MAX_TEXT_LENGTH > 1:
                     shorter_repr = fallback_bytes[:MAX_TEXT_LENGTH-1].decode('utf-8', errors='replace')
                     if not shorter_repr.endswith('\ufffd'):
                          truncated_repr = shorter_repr

                truncated_repr += "[TRUNCATED]"
                logger.warning(f"Fallback string representation truncated for type {type(obj)}.")
            else:
                truncated_repr = fallback_repr

            # Create structured representation of the error
            json_str = json.dumps({
                "error": f"Serialization failed for type {type(obj)}.",
                "fallback_repr": truncated_repr
            }, ensure_ascii=False)

        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}")
            json_str = json.dumps({
                "error": f"Unserializable object type {type(obj)}. Fallback failed.",
                "critical_error": str(fallback_e)
            }, ensure_ascii=False)

    # Check final length regardless of serialization path
    if json_str is None:
         logger.error(f"Internal error: json_str is None after serialization attempt for object of type
         {type(obj)}")
         return json.dumps({
             "error": "Internal serialization error occurred.",
             "original_type": str(type(obj))
         }, ensure_ascii=False)
    # Check if final result exceeds max length
    final_bytes = json_str.encode('utf-8')
    if len(final_bytes) > MAX_TEXT_LENGTH:
        logger.warning(f"Serialized JSON string exceeds max length ({MAX_TEXT_LENGTH} bytes)")
        preview_str = json_str[:200] + ("..." if len(json_str) > 200 else "")
        return json.dumps({
            "error": "Serialized content exceeded maximum length.",
            "original_type": str(type(obj)),
            "preview": preview_str
        }, ensure_ascii=False)
    else:
        return json_str
```

This highly sophisticated serialization function:

1. Handles arbitrary Python objects
2. Implements multiple fallback strategies
3. Properly handles UTF-8 encoding and truncation
4. Preserves information about serialization failures
5. Returns structured error information
6. Enforces maximum content length limits

LLM Prompt Templates for Meta-Cognition

The system uses sophisticated prompt templates for LLM-based reflection:

Consolidation Prompts

```
def _generate_consolidation_prompt(memories: List[Dict], consolidation_type: str) -> str:
    """Generates a prompt for memory consolidation."""
    # Format memories with metadata
    memory_texts = []
    for i, memory in enumerate(memories[:20], 1):
        desc = memory.get("description") or ""
        content_preview = (memory.get("content", "") or "")[:300]
        mem_type = memory.get("memory_type", "N/A")
        importance = memory.get("importance", 5.0)
        confidence = memory.get("confidence", 1.0)
        created_ts = memory.get("created_at", 0)
        created_dt_str = datetime.fromtimestamp(created_ts).strftime('%Y-%m-%d %H:%M') if created_ts else "Unknown
        Date"
        mem_id_short = memory.get("memory_id", "UNKNOWN")[:8]

        formatted = f"--- MEMORY #{i} (ID: {mem_id_short}..., Type: {mem_type}, Importance: {importance:.1f},
        Confidence: {confidence:.1f}, Date: {created_dt_str}) ---\n"
        if desc:
            formatted += f"Description: {desc}\n"
        formatted += f"Content Preview: {content_preview}"
        # Indicate truncation
        if len(memory.get("content", "")) > 300:
            formatted += "...\n"
        else:
            formatted += "\n"
        memory_texts.append(formatted)

    memories_str = "\n".join(memory_texts)

    # Base prompt template
    base_prompt = f"""You are an advanced cognitive system processing and consolidating memories for an AI agent.
    Below are {len(memories)} memory items containing information, observations, and insights relevant to a task.
    Your goal is to perform a specific type of consolidation: '{consolidation_type}'.

Analyze the following memories carefully:

{memories_str}
--- END OF MEMORIES ---
"""

    # Add type-specific instructions
    if consolidation_type == "summary":
        base_prompt += """TASK: Create a comprehensive and coherent summary...
        [detailed instructions for summarization]
        """
    elif consolidation_type == "insight":
        base_prompt += """TASK: Generate high-level insights...
        [detailed instructions for insight generation]
        """
    # Additional consolidation types...

    return base_prompt
```

Reflection Prompts

```
def _generate_reflection_prompt(
    workflow_name: str,
    workflow_desc: Optional[str],
    operations: List[Dict],
    memories: Dict[str, Dict],
    reflection_type: str
) -> str:
    """Generates a prompt for reflective analysis."""
    # Format operations with context
    op_texts = []
    for i, op_data in enumerate(operations[:30], 1):
        op_ts_unix = op_data.get("timestamp", 0)
        op_ts_str = datetime.fromtimestamp(op_ts_unix).strftime('%Y-%m-%d %H:%M:%S') if op_ts_unix else "Unknown
        Time"
        op_type = op_data.get('operation', 'UNKNOWN').upper()
        mem_id = op_data.get('memory_id')
        action_id = op_data.get('action_id')

        # Extract operation details
        op_details_dict = {}
        op_data_raw = op_data.get('operation_data')
        if op_data_raw:
             try:
                  op_details_dict = json.loads(op_data_raw)
             except (json.JSONDecodeError, TypeError):
                  op_details_dict = {"raw_data": str(op_data_raw)[:50]}

        # Build rich description
        desc_parts = [f"OP #{i} ({op_ts_str})", f"Type: {op_type}"]
        if mem_id:
            mem_info = memories.get(mem_id)
            mem_desc_text = f"Mem({mem_id[:6]}..)"
            if mem_info:
                 mem_desc_text += f" Desc: {mem_info.get('description', 'N/A')[:40]}"
                 if mem_info.get('memory_type'):
                      mem_desc_text += f" Type: {mem_info['memory_type']}"
            desc_parts.append(mem_desc_text)

        if action_id:
            desc_parts.append(f"Action({action_id[:6]}..)")

        # Add operation data details
        detail_items = []
        for k, v in op_details_dict.items():
             if k not in ['content', 'description', 'embedding', 'prompt']:
                  detail_items.append(f"{k}={str(v)[:30]}")
        if detail_items:
            desc_parts.append(f"Data({', '.join(detail_items)})")

        op_texts.append(" | ".join(desc_parts))

    operations_str = "\n".join(op_texts)

    # Base prompt template
    base_prompt = f"""You are an advanced meta-cognitive system analyzing an AI agent's workflow: "{workflow_name}".
Workflow Description: {workflow_desc or 'N/A'}
Your task is to perform a '{reflection_type}' reflection based on the recent memory operations listed below. Analyze
these operations to understand the agent's process, progress, and knowledge state.

RECENT OPERATIONS (Up to 30):
{operations_str}
"""
```

```
    # Add type-specific instructions
    if reflection_type == "summary":
        base_prompt += """TASK: Create a reflective summary...
        [detailed instructions for reflective summarization]
        """
    elif reflection_type == "progress":
        base_prompt += """TASK: Analyze the progress...
        [detailed instructions for progress analysis]
        """
    # Additional reflection types...

    return base_prompt
```

These templates implement:

1. Rich context formatting with metadata
2. Type-specific detailed instructions
3. Structured memory representation
4. Operation history formatting with context
5. Guidance tailored to different meta-cognitive tasks

Integration Patterns for Complex Operations

The system implements several integration patterns for complex operations:

Workflow Creation with Initial Thought

```
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."""
    # Validation and initialization...

    try:
        async with DBConnection(db_path) as conn:
            # Check parent workflow existence...

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

            # Insert the main workflow record
            await conn.execute("""INSERT INTO workflows...""")

            # Process and associate tags
            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}"
            await conn.execute("""INSERT INTO thought_chains...""")

            # If a goal was provided, add it as the first thought in the default chain
            if goal:
                thought_id = MemoryUtils.generate_id()
                seq_no = await MemoryUtils.get_next_sequence_number(conn, thought_chain_id, "thoughts",
                "thought_chain_id")
                await conn.execute("""INSERT INTO thoughts...""")

            # Commit the transaction
            await conn.commit()

            # Prepare and return result
            # ...
    except ToolInputError:
        raise
    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
```

This pattern:

1. Creates multiple related objects in one transaction
2. Establishes default chain for reasoning
3. Optionally adds initial thought/goal
4. Ensures atomicity through transaction management

Action Recording with Episodic Memory

```
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."""
    # Validation and initialization...

    try:
        async with DBConnection(db_path) as conn:
            # Existence checks...

            # Determine sequence and auto-title...

            # Insert action record
            tool_args_json = await MemoryUtils.serialize(tool_args)
            await conn.execute("""INSERT INTO actions...""")

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

            # Link 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
            memory_id = MemoryUtils.generate_id()
            memory_content = f"Started action [{sequence_number}] '{auto_title}' ({action_type_enum.value}).
            Reasoning: {reasoning}"
            if tool_name:
                 memory_content += f" Tool: {tool_name}."
            mem_tags = ["action_start", action_type_enum.value] + (tags or [])
            mem_tags_json = json.dumps(list(set(mem_tags)))

            await conn.execute("""INSERT INTO memories...""")
            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_unix, now_unix, workflow_id))

            # Commit transaction
            await conn.commit()

            # Prepare and return result
            # ...
    except ToolInputError:
        raise
    except Exception as e:
        logger.error(f"Error recording action start: {e}", exc_info=True)
        raise ToolError(f"Failed to record action start: {str(e)}") from e
```

This pattern:

1. Records action details
2. Automatically creates linked episodic memory
3. Updates related entities (thoughts, workflow)
4. Maintains bidirectional references
5. Ensures proper tagging and categorization

Thought Recording with Optional Memory Creation

```python
async def record_thought(
    workflow_id: str,
    content: str,
    thought_type: str = "inference",
    thought_chain_id: Optional[str] = None,
    parent_thought_id: Optional[str] = None,
    relevant_action_id: Optional[str] = None,
    relevant_artifact_id: Optional[str] = None,
    relevant_memory_id: Optional[str] = None,
    db_path: str = DEFAULT_DB_PATH,
    conn: Optional[aiosqlite.Connection] = None
) -> Dict[str, Any]:
    """Records a thought in a reasoning chain, potentially linking to memory and creating an associated memory
    entry."""
    # Validation...

    thought_id = MemoryUtils.generate_id()
    now_unix = int(time.time())
    linked_memory_id = None

    async def _perform_db_operations(db_conn: aiosqlite.Connection):
        """Inner function to perform DB ops using the provided connection."""
        nonlocal linked_memory_id

        # Existence checks...

        # Determine target thought chain...

        # Get sequence number...

        # Insert thought record...

        # Update workflow timestamp...

        # Create linked memory for important thoughts
        important_thought_types = [
            ThoughtType.GOAL.value, ThoughtType.DECISION.value, ThoughtType.SUMMARY.value,
            ThoughtType.REFLECTION.value, ThoughtType.HYPOTHESIS.value
        ]

        if thought_type_enum.value in important_thought_types:
            linked_memory_id = MemoryUtils.generate_id()
            mem_content = f"Thought [{sequence_number}] ({thought_type_enum.value.capitalize()}): {content}"
            mem_tags = ["reasoning", thought_type_enum.value]
            mem_importance = 7.5 if thought_type_enum.value in [ThoughtType.GOAL.value, ThoughtType.DECISION.value]
            else 6.5

            await db_conn.execute("""INSERT INTO memories...""")
            await MemoryUtils._log_memory_operation(db_conn, workflow_id, "create_from_thought", linked_memory_id,
            None)

        return target_thought_chain_id, sequence_number

    try:
        target_thought_chain_id_res = None
        sequence_number_res = None

        if conn:
            # Use provided connection (transaction nesting)
            target_thought_chain_id_res, sequence_number_res = await _perform_db_operations(conn)
            # No commit - handled by outer transaction
        else:
            # Manage local transaction
            db_manager = DBConnection(db_path)
            async with db_manager.transaction() as local_conn:
                target_thought_chain_id_res, sequence_number_res = await _perform_db_operations(local_conn)
            # Commit handled by transaction manager

        # Prepare and return result
        # ...
    except ToolInputError:
        raise
    except Exception as e:
        logger.error(f"Error recording thought: {e}", exc_info=True)
        raise ToolError(f"Failed to record thought: {str(e)}") from e
```

This pattern:

1. Supports transaction nesting via optional connection parameter
2. Conditionally creates memory entries for important thoughts
3. Implements comprehensive linking between entities
4. Uses inner functions for encapsulation
5. Determines correct thought chain automatically

Memory Consolidation with Linking

```
async def consolidate_memories(
    workflow_id: Optional[str] = None,
    target_memories: Optional[List[str]] = None,
    consolidation_type: str = "summary",
    query_filter: Optional[Dict[str, Any]] = None,
    max_source_memories: int = 20,
    prompt_override: Optional[str] = None,
    provider: str = LLMGatewayProvider.OPENAI.value,
    model: Optional[str] = None,
    store_result: bool = True,
    store_as_level: str = MemoryLevel.SEMANTIC.value,
    store_as_type: Optional[str] = None,
    max_tokens: int = 1000,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
    """Consolidates multiple memories using an LLM to generate summaries, insights, etc."""
    # Validation...

    source_memories_list = []
    source_memory_ids = []
    effective_workflow_id = workflow_id

    try:
        async with DBConnection(db_path) as conn:
            # Select source memories (full logic)...

            # Generate consolidation prompt...

            # Call LLM via Gateway...
            provider_instance = await get_provider(provider)
            llm_result = await provider_instance.generate_completion(
                prompt=prompt, model=final_model, max_tokens=max_tokens, temperature=0.6
            )
            consolidated_content = llm_result.text.strip()

            # Store result as new memory...
            if store_result and consolidated_content:
                # Use derived importance and confidence...
                derived_importance = min(max(source_importances) + 0.5, 10.0)
                derived_confidence = min(sum(source_confidences) / len(source_confidences), 1.0)
                derived_confidence *= (1.0 - min(0.2, (len(source_memories_list) - 1) * 0.02))

                # Store the new memory...
                store_result_dict = await store_memory(
                    workflow_id=effective_workflow_id,
                    content=consolidated_content,
                    memory_type=result_type.value,
                    memory_level=result_level.value,
                    importance=round(derived_importance, 2),
                    confidence=round(derived_confidence, 3),
                    description=result_desc,
                    source=f"consolidation_{consolidation_type}",
                    tags=result_tags, context_data=result_context,
                    generate_embedding=True, db_path=db_path
                )
                stored_memory_id = store_result_dict.get("memory_id")

                # Link result to sources...
                if stored_memory_id:
                    link_tasks = []
                    for source_id in source_memory_ids:
                         link_task = create_memory_link(
                             source_memory_id=stored_memory_id,
                             target_memory_id=source_id,
                             link_type=LinkType.GENERALIZES.value,
                             description=f"Source for consolidated {consolidation_type}",
                             db_path=db_path
                         )
                         link_tasks.append(link_task)
                    await asyncio.gather(*link_tasks, return_exceptions=True)

            # Log operation...

            # Commit...

            # Prepare and return result...
    except (ToolInputError, ToolError):
        raise
    except Exception as e:
        logger.error(f"Failed to consolidate memories: {str(e)}", exc_info=True)
        raise ToolError(f"Failed to consolidate memories: {str(e)}") from e
```

This pattern:

1. Integrates with external LLM services
2. Implements sophisticated source memory selection
3. Derives importance and confidence heuristically
4. Creates bidirectional links to source memories
5. Uses asynchronous link creation with gather

Hybrid Search with Weighted Scoring

```
async def hybrid_search_memories(
    query: str,
    workflow_id: Optional[str] = None,
    limit: int = 10,
    offset: int = 0,
    semantic_weight: float = 0.6,
    keyword_weight: float = 0.4,
    # Additional parameters...
) -> Dict[str, Any]:
    """Performs a hybrid search combining semantic similarity and keyword/filtered relevance."""
    # Validation...

    try:
        async with DBConnection(db_path) as conn:
            # --- Step 1: Semantic Search ---
            semantic_results: List[Tuple[str, float]] = []
            if norm_sem_weight > 0:
                try:
                    semantic_candidate_limit = min(max(limit * 5, 50), MAX_SEMANTIC_CANDIDATES)
                    semantic_results = await _find_similar_memories(
                        conn=conn,
                        query_text=query,
                        workflow_id=workflow_id,
                        limit=semantic_candidate_limit,
                        threshold=0.1,  # Lower threshold for hybrid
                        memory_level=memory_level,
                        memory_type=memory_type
                    )
                    for mem_id, score in semantic_results:
                        combined_scores[mem_id]["semantic"] = score
                except Exception as sem_err:
                    logger.warning(f"Semantic search part failed in hybrid search: {sem_err}")

            # --- Step 2: Keyword/Filtered Search ---
            if norm_key_weight > 0:
                # Build query with filters...
                # Execute query...
                # Calculate raw scores...
                # Normalize keyword scores...
                for mem_id, raw_score in raw_keyword_scores.items():
                    normalized_kw_score = min(max(raw_score / normalization_factor, 0.0), 1.0)
                    combined_scores[mem_id]["keyword"] = normalized_kw_score

            # --- Step 3: Calculate Hybrid Score ---
            if combined_scores:
                for _mem_id, scores in combined_scores.items():
                    scores["hybrid"] = (scores["semantic"] * norm_sem_weight) + (scores["keyword"] *
                    norm_key_weight)

                # Sort by hybrid score
                sorted_ids_scores = sorted(combined_scores.items(), key=lambda item: item[1]["hybrid"],
                reverse=True)

                # Apply pagination after ranking
                paginated_ids_scores = sorted_ids_scores[offset : offset + limit]
                final_ranked_ids = [item[0] for item in paginated_ids_scores]
                final_scores_map = {item[0]: item[1] for item in paginated_ids_scores}

            # --- Step 4-7: Fetch details, links, reconstruct results, update access ---
            # ...

            # Return final results...
    except ToolInputError:
        raise
    except Exception as e:
        logger.error(f"Hybrid search failed: {str(e)}", emoji_key="x", exc_info=True)
        raise ToolError(f"Hybrid search failed: {str(e)}") from e
```

This pattern:

1. Combines vector similarity and keyword search
2. Implements weighted scoring with normalization
3. Applies filters and pagination efficiently
4. Handles score normalization for different ranges
5. Optimizes database access with batched operations

System Initialization and Configuration

The system includes comprehensive initialization:

```
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."""
    start_time = time.time()
    logger.info("Initializing Unified Memory System...", emoji_key="rocket")
    embedding_service_warning = None

    try:
        # Initialize/Verify Database Schema
        async with DBConnection(db_path) as conn:
             # Test connection with simple query
            cursor = await conn.execute("SELECT count(*) FROM workflows")
            _ = await cursor.fetchone()
            await cursor.close()
        logger.success("Unified Memory System database connection verified.", emoji_key="database")

        # Verify EmbeddingService functionality
        try:
            embedding_service = get_embedding_service()
            if embedding_service.client is not None:
                logger.info("EmbeddingService initialized and functional.", emoji_key="brain")
            else:
                embedding_service_warning = "EmbeddingService client not available. Embeddings disabled."
                logger.error(embedding_service_warning, emoji_key="warning")
                raise ToolError(embedding_service_warning)
        except Exception as embed_init_err:
             if not isinstance(embed_init_err, ToolError):
                 embedding_service_warning = f"Failed to initialize EmbeddingService: {str(embed_init_err)}"
                 logger.error(embedding_service_warning, emoji_key="error", exc_info=True)
                 raise ToolError(embedding_service_warning) from embed_init_err
             else:
                 raise embed_init_err

        # Return success status
        processing_time = time.time() - start_time
        logger.success("Unified Memory System 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": True,
            "embedding_service_warning": None,
            "processing_time": processing_time
        }
    except Exception as e:
        processing_time = time.time() - start_time
        logger.error(f"Failed to initialize memory system: {str(e)}", emoji_key="x", exc_info=True,
        time=processing_time)
        if isinstance(e, ToolError):
            raise e
        else:
            raise ToolError(f"Memory system initialization failed: {str(e)}") from e
```

This initialization:

1. Verifies database connection and schema
2. Checks embedding service functionality
3. Provides detailed diagnostics
4. Implements robust error handling
5. Returns comprehensive status information

System Architecture Summary

The Unified Agent Memory and Cognitive System represents a sophisticated architecture for LLM agent cognitive modeling and workflow tracking. Its key architectural components include:

1. **Multi-Level Memory Hierarchy**:
    - Working memory for active processing
    - Episodic memory for experiences and events
    - Semantic memory for knowledge and facts
    - Procedural memory for skills and procedures
2. **Workflow Tracking Structure**:
    - Workflows as top-level containers
    - Actions for agent activities and tool use
    - Artifacts for outputs and files
    - Thought chains for reasoning processes
3. **Associative Memory Graph**:
    - Bidirectional links between memories
    - Type-classified relationships
    - Weighted link strengths
    - Hierarchical organization
4. **Cognitive State Management**:

   - Working memory management with capacity limits
   - Focus tracking and automatic updating
   - State persistence for context recovery
   - Workflow context summarization
5. **Meta-Cognitive Capabilities**:
   - Memory consolidation (summary, insight, procedural)
   - Reflection generation (summary, progress, gaps, strengths, plan)
   - Memory promotion based on usage patterns
   - Complex visualization generation
6. **Vector-Based Semantic Search**:
   - Integration with embedding services
   - Cosine similarity calculation
   - Hybrid search combining vector and keyword approaches
   - Optimized candidate selection
7. **Operation Audit and Analytics**:
   - Comprehensive operation logging
   - Statistical analysis and reporting
   - Performance measurement
   - Memory access tracking

This architecture enables advanced agent cognition through:

1. Systematic knowledge organization
2. Context-aware reasoning
3. Memory evolution and refinement
4. Meta-cognitive reflection
5. Structured workflow management
6. Rich visualization and reporting

The system provides a comprehensive foundation for sophisticated AI agent development with human-like memory organization and cognitive processes.

Architectural Motivation and Design Philosophy

The Unified Agent Memory and Cognitive System emerges from a fundamental challenge in AI agent development: creating systems that can maintain context, learn from experiences, understand patterns, and exhibit increasingly human-like cognitive capabilities. Traditional approaches to LLM agent architecture frequently suffer from several limitations:

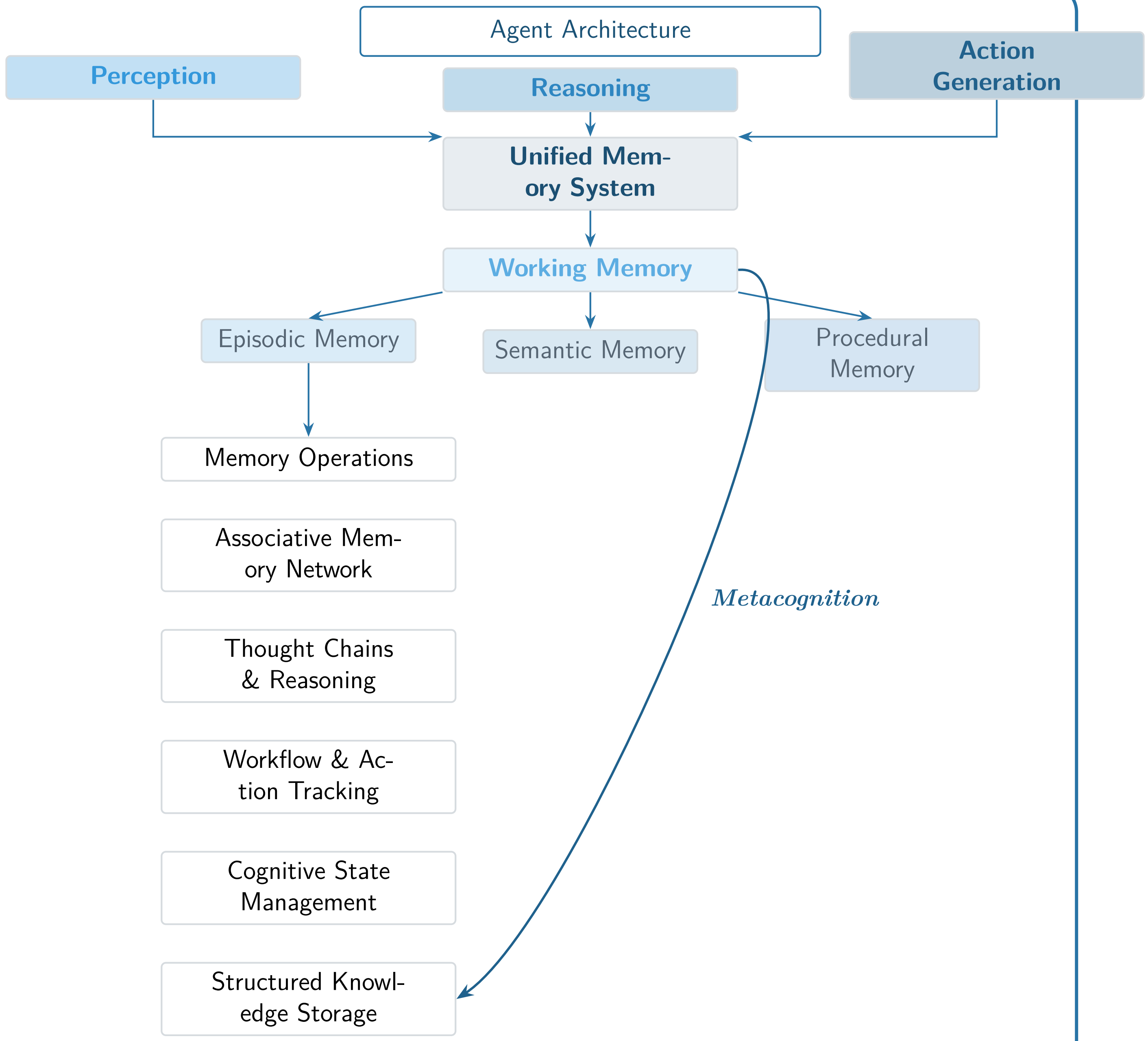
1. **Context Window Constraints**: LLMs have finite context windows, making long-term memory management essential
2. **Memory Organization**: Flat memory structures lack the nuanced organization that enables efficient retrieval
3. **Cognitive Continuity**: Maintaining coherent agent identity and learning across sessions
4. **Metacognitive Capabilities**: Enabling self-reflection and knowledge consolidation

This memory system addresses these challenges through a cognitive architecture inspired by human memory models while being optimized for computational implementation. The four-tiered memory hierarchy (working, episodic, semantic, procedural) draws from established psychological frameworks but adapts them for practical AI implementation:

```
Working Memory  → Episodic Memory  → Semantic Memory  → Procedural Memory
(Active focus)    (Experiences)      (Knowledge)        (Skills)
TTL: 30 minutes   TTL: 7 days        TTL: 30 days       TTL: 90 days
```

This progression models how information flows through and evolves within the system, mimicking how human cognition transforms experiences into knowledge and eventually into skills.

Integration with Agent Architecture



1. **Input Integration** ➔ Perceptions, observations, and inputs flow into episodic memory
2. **Reasoning Support** ➔ Thought chains & semantic memory support reasoning processes
3. **Action Context** ➔ Actions are recorded with reasoning and outcomes for future reference
4. **Metacognition** ➔ Consolidation and reflection processes enable higher-order cognition

Every part of the agent's functioning creates corresponding memory entries, allowing for persistent cognitive continuity across interactions.

Biomimetic Design and Cognitive Science Foundations

The system incorporates several principles from cognitive science:

Spreading Activation and Associative Networks

The memory link structure and semantic search implement a form of spreading activation, where retrieval of one memory activates related memories. Through functions like ‘get$_l inked_m emories()‘andtheworkin$

Memory Decay and Reinforcement

The implementation of importance decay and access-based reinforcement mirrors human memory dynamics:

```python
def _compute_memory_relevance(importance, confidence, created_at, access_count, last_accessed):
    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)
    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
    relevance = (decayed_importance * usage_boost * confidence * recency_factor)
    return min(max(relevance, 0.0), 10.0)
```

This function incorporates multiple cognitive principles:

- Memories decay over time with a configurable rate
- Frequently accessed memories remain relevant longer
- Recently accessed memories are prioritized
- Confidence acts as a weighting factor for reliability

Memory Evolution Pathways

The system models how information evolves through cognitive processing:

1. **Observation → Episodic**: Direct experiences and inputs enter as episodic memories
2. **Episodic → Semantic**: Through ‘promote$_m emory_l evel()‘, frequentlyaccessedepisodicmemoriesevolv$
   $Knowledgethatrepresentsskillsorprocedurescanbefurtherpromoted$
3. **Consolidation**: Through ‘consolidate$_m emories()‘, multiplerelatedmemoriessynthesizeintohigher-orderinsights$

This progression mimics human learning processes where repeated experiences transform into consolidated knowledge and eventually into skills and habits.

Architectural Implementation Details

The system implements these cognitive principles through sophisticated database design and processing logic:

Circular References and Advanced SQL Techniques

One unique aspect not fully explored in previous sections is the handling of circular references between memories and thoughts:

```sql
-- Deferrable Circular Foreign Key Constraints for thoughts <-> memories
BEGIN IMMEDIATE TRANSACTION;
PRAGMA defer_foreign_keys = ON;

ALTER TABLE thoughts ADD CONSTRAINT fk_thoughts_memory
    FOREIGN KEY (relevant_memory_id) REFERENCES memories(memory_id)
    ON DELETE SET NULL DEFERRABLE INITIALLY DEFERRED;

ALTER TABLE memories ADD CONSTRAINT fk_memories_thought
    FOREIGN KEY (thought_id) REFERENCES thoughts(thought_id)
    ON DELETE SET NULL DEFERRABLE INITIALLY DEFERRED;

COMMIT;
```

This implementation uses SQLite's deferred constraints to solve the chicken-and-egg problem of bidirectional references. This enables the creation of thoughts that reference memories, and memories that reference thoughts, without circular dependency issues during insertion.

Embedding Integration and Vector Search

The vector embedding system represents a crucial advancement in semantic retrieval. The code implements:

1. **Dimension-Aware Storage**: Embeddings include dimension metadata for compatibility checking
2. **Binary BLOB Storage**: Vectors are efficiently stored as binary blobs
3. **Model Tracking**: Embedding model information is preserved for future compatibility
4. **Optimized Retrieval**: Candidate pre-filtering happens before similarity calculation
5. **Hybrid Retrieval**: Combined vector and keyword search for robust memory access

This sophisticated approach enables the ”remembering-by-meaning” capability essential for human-like memory retrieval.

LLM Integration for Meta-Cognitive Functions

A distinctive aspect of this architecture is its use of LLMs for meta-cognitive processes:

Prompt Engineering for Cognitive Functions

The system includes carefully crafted prompts for various cognitive operations:

```python
def _generate_consolidation_prompt(memories: List[Dict], consolidation_type: str) -> str:
    # Format memory details...
    base_prompt = f"""You are an advanced cognitive system processing and consolidating
    memories for an AI agent. Below are {len(memories)} memory items containing
    information, observations, and insights relevant to a task. Your goal is to
    perform a specific type of consolidation: '{consolidation_type}'...
    """

    if consolidation_type == "summary":
        base_prompt += """TASK: Create a comprehensive and coherent summary that
        synthesizes the key information and context from ALL the provided memories...
        """
    # Additional consolidation types...
```

These prompts implement different cognitive functions by leveraging the LLM's capabilities within structured contexts:

1. **Summary**: Integration of information across memories
2. **Insight**: Pattern recognition and implication detection
3. **Procedural**: Extraction of generalizable procedures and methods
4. **Question**: Identification of knowledge gaps and uncertainties

Similarly, the reflection system analyzes agent behavior through targeted prompts:

```python
def _generate_reflection_prompt(workflow_name, workflow_desc, operations, memories, reflection_type):
    # Format operations with memory context...
    base_prompt = f"""You are an advanced meta-cognitive system analyzing an AI agent's
    workflow: "{workflow_name}"...
    """

    if reflection_type == "summary":
        base_prompt += """TASK: Create a reflective summary of this workflow's
        progress and current state...
        """
    # Additional reflection types...
```

These meta-cognitive capabilities represent an emergent property when LLMs are used to analyze the agent's own memory and behavior.

Cognitive State Management

An essential aspect of the system is its sophisticated cognitive state management:

Working Memory Optimization

The working memory implements capacity-constrained optimization:

```
async def optimize_working_memory(
    context_id: str,
    target_size: int = MAX_WORKING_MEMORY_SIZE,
    strategy: str = "balanced",  # balanced, importance, recency, diversity
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function implements multiple strategies for managing limited attentional capacity:

1. **Balanced**: Considers all relevance factors
2. **Importance**: Prioritizes important memories
3. **Recency**: Prioritizes recent memories
4. **Diversity**: Ensures varied memory types for broader context

These strategies mirror different cognitive styles and attentional priorities in human cognition.

Focus Management and Attention

The system implements attentional mechanisms through focus management:

```
async def auto_update_focus(
    context_id: str,
    recent_actions_count: int = 3,
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
```

This function models automatic attention shifting through sophisticated heuristics:

- Relevant to recent actions (recency bias)
- Memory type (questions and plans get priority)
- Memory level (semantic/procedural knowledge gets higher priority)
- Base relevance (importance, confidence)

This dynamic focus management creates an emergent attentional system resembling human cognitive focus.

Practical System Applications

The unified memory system enables several practical capabilities for AI agents:

Persistent Context Across Sessions

Through `save_cognitive_state()` and `load_cognitive_state()`, the system enables agents to maintain cognitive continuity across sessions. This allows for:

1. Persistent user relationships that evolve over time
2. Long-running projects with progress maintained between interactions
3. Incremental knowledge accumulation and refinement

Knowledge Evolution and Refinement

The memory evolution pathways (episodic → semantic → procedural) enable knowledge maturation. Key applications include:

1. Learning from repeated experiences
2. Developing expertise through information refinement
3. Converting learned patterns into reusable skills
4. Building increasingly sophisticated domain understanding

Meta-Cognitive Self-Improvement

Through reflection and consolidation, the system enables emergent self-improvement capabilities:

1. Identifying knowledge gaps through reflection
2. Consolidating fragmented observations into coherent insights
3. Recognizing patterns in its own problem-solving approaches
4. Refining strategies based on past successes and failures

These capabilities represent stepping stones toward more sophisticated cognitive agents with emergent meta-learning capabilities.

Performance Optimization and Scaling

The system incorporates numerous optimizations for practical deployment:

Database Performance Tuning

```
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"
]
```

These pragmas optimize SQLite for:

1. Write-Ahead Logging for concurrency
2. Memory-based temporary storage
3. Large cache size (32MB)
4. Memory-mapped I/O for performance
5. Extended busy timeout for reliability

Query Optimization

The schema includes comprehensive indexing:

```
-- 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);
-- Action indices
CREATE INDEX IF NOT EXISTS idx_actions_workflow_id ON actions(workflow_id);
-- Memory indices
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);
-- Many more indices...
```

With over 30 carefully designed indices covering most query patterns, the system ensures efficient database access despite complex query patterns.

Memory Management

The system implements sophisticated memory lifecycle management:

1. **Time-To-Live (TTL)**: Different memory levels have appropriate default lifespans
2. **Expiration Management**: `delete_expired_memories()` handles cleanup
3. **Importance-Based Prioritization**: More important memories persist longer
4. **Access Reinforcement**: Frequently used memories remain accessible

For large-scale deployments, the system could be extended with:

- Archival mechanisms for cold storage
- Distributed database backends for horizontal scaling
- Memory sharding across workflows

Visualization and Reporting Capabilities

The system includes sophisticated visualization that wasn't fully explored in previous sections:

Interactive Mermaid Diagrams

The `visualize_memory_network()` and `visualize_reasoning_chain()` functions generate interactive Mermaid diagrams that represent complex cognitive structures:

```
```

These visualizations enable:

1. Understanding complex memory relationships
2. Tracing reasoning pathways
3. Identifying key knowledge structures
4. Visualizing the agent's cognitive evolution

Comprehensive Reports

The `generate_workflow_report()` function creates detailed reports in multiple formats and styles:

1. **Professional**: Formal business-style reporting
2. **Concise**: Brief executive summaries
3. **Narrative**: Story-based explanations
4. **Technical**: Data-oriented technical documentation

These reporting capabilities make the agent's internal processes transparent and understandable to human collaborators.

Integration Examples and Workflow

Let's examine a complete workflow to understand how all components integrate:

1. **Workflow Creation**: Agent creates a workflow container for a data analysis task with `create_workflow()`
2. **Initial Goals**: Records initial goals as thoughts with `record_thought()`
3. **Action Planning**: Plans data loading as an action with `record_action_start()`
4. **Tool Execution**: Executes the data loading tool and records results with `record_action_completion(`
5. **Artifact Creation**: Saves loaded data as an artifact with `record_artifact()`
6. **Observation Creation**: Records observations about data as memories with `store_memory()`
7. **Memory Linking**: Creates associations between related observations with `create_memory_link()`

8. **Insight Generation**: Consolidates observations into insights with `consolidate_memories()`
9. **Action Planning (Continued)**: Plans analysis methods based on insights
10. **Execution and Recording**: Continues execution, recording results
11. **Reflection**: Periodically reflects on progress with `generate_reflection()`
12. **Focus Management**: Shifts focus based on current priorities with `auto_update_focus()`
13. **Memory Evolution**: Frequently accessed observations evolve into semantic knowledge with `promote_memory_level()`
14. **State Preservation**: Saves cognitive state with `save_cognitive_state()` for later continuation

This integrated workflow demonstrates how the memory system supports sophisticated cognitive processes while maintaining continuity, evolving knowledge, and enabling metacognition.

Future Extensions and Research Directions

The architecture lays groundwork for several advanced capabilities:

Multi-Agent Memory Sharing

The system could be extended for knowledge sharing between agents through:

- Standardized memory export/import
- Selective memory sharing protocols
- Cross-agent memory linking
- Collaborative knowledge building

Emotional and Motivational Components

Cognitive architectures could incorporate:

- Affective tagging of memories
- Motivation-based memory prioritization
- Emotional context for memory formation
- Value-aligned memory evolution

Neural-Symbolic Integration

Future versions might incorporate:

- Structured knowledge representations
- Logical reasoning over memory networks
- Constraint satisfaction for memory consistency
- Rule-based memory consolidation

Learning Optimizations

The system could be enhanced with:

- Adaptive memory promotion thresholds
- Personalized decay rates
- Learning rate parameters for different domains
- Automated memory organization optimization

Conclusion: Toward Emergent Cognitive Systems

The Unified Agent Memory and Cognitive System represents a sophisticated architecture that bridges traditional database systems with cognitive science-inspired memory models. By implementing a structured yet flexible memory architecture with meta-cognitive capabilities, it creates a foundation for increasingly sophisticated AI agents that can:

1. Learn from experiences through structured memory evolution
2. Maintain cognitive continuity across sessions
3. Develop increasingly refined understanding through consolidation
4. Engage in self-reflection and improvement
5. Organize and prioritize information effectively

As LLM-based agents continue to evolve, sophisticated memory architectures like this one will become increasingly essential for overcoming the limitations of context windows and enabling truly persistent, learning agents with emergent cognitive capabilities.

The system ultimately aims to address a core challenge in AI development: creating agents that don't just simulate intelligence in the moment, but that accumulate, refine, and evolve knowledge over time - a crucial stepping stone toward more capable and general artificial intelligence.

B Appendix B: Agent Master Loop (AML) Technical Analysis

EideticEngine Agent Master Loop (AML) Technical Analysis

Overview and Architecture

The code implements the EideticEngine Agent Master Loop (AML), an AI agent orchestration system that manages a sophisticated cognitive agent with capabilities inspired by human memory and reasoning.

The system orchestrates a primary think-act cycle where the agent:

1. Gathers comprehensive context from its memory systems
2. Consults a Large Language Model (LLM) for decision-making
3. Executes actions via tools
4. Updates its plans and goals
5. Performs periodic meta-cognitive operations
6. Maintains persistent state for continuity

Core Components

1. **AgentMasterLoop**: The main orchestrator class managing the entire agent lifecycle
2. **AgentState**: Dataclass representing the runtime state of the agent
3. **PlanStep**: Pydantic model representing individual steps in the agent's plan
4. **External Dependencies**:
    - MCPClient: Interface for the Unified Memory System (UMS)
    - AsyncAnthropic: Client for the Anthropic Claude LLM

Key Features and Innovations

- **Goal Stack Management**: Hierarchical goal decomposition with explicit goal states
- **Mental Momentum Bias**: Preference for completing current plan steps when progress is stable
- **Adaptive Thresholds**: Dynamically adjusts reflection and consolidation based on performance metrics
- **Background Task Management**: Robust concurrent processing with semaphores and timeouts

- **Structure-Aware Context**: Multi-faceted context with freshness indicators and priority weighting
- **Plan Validation**: Detects dependency cycles and validates plan structure
- **Categorized Error Handling**: Sophisticated error recovery with typed errors and fallback strategies

Core Data Structures

PlanStep (Pydantic BaseModel)

```python
class PlanStep(BaseModel):
    id: str = Field(default_factory=lambda: f"step-{MemoryUtils.generate_id()[:8]}")
    description: str
    status: str = "planned"
    depends_on: List[str] = Field(default_factory=list)
    assigned_tool: Optional[str] = None
    tool_args: Optional[Dict[str, Any]] = None
    result_summary: Optional[str] = None
    is_parallel_group: Optional[str] = None
```

This model represents a single step in the agent's plan, with fields for tracking:

- A unique identifier
- Description of the action
- Current status (planned, in_progress, completed, failed, skipped)
- Dependencies on other steps (for sequencing)
- Tool and arguments for execution
- Results after completion
- Optional parallel execution grouping

AgentState (Dataclass)

```python
@dataclass
class AgentState:
    # Workflow management
    workflow_id: Optional[str] = None
    context_id: Optional[str] = None
    workflow_stack: List[str] = field(default_factory=list)

    # Goal management
    goal_stack: List[Dict[str, Any]] = field(default_factory=list)
    current_goal_id: Optional[str] = None

    # Planning & reasoning
    current_plan: List[PlanStep] = field(
        default_factory=lambda: [PlanStep(description=DEFAULT_PLAN_STEP)]
    )
    current_thought_chain_id: Optional[str] = None
    last_action_summary: str = "Loop initialized."
    current_loop: int = 0
    goal_achieved_flag: bool = False

    # Error tracking
    consecutive_error_count: int = 0
    needs_replan: bool = False
    last_error_details: Optional[Dict[str, Any]] = None

    # Meta-cognition metrics
    successful_actions_since_reflection: float = 0.0
    successful_actions_since_consolidation: float = 0.0
    loops_since_optimization: int = 0
    loops_since_promotion_check: int = 0
    loops_since_stats_adaptation: int = 0
    loops_since_maintenance: int = 0
    reflection_cycle_index: int = 0
    last_meta_feedback: Optional[str] = None
    # Adaptive thresholds
    current_reflection_threshold: int = BASE_REFLECTION_THRESHOLD
    current_consolidation_threshold: int = BASE_CONSOLIDATION_THRESHOLD

    # Tool statistics
    tool_usage_stats: Dict[str, Dict[str, Union[int, float]]] = field(
        default_factory=_default_tool_stats
    )

    # Background tasks (transient)
    background_tasks: Set[asyncio.Task] = field(
        default_factory=set, init=False, repr=False
    )
```

This comprehensive state object maintains:

- Workflow context and goal hierarchy
- Current execution plan and thought records
- Error tracking information
- Meta-cognitive metrics and thresholds
- Tool usage statistics
- Active background tasks

Key Constants and Configuration

The system uses numerous constants for configuration, many of which can be overridden via environment variables:

```
# File and agent identification
AGENT_STATE_FILE = "agent_loop_state_v4.1.json"
AGENT_NAME = "EidenticEngine4.1"
MASTER_LEVEL_AGENT_LLM_MODEL_STRING = "claude-3-7-sonnet-20250219"

# Meta-cognition thresholds
BASE_REFLECTION_THRESHOLD = int(os.environ.get("BASE_REFLECTION_THRESHOLD", "7"))
BASE_CONSOLIDATION_THRESHOLD = int(os.environ.get("BASE_CONSOLIDATION_THRESHOLD", "12"))
MIN_REFLECTION_THRESHOLD = 3
MAX_REFLECTION_THRESHOLD = 15
MIN_CONSOLIDATION_THRESHOLD = 5
MAX_CONSOLIDATION_THRESHOLD = 25
THRESHOLD_ADAPTATION_DAMPENING = float(os.environ.get("THRESHOLD_DAMPENING", "0.75"))
MOMENTUM_THRESHOLD_BIAS_FACTOR = 1.2

# Interval constants
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"))
MAINTENANCE_INTERVAL = int(os.environ.get("MAINTENANCE_INTERVAL", "50"))

# Context limits
CONTEXT_RECENT_ACTIONS_FETCH_LIMIT = 10
CONTEXT_IMPORTANT_MEMORIES_FETCH_LIMIT = 7
CONTEXT_KEY_THOUGHTS_FETCH_LIMIT = 7
# ...and many more context limit constants

# Background task management
BACKGROUND_TASK_TIMEOUT_SECONDS = 60.0
MAX_CONCURRENT_BG_TASKS = 10
```

Unified Memory System Tool Constants

The system defines constants for UMS tool names to ensure consistency:

```
# Core memory tools
TOOL_STORE_MEMORY = "unified_memory:store_memory"
TOOL_UPDATE_MEMORY = "unified_memory:update_memory"
TOOL_GET_MEMORY_BY_ID = "unified_memory:get_memory_by_id"
TOOL_HYBRID_SEARCH = "unified_memory:hybrid_search_memories"
```

```python
TOOL_SEMANTIC_SEARCH = "unified_memory:search_semantic_memories"
TOOL_QUERY_MEMORIES = "unified_memory:query_memories"

# Working memory tools
TOOL_GET_WORKING_MEMORY = "unified_memory:get_working_memory"
TOOL_OPTIMIZE_WM = "unified_memory:optimize_working_memory"
TOOL_AUTO_FOCUS = "unified_memory:auto_update_focus"

# Meta-cognitive tools
TOOL_REFLECTION = "unified_memory:generate_reflection"
TOOL_CONSOLIDATION = "unified_memory:consolidate_memories"
TOOL_PROMOTE_MEM = "unified_memory:promote_memory_level"

# Goal stack tools
TOOL_PUSH_SUB_GOAL = "unified_memory:push_sub_goal"
TOOL_MARK_GOAL_STATUS = "unified_memory:mark_goal_status"
TOOL_GET_GOAL_DETAILS = "unified_memory:get_goal_details"

# Workflow tools
TOOL_CREATE_WORKFLOW = "unified_memory:create_workflow"
TOOL_UPDATE_WORKFLOW_STATUS = "unified_memory:update_workflow_status"
TOOL_GET_WORKFLOW_DETAILS = "unified_memory:get_workflow_details"

# Internal agent tool
AGENT_TOOL_UPDATE_PLAN = "agent:update_plan"
```

Core Utility Functions

The system includes several helper functions:

```python
def _fmt_id(val: Any, length: int = 8) -> str:
    """Format an ID for readable logs, truncated to specified length."""

def _utf8_safe_slice(s: str, max_len: int) -> str:
    """Return a UTF-8 boundary-safe slice within max_len bytes."""

def _truncate_context(context: Dict[str, Any], max_len: int = 25_000) -> str:
    """Structure-aware context truncation with UTF-8 safe fallback."""

def _default_tool_stats() -> Dict[str, Dict[str, Union[int, float]]]:
    """Factory function for initializing tool usage statistics dictionary."""

def _detect_plan_cycle(self, plan: List[PlanStep]) -> bool:
    """Detects cyclic dependencies in the agent's plan using Depth First Search."""
```

A distinctive feature of the context system is the explicit inclusion of temporal awareness through 'freshness' indicators:

```python
retrieval_timestamp = datetime.now(timezone.utc).isoformat()
```

Throughout context gathering, each component is tagged with when it was retrieved:

```python
base_context['core_context']['retrieved_at'] = retrieval_timestamp
```

These timestamp indicators serve several critical functions:

1. They enable the LLM to reason about potentially stale information
2. They help prioritize more recent information in decision-making
3. They provide clear signals about the temporal relationship between different context components

By tagging context components with retrieval timestamps, the system creates a time-aware context representation that helps the LLM make more temporally grounded decisions, mimicking human awareness of information recency.

AgentMasterLoop Core Implementation

Initialization and Setup

```
def __init__(self, mcp_client_instance: MCPClient, agent_state_file: str = AGENT_STATE_FILE):
```

The constructor initializes the agent with a reference to the MCPClient (interface to the Unified Memory System) and state file path. It:

- Validates the MCPClient dependency is available
- Stores a reference to the Anthropic client for LLM interaction
- Sets up logger configuration
- Initializes state and synchronization primitives
- Configures cognitive process parameters including thresholds and reflection types

```
async def initialize(self) -> bool:
```

This method performs crucial initialization steps:

1. Loads prior agent state from the file system
2. Fetches available tool schemas from MCPClient
3. Filters schemas to only those relevant to this agent (UMS tools and internal agent tools)
4. Verifies essential tools are available
5. Validates the workflow ID loaded from state file
6. Validates the loaded goal stack consistency
7. Sets the default thought chain ID if needed

The method uses these helper methods for specific tasks:

```
async def _load_agent_state(self) -> None:
```

Loads and validates agent state from the JSON file, handling potential issues:

- File not found (initializes with defaults)
- JSON decoding errors
- Structure mismatches between saved state and current AgentState dataclass
- Type validation and conversion for complex nested structures
- Consistency checks for loaded thresholds and goal stack

```
async def _validate_goal_stack_on_load(self):
```

Verifies goal stack integrity:

- Checks if goals still exist in the UMS
- Confirms goals belong to the correct workflow
- Removes invalid/missing goals
- Updates current_goal_id if needed
- Handles empty stack case

```
async def _set_default_thought_chain_id(self):
```

Sets the active thought chain:

- Retrieves thought chains associated with the current workflow
- Selects the primary (usually first created) chain
- Updates the agent state with the chain ID

```
async def _check_workflow_exists(self, workflow_id: str) -> bool:
```

Efficiently verifies workflow existence by making a minimal UMS query

Main Agent Loop

The primary execution loop is implemented in:

```
async def run(self, goal: str, max_loops: int = 100) -> None:
```

This method orchestrates the entire agent lifecycle:

1. **Setup Phase**:
   - Initializes workflow if none exists
   - Creates initial thought chain if needed
   - Ensures current_goal_id is set
2. **Main Loop Execution**: In each iteration:
   - Run periodic cognitive tasks
   - Gather comprehensive context
   - Call LLM for decision
   - Execute the decided action
   - Apply plan updates (explicit or heuristic)
   - Check error limits
   - Save state
3. **Termination Handling**:
   - Goal achieved signal
   - Max loop reached
   - Shutdown signal
   - Error limit exceeded
   - Final state save and cleanup

This loop continues until a termination condition is met: achieving the goal, reaching max iterations, receiving a shutdown signal, or hitting the error limit.

Context Gathering System

```
async def _gather_context(self) -> Dict[str, Any]:
```

This sophisticated method assembles a multi-faceted context for the LLM: Here is the fully revised version with all backticked expressions converted to “ and underscores escaped:

1. **Base Context Structure**: Creates initial context with basic state information
   - Current loop count, workflow/context IDs, current plan
   - Error details, replan flag, workflow stack
   - Placeholders for fetched components
2. **Goal Stack Context**: Retrieves and structures goal hierarchy data
   - Fetches details of current goal
   - Includes summary of goal stack (limited by `CONTEXT_GOAL_STACK_SHOW_LIMIT`)
   - Handles cases where goal tools are unavailable
3. **Core Context**: Retrieves foundational workflow context
   - Recent actions (limited by `CONTEXT_RECENT_ACTIONS_FETCH_LIMIT`)
   - Important memories (limited by `CONTEXT_IMPORTANT_MEMORIES_FETCH_LIMIT`)
   - Key thoughts (limited by `CONTEXT_KEY_THOUGHTS_FETCH_LIMIT`)
   - Adds freshness timestamp
4. **Working Memory**: Retrieves active memories and focal point

    - Gets current working memories and `focal_memory_id`
    - Stores full result with freshness timestamp
    - Extracts focal ID and memory list for later use
5. **Proactive Memories**: Performs goal-directed memory retrieval
    - Formulates query based on current step/goal
    - Uses hybrid search with semantic/keyword balance
    - Formats results with scores and freshness
6. **Procedural Memories**: Finds relevant how-to knowledge
    - Focuses query on accomplishing current step/goal
    - Explicitly filters for procedural memory level
    - Formats with scores and freshness
7. **Contextual Link Traversal**: Explores memory relationships
    - Prioritizes focal memory, falls back to working or important memories
    - Retrieves incoming and outgoing links
    - Creates concise summary with limited link details
8. **Context Compression**: Checks if context exceeds token threshold
    - Estimates token count using Anthropic API
    - Compresses verbose parts if needed
    - Generates summary using UMS summarization tool

Let me know if you'd like all these replacements applied automatically across a LaTeX source file.

The gathered context includes freshness indicators for each component and detailed error tracking throughout the process.

Contextual Link Traversal Strategy

The system implements a sophisticated prioritization strategy for memory graph exploration during contextual link traversal:

1. First prioritizes the focal memory from working memory (mimicking conscious attention)
2. Falls back to the first memory in working memory if no focal exists (using working memory prominence)
3. Falls back further to important memories from core context (leveraging episodic salience)

This multi-level fallback approach models human associative memory traversal, where we naturally follow connections from whatever is most present in our awareness at the moment. By prioritizing the focal memory, the system mirrors how human attention guides associative thinking.

The system implements a sophisticated prioritization strategy during contextual link traversal that models human associative memory. It follows a three-tier fallback approach:

1. First prioritizes the focal memory from working memory (mimicking conscious attention):

    ```
    if focal_mem_id_from_wm:
        mem_id_to_traverse = focal_mem_id_from_wm
    ```

2. Falls back to the first memory in working memory if no focal exists:

    ```
    if not mem_id_to_traverse and working_mem_list_from_wm:
        first_wm_item = working_mem_list_from_wm[0]
        mem_id_to_traverse = first_wm_item.get('memory_id')
    ```

3. Falls back further to important memories from core context:

```
if not mem_id_to_traverse:
    important_mem_list = core_ctx_data.get('important_memories', [])
    if important_mem_list:
        first_mem = important_mem_list[0]
        mem_id_to_traverse = first_mem.get('memory_id')
```

This multi-level fallback approach models how human attention guides associative thinking, always traversing from whatever is most present in our awareness at the moment.

LLM Interaction

```
async def _call_agent_llm(self, goal: str, context: Dict[str, Any]) -> Dict[str, Any]:
```

This method handles the LLM interaction:

1. Constructs the LLM prompt using `_construct_agent_prompt`
2. Defines tools for the Anthropic API
3. Makes the API call with retry logic for transient errors
4. Parses the LLM response to extract the agent's decision:
    - Tool call (`"decision": "call_tool", "tool_name": str, "arguments": dict`)
    - Thought process (`"decision": "thought_process", "content": str`)
    - Goal completion (`"decision": "complete", "summary": str`)
    - Error (`"decision": "error", "message": str`)
    - Plan update (`"decision": "plan_update", "updated_plan_steps": List[PlanStep]`)

The prompt construction is particularly sophisticated:

```
def _construct_agent_prompt(self, goal: str, context: Dict[str, Any]) -> List[Dict[str, Any]]:
```

This method creates a detailed prompt structure:

1. **System Instructions**:
    - Agent identity and capabilities
    - Available tools with schemas (highlighting essential cognitive tools)
    - Current goal context from goal stack
    - Detailed process instructions (analysis, error handling, reasoning, planning, action)
    - Key considerations (goal focus, mental momentum, dependencies)
    - Recovery strategies for different error types
2. **User Message**:
    - Current context (JSON, robustly truncated)
    - Current plan (JSON)
    - Last action summary
    - Error details (prominently highlighted if present)
    - Meta-cognitive feedback
    - Current goal reminder
    - Final instruction

The prompt emphasizes goals, error recovery, plan repair, and mental momentum bias.

Tool Execution System

```
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]:
```

This central method handles all tool execution with comprehensive functionality:

1. **Server Lookup**: Finds the appropriate server for the tool

2. **Context Injection**: Adds workflow/context IDs if missing but relevant
3. **Dependency Check**: Verifies prerequisites before execution
4. **Internal Tool Handling**: Processes the `AGENT_TOOL_UPDATE_PLAN` tool internally
5. **Plan Validation**: Checks for dependency cycles in plan updates
6. **Action Recording**: Optionally records action start/dependencies in UMS
7. **Tool Execution**: Calls the tool via MCPClient with retry logic
8. **Result Processing**: Standardizes result format and categorizes errors
9. **Background Triggers**: Initiates auto-linking and promotion checks when appropriate
10. **State Updates**: Updates statistics, error details, and last action summary
11. **Action Completion**: Records the outcome in UMS
12. **Side Effects**: Handles workflow and goal stack implications

The method uses several helper methods for specific subtasks:

```
def _find_tool_server(self, tool_name: str) -> Optional[str]:
```

Locates the server providing the specified tool, handling internal tools and server availability.

```
async def _check_prerequisites(self, ids: List[str]) -> Tuple[bool, str]:
```

Verifies that all specified prerequisite action IDs have status 'completed'.

```
async def _record_action_start_internal(self, tool_name: str, tool_args: Dict[str, Any], planned_dependencies:
Optional[List[str]] = None) -> Optional[str]:
```

Records the start of an action in UMS, handling dependencies.

```
async def _record_action_dependencies_internal(self, source_id: str, target_ids: List[str]) -> None:
```

Records dependency relationships between actions in UMS.

```
async def _record_action_completion_internal(self, action_id: str, result: Dict[str, Any]) -> None:
```

Records the completion status and result for a given action.

```
async def _handle_workflow_and_goal_side_effects(self, tool_name: str, arguments: Dict, result_content: Dict):
```

Manages state changes triggered by specific tool outcomes, including:

- Workflow creation/termination
- Goal stack updates (push/pop)
- Goal status changes
- Sub-workflow management

```
async def _with_retries(self, coro_fun, *args, max_retries: int = 3, retry_exceptions: Tuple[type[BaseException],
...] = (...), retry_backoff: float = 2.0, jitter: Tuple[float, float] = (0.1, 0.5), **kwargs):
```

Generic retry wrapper with exponential backoff and jitter, handling various exception types.

Plan Management

```
async def _apply_heuristic_plan_update(self, last_decision: Dict[str, Any], last_tool_result_content:
Optional[Dict[str, Any]] = None):
```

This method updates the plan based on action outcomes when the LLM doesn't explicitly call 'agent:update$_plan$' :

Success Case:

- Marks step as completed
- Removes completed step from plan
- Generates a summary of the result
- Adds a final analysis step if plan becomes empty
- Resets error counter

Failure Case:

- Marks step as failed
- Keeps failed step in plan for context
- Inserts an analysis step for recovery
- Sets `needs_replan` flag to true
- Updates consecutive error count

Thought Case:

- Marks step as completed
- Creates summary from thought content
- Adds next action step if plan becomes empty
- Counts as partial progress for metacognitive metrics

Completion Case:

- Marks as success
- Updates plan with a finalization step
- Status handled in main loop

The method also updates meta-cognitive counters based on success/failure.

The heuristic plan update system implements nuanced strategies for different decision types:

1. **Success Case (Tool Call)**:
    - Marks step as completed with detailed result summary
    - Removes step from plan and handles empty plan
    - Resets error counter and `needs_replan` flag
    - Increments meta-cognitive success counters
2. **Failure Case (Tool Call)**:
    - Marks step as failed but preserves it in the plan
    - Inserts an analysis step after the failed step
    - Sets `needs_replan` flag
    - Increments `consecutive_error_count`
    - Resets reflection counter to trigger faster reflection
3. **Thought Process Case**:
    - Marks step as completed with thought content summary
    - Removes step from plan and handles empty plan
    - Increments meta-cognitive counters at reduced weight (0.5)
    - Preserves `needs_replan` state
4. **Completion Signal Case**:
    - Creates finalization step
    - Sets `goal_achieved_flag` for main loop termination
5. **Error or Unknown Decision Case**:
    - Differentiates between plan update tool failures and other errors
    - Inserts re-evaluation step
    - Forces `needs_replan` flag

- Updates error counters appropriately

These sophisticated heuristics enable coherent plan progression even when the LLM doesn't explicitly update the plan, creating a robust fallback that maintains execution consistency.

```
def _detect_plan_cycle(self, plan: List[PlanStep]) -> bool:
```

This method uses depth-first search to detect cyclic dependencies in the agent's plan:

1. Builds an adjacency list from dependency relationships
2. Implements DFS with path tracking for cycle detection
3. Returns true if any cycle is found, false otherwise

Goal Stack Management

The core of the goal stack feature is distributed across several methods:

1. **Goal Context Gathering**:
   - Fetches current goal details from UMS
   - Provides a summary of the goal stack
   - Includes in LLM context
2. **Goal Stack Side Effects**:
   - Creates root goal when creating new workflow
   - Updates goal status when marked by LLM
   - Pops completed/failed goals from stack
   - Sets `goal_achieved_flag` when root goal completes
   - Manages workflow status when goal stack empties
3. **Sub-Workflow Integration**:
   - Associates sub-workflows with goals
   - Updates goal status when sub-workflow completes
   - Returns to parent workflow context when sub-workflow finishes

The goal stack is stored in the `AgentState`:

```
goal_stack: List[Dict[str, Any]] = field(default_factory=list)
current_goal_id: Optional[str] = None
```

Where each goal dictionary contains:

```
{
  "goal_id": str,          # Unique identifier
  "description": str,      # Text description of the goal
  "status": str,           # "active", "completed", "failed"
  "parent_goal_id": str,   # Optional reference to parent goal
  # Other potential fields from UMS
}
```

The system maintains a clear separation between two hierarchical stacks that serve distinct purposes:

1. **Workflow Stack**: Manages execution contexts across potentially different environments or domains. A sub-workflow might represent an entirely separate task context with its own memory space and thought chains, but still connected to the parent workflow. This enables modularity and compartmentalization of execution environments.
2. **Goal Stack**: Manages hierarchical decomposition of objectives within a single workflow context. Goals in the stack share the same memory space and thought chain, representing progressive refinement of intentions rather than context switching.

This dual-stack approach enables sophisticated task management where the agent can both decompose goals hierarchically (via goal stack) and switch entire working contexts (via workflow stack). When a sub-workflow completes, the system automatically marks the corresponding goal in the parent workflow, creating a seamless bridge between these two hierarchical mechanisms.

Adaptive Threshold System

```
def _adapt_thresholds(self, stats: Dict[str, Any]) -> None:
```

This method dynamically adjusts reflection and consolidation thresholds:

1. **Consolidation Threshold Adaptation**:
   - Analyzes episodic memory ratio in total memories
   - Computes deviation from target ratio range
   - Adjusts threshold to maintain optimal memory balance
   - Applies dampening to prevent large swings
2. **Reflection Threshold Adaptation**:
   - Analyzes tool failure rate from usage statistics
   - Computes deviation from target failure rate
   - Adjusts threshold based on performance
   - Applies dampening factor to stabilize changes
3. **Mental Momentum Bias**:
   - Detects stable progress periods (low error rate)
   - Applies positive bias to reflection threshold when stable
   - Makes agent less likely to reflect during productive periods
   - Multiplies adjustments by `MOMENTUM_THRESHOLD_BIAS_FACTOR`

The system ensures all thresholds stay within configured `MIN/MAX` bounds. The Mental Momentum Bias is implemented as a key cognitive feature in this function. When the agent detects stable progress periods (failure rate $<$ 50% of target and zero consecutive errors), it applies the `MOMENTUM_THRESHOLD_BIAS_FACTOR` multiplier (1.2) specifically to positive threshold adjustments. This creates a measurable 'flow state' effect where reflection becomes less frequent during productive periods, allowing the agent to maintain momentum. The system only applies this bias to increasing threshold adjustments, not to decreasing ones, ensuring quick response to errors while extending productive periods. This momentum effect is then dampened through the `THRESHOLD_ADAPTATION_DAMPENING` factor (0.75) to prevent overreaction to temporary improvements.

The Mental Momentum Bias represents a key cognitive feature of the system. When the agent is making stable progress (low error rates), it applies a multiplier (`MOMENTUM_THRESHOLD_BIAS_FACTOR`) to increase the reflection threshold adjustment. This creates a 'flow state' where the agent is less likely to interrupt its productive work with reflection, mirroring how humans maintain momentum when things are going well. This bias dynamically balances between steady execution and necessary adaptation, creating more human-like task progression patterns.

Background Task Management

The system implements sophisticated background task handling:

```
def _start_background_task(self, coro_fn, *args, **kwargs) -> asyncio.Task:
```

This method creates and manages background tasks with five critical reliability features:

1. **State snapshotting**: Captures workflow/context IDs at task creation time, ensuring tasks operate with consistent state even if the main agent state changes before execution.

   ```
   # Snapshot critical state needed by the background task
   snapshot_wf_id = self.state.workflow_id
   snapshot_ctx_id = self.state.context_id
   ```

2. **Semaphore-based concurrency limiting**: Prevents resource exhaustion by limiting concurrent background tasks to MAX_CONCURRENT_BG_TASKS.

   ```
   await self._bg_task_semaphore.acquire()
   ```

3. **Timeout handling**: All background tasks have built-in timeouts to prevent hanging.

   ```
   await asyncio.wait_for(
       coro_fn(...),
       timeout=BACKGROUND_TASK_TIMEOUT_SECONDS
   )
   ```

4. **Thread-safe tracking**: Uses asyncio locks to safely manage the background task set.

   ```
   async with self._bg_tasks_lock:
       self.state.background_tasks.add(task)
   ```

5. **Guaranteed resource release**: Completion callbacks ensure semaphores are released even on failure.

   ```
   task.add_done_callback(self._background_task_done)
   ```

The system provides robust shutdown cleanup through the `_cleanup_background_tasks` method, which safely cancels all pending tasks, awaits their completion, and verifies semaphore state integrity. This comprehensive approach ensures cognitive background processes operate reliably without interrupting the main agent loop, mimicking human parallel thought processes.

Periodic Meta-Cognitive Tasks

```
async def _run_periodic_tasks(self):
```

This method orchestrates scheduled cognitive maintenance tasks:

1. **Stats Computation & Threshold Adaptation**:
   - Triggered by `STATS_ADAPTATION_INTERVAL`
   - Computes memory statistics
   - Adapts thresholds based on performance
   - Potentially triggers immediate consolidation if needed
2. **Reflection**:
   - Triggered by successful actions exceeding threshold or `replan` flag
   - Cycles through different reflection types
   - Provides feedback for next prompt
   - Forces replanning after significant insights
3. **Consolidation**:
   - Triggered by successful actions exceeding threshold
   - Summarizes episodic memories into semantic knowledge

- Integrates information across sources
- Provides feedback for next prompt

4. **Working Memory Optimization**:
    - Triggered by `OPTIMIZATION_LOOP_INTERVAL`
    - Improves relevance/diversity of working memory
    - Updates focus to most important memory
    - Maintains cognitive efficiency
5. **Memory Promotion Check**:
    - Triggered by `MEMORY_PROMOTION_LOOP_INTERVAL`
    - Finds recently accessed memories
    - Checks promotion criteria for each
    - Elevates memory levels when appropriate
6. **Maintenance**:
    - Triggered by `MAINTENANCE_INTERVAL`
    - Deletes expired memories
    - Maintains memory system health

The method prioritizes tasks, handles exceptions, and ensures graceful shutdown if requested.

```
async def _trigger_promotion_checks(self):
```

Helper method that:

- Queries for recently accessed memories
- Identifies candidates for promotion
- Schedules background checks for each
- Handles both Episodic→Semantic and Semantic→Procedural candidates

Comprehensive Error Handling System

The AML implements a sophisticated error handling framework that categorizes errors, provides detailed context for recovery, and implements graceful degradation strategies.

Error Categorization and Recording

```
# Error state tracking in AgentState
consecutive_error_count: int = 0
needs_replan: bool = False
last_error_details: Optional[Dict[str, Any]] = None

# Error categorization in _execute_tool_call_internal
error_type = "ToolExecutionError"  # Default category
status_code = res.get("status_code")
error_message = res.get("error", "Unknown failure")

if status_code == 412: error_type = "DependencyNotMetError"
elif status_code == 503: error_type = "ServerUnavailable"
elif "input" in str(error_message).lower() or "validation" in str(error_message).lower(): error_type =
"InvalidInputError"
elif "timeout" in str(error_message).lower(): error_type = "NetworkError"
elif tool_name in [TOOL_PUSH_SUB_GOAL, TOOL_MARK_GOAL_STATUS] and ("not found" in str(error_message).lower()
or "invalid" in str(error_message).lower()):
     error_type = "GoalManagementError"
```

The system maintains a structured error record with:

- Tool name and arguments that caused the error

- Error message and status code
- Categorized error type
- Additional context-specific fields

This structured approach allows for targeted recovery strategies and detailed feedback to the LLM.

Error Types and Recovery Strategies

The system implements a sophisticated error classification system with at least ten distinct categories, each triggering specific recovery behaviors:

1. **InvalidInputError**: Occurs when tool arguments fail validation. Recovery involves reviewing schemas, correcting arguments, or selecting alternative tools.
2. **DependencyNotMetError**: Triggered when prerequisite actions aren't completed. The system checks dependency status, waits for completion, or adjusts plan order.
3. **ServerUnavailable/NetworkError**: Indicates tool servers are unreachable. The agent attempts alternative tools, implements waiting periods, or adjusts plans to account for unavailable services.
4. **APILimitError/RateLimitError**: Occurs when external API limits are reached. Recovery includes implementing wait periods and reducing request frequency.
5. **ToolExecutionError/ToolInternalError**: Represents failures during tool execution. The agent analyzes error messages to determine if different arguments or alternative tools might succeed.
6. **PlanUpdateError**: Indicates invalid plan structure. The system re-examines steps and dependencies to correct structural issues.
7. **PlanValidationError**: Triggered when logical issues like cycles are detected. The agent debugs dependencies and proposes corrected structures.
8. **CancelledError**: Occurs when actions are cancelled, often during shutdown. The agent re-evaluates the current step upon resumption.
9. **GoalManagementError**: Indicates failures in goal stack operations. Recovery involves reviewing the goal context and stack logic.
10. **UnknownError/UnexpectedExecutionError**: Catchall for unclassified errors. The agent analyzes messages, simplifies steps, or seeks clarification.

The system differentiates between transient errors (appropriate for retry) and permanent ones, with dedicated handling strategies for each category. This explicit categorization is communicated to the LLM for targeted recovery actions.

Error Handling Implementation

The error handling is distributed across several layers:

1. **Tool Call Level** (`_execute_tool_call_internal`):
   - Categorizes errors based on messages/codes
   - Updates `state.last_error_details` with structured info
   - Increments `consecutive_error_count`
   - Sets `needs_replan` flag when appropriate
   - Updates `last_action_summary` with error context
2. **LLM Decision Level** (`_call_agent_llm`):
   - Handles API errors with specific categorization
   - Retries transient errors with exponential backoff
   - Returns structured error decision when needed

3. **Plan Update Level** (`_apply_heuristic_plan_update`):
    - Handles errors by marking steps as failed
    - Inserts analysis steps for error recovery
    - Adjusts counters based on failure type
4. **Main Loop Level** (`run`):
    - Checks `consecutive_error_count` against `MAX_CONSECUTIVE_ERRORS`
    - Terminates execution if error threshold exceeded
    - Updates workflow status to `FAILED` if appropriate
5. **Background Task Level**:
    - Implements timeout handling for all background tasks
    - Manages exceptions without disrupting main loop
    - Ensures semaphore release even on failure

The system also exposes error details prominently in the LLM prompt:

```
# From _construct_agent_prompt
if self.state.last_error_details:
    user_blocks += [
        "**CRITICAL: Address Last Error Details**:",
        "```json",
        json.dumps(self.state.last_error_details, indent=2, default=str),
        "```",
        "",
    ]
```

Retry Mechanism

The system implements sophisticated retry logic with:

```
async def _with_retries(
    self,
    coro_fun,
    *args,
    max_retries: int = 3,
    retry_exceptions: Tuple[type[BaseException], ...] = (...),
    retry_backoff: float = 2.0,
    jitter: Tuple[float, float] = (0.1, 0.5),
    **kwargs,
):
```

This wrapper provides:

- Configurable max retry attempts
- Exponential backoff (each delay = previous * backoff_factor)
- Random jitter to prevent thundering herd problems
- Selective retry based on exception types
- Cancellation detection during retry waits
- Detailed logging of retry attempts

It's selectively applied based on operation idempotency:

```
# In _execute_tool_call_internal
idempotent = tool_name in {
    # Read-only operations are generally safe to retry
    TOOL_GET_CONTEXT, TOOL_GET_MEMORY_BY_ID, TOOL_SEMANTIC_SEARCH,
    TOOL_HYBRID_SEARCH, TOOL_GET_ACTION_DETAILS, TOOL_LIST_WORKFLOWS,
    # ...many more tools
}

# Execute with appropriate retry count
raw = await self._with_retries(
```

```
    _do_call,
    max_retries=3 if idempotent else 1,  # Retry only idempotent tools
    # Specify exceptions that should trigger a retry attempt
    retry_exceptions=(
        ToolError, ToolInputError,  # Specific MCP errors
        asyncio.TimeoutError, ConnectionError,  # Common network issues
        APIConnectionError, RateLimitError, APIStatusError,  # Anthropic/API issues
    ),
)
```

Beyond retry logic, the system implements comprehensive dynamic tool availability management:

```
def _find_tool_server(self, tool_name: str) -> Optional[str]:
```

This method handles tool availability by:

1. Checking if the tool's server is currently active in the `MCPClient`'s server manager
2. Providing special handling for the internal `AGENT_TOOL_UPDATE_PLAN` tool
3. Attempting to route core tools to the `'CORE'` server if available

Throughout the codebase, tool calls are guarded with availability checks:

```
if self._find_tool_server(tool_name):
    # Execute tool with proper handling
else:
    # Log unavailability and implement fallback behavior
```

The system implements sophisticated fallback mechanisms when preferred tools are unavailable:

- For search operations, falling back from hybrid search to pure semantic search
- For context gathering, continuing with partial context when certain components can't be fetched
- For meta-cognitive operations, skipping non-critical tasks while preserving core functionality

This robust handling of tool availability ensures the agent degrades gracefully in distributed environments where services may be temporarily unavailable, mimicking human adaptability to missing resources.

Token Estimation and Context Management

Token Estimation

```
async def _estimate_tokens_anthropic(self, data: Any) -> int:
```

This method provides accurate token counting:

1. Uses the Anthropic API's `count_tokens` method for precise estimation
2. Handles both string and structured data by serializing if needed
3. Provides a fallback heuristic (chars/4) if the API call fails
4. Returns consistent integer results for all cases

This token counting is crucial for:

- Determining when context compression is needed
- Ensuring LLM inputs stay within model context limits
- Optimizing token usage for cost efficiency

Context Truncation and Compression

```
def _truncate_context(context: Dict[str, Any], max_len: int = 25_000) -> str:
```

This sophisticated utility implements a cognitively-informed, multi-stage context truncation strategy:

1. **Initial JSON Serialization**: Attempts standard serialization
2. **Structure-Aware Prioritized Truncation**: If size exceeds limit, applies intelligent reductions:
   - Truncates lists based on priority and `SHOW_LIMIT` constants
   - Applies different limits to different context types (`working_memory`, `recent_actions`, `goal_stack`, etc.)
   - Adds explicit notes about omissions to maintain context coherence
   - Preserves original structure and semantics
3. **Prioritized Component Removal**: If still too large, removes entire low-priority components in this specific order:
   - `relevant_procedures` (lowest priority)
   - `proactive_memories`
   - `contextual_links`
   - core context components (in priority order)
   - `current_working_memory` (higher priority)
   - `current_goal_context` (highest priority)
4. **UTF-8 Safe Byte Slice**: As last resort, applies direct byte slicing
   - Attempts to find valid JSON boundaries
   - Adds explicit truncation markers
   - Ensures resulting string is valid UTF-8

This approach not only preserves size constraints but maintains the most critical information for decision-making.

For context exceeding token limits, the system implements LLM-based compression:

```
# In _gather_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 summary compression.")
    # Check if summarization tool is available
    if self._find_tool_server(TOOL_SUMMARIZE_TEXT):
        # Strategy: Summarize the potentially longest/most verbose part first
        # Example: Summarize 'core_context' -> 'recent_actions' if it exists and is large
        core_ctx = base_context.get("core_context")
        actions_to_summarize = None
        # ... summarization logic ...
        summary_result = await self._execute_tool_call_internal(
            TOOL_SUMMARIZE_TEXT,
            {
                "text_to_summarize": actions_text,
                "target_tokens": CONTEXT_COMPRESSION_TARGET_TOKENS,
                "prompt_template": "summarize_context_block",
                "context_type": "actions",
                "workflow_id": current_wf_id,
                "record_summary": False
            },
            record_action=False
        )
```

This compression targets the most verbose parts first, generating concise summaries while preserving critical details.

A distinctive feature of the context system is the explicit inclusion of temporal awareness through 'freshness' indicators:

```
retrieval_timestamp = datetime.now(timezone.utc).isoformat()
```

Throughout context gathering, each component is tagged with when it was retrieved:

```
base_context['core_context']['retrieved_at'] = retrieval_timestamp
```

These timestamp indicators serve several critical functions:

1. They enable the LLM to reason about potentially stale information
2. They help prioritize more recent information in decision-making
3. They provide clear signals about the temporal relationship between different context components

By tagging context components with retrieval timestamps, the system creates a time-aware context representation that helps the LLM make more temporally grounded decisions, mimicking human awareness of information recency.

State Persistence System

State Saving

```
async def _save_agent_state(self) -> None:
```

This method implements atomically reliable state persistence designed to survive system crashes:

1. **State Serialization**:
   - Converts dataclass to dictionary with `dataclasses.asdict`
   - Adds timestamp for when state was saved
   - Removes non-serializable fields (`background_tasks`)
   - Converts complex types (`PlanStep`, `defaultdict`) to serializable forms
2. **Atomic File Write**:
   - Creates a process-specific temporary file to prevent collisions

     ```
     tmp_file = self.agent_state_file.with_suffix(f'.tmp_{os.getpid()}')
     ```

   - Uses `os.fsync` to ensure physical disk write

     ```
     os.fsync(f.fileno())
     ```

   - Atomically replaces old file with `os.replace`

     ```
     os.replace(tmp_file, self.agent_state_file)
     ```

3. **Error Handling**:
   - Ensures directory exists before writing
   - Handles serialization errors with fallbacks
   - Cleans up temporary file on write failure
   - Preserves original file if any step fails

This approach ensures state integrity even with process crashes or power failures, providing guaranteed persistence for long-running agents.

State Loading

```
async def _load_agent_state(self) -> None:
```

This method handles robust state restoration:

1. **File Reading**:
   - Checks if state file exists
   - Reads and parses JSON asynchronously
2. **Field Processing**:
   - Iterates through `AgentState` dataclass fields
   - Handles special fields requiring conversion:
     - `current_plan`: Validates and converts to `PlanStep` objects
     - `tool_usage_stats`: Reconstructs `defaultdict` structure
     - `goal_stack`: Validates structure and content
3. **Validation and Correction**:
   - Ensures thresholds are within `MIN/MAX` bounds
   - Verifies goal stack consistency
   - Ensures `current_goal_id` points to a goal in the stack
   - Checks for unknown fields in saved state
4. **Error Recovery**:
   - Handles file not found gracefully
   - Recovers from JSON decoding errors
   - Falls back to defaults on structure mismatches
   - Ensures critical fields are always initialized

This implementation balances flexibility with safety, allowing for schema evolution while maintaining stability.

Shutdown Mechanisms

The system implements comprehensive shutdown handling:

```
async def shutdown(self) -> None:
```

This method provides graceful termination:

1. Sets the shutdown event to signal loops and tasks
2. Waits for background tasks to complete or cancel
3. Saves the final agent state
4. Logs completion of shutdown process

The shutdown signal propagates throughout the system:

1. **Main Loop Detection**:

   ```
   # In run method
   if self._shutdown_event.is_set():
       break
   ```

2. **Background Task Cancellation**:

```python
async def _cleanup_background_tasks(self) -> None:
    # ... task gathering ...
    for t in tasks_to_cleanup:
        if not t.done():
            t.cancel()
    # ... wait for completion ...
```

3. **Retry Abortion**:

```python
# In _with_retries
if self._shutdown_event.is_set():
    self.logger.warning(f"Shutdown signaled during retry wait for {coro_fun.__name__}. Aborting
    retry.")
    raise asyncio.CancelledError(f"Shutdown during retry for {coro_fun.__name__}") from last_exception
```

4. **Periodic Task Termination**:

```python
# In _run_periodic_tasks
if self._shutdown_event.is_set():
    self.logger.info("Shutdown detected during periodic tasks, aborting remaining.")
    break
```

The design ensures all operations check for shutdown signals regularly, maintaining responsiveness while allowing for clean termination.

Signal Handling Integration

The system integrates with OS signals via asyncio:

```python
# In run_agent_process

# Define the signal handler function
def signal_handler_wrapper(signum):
    signal_name = signal.Signals(signum).name
    log.warning(f"Signal {signal_name} received. Initiating graceful shutdown.")
    # Set the event to signal other tasks
    stop_event.set()
    # Trigger the agent's internal shutdown method asynchronously
    if agent_loop_instance:
         asyncio.create_task(agent_loop_instance.shutdown())

# Register the handler for SIGINT (Ctrl+C) and SIGTERM
for sig in [signal.SIGINT, signal.SIGTERM]:
    try:
        loop.add_signal_handler(sig, signal_handler_wrapper, sig)
        log.debug(f"Registered signal handler for {sig.name}")
    except ValueError:
        log.debug(f"Signal handler for {sig.name} may already be registered.")
    except NotImplementedError:
        log.warning(f"Signal handling for {sig.name} not supported on this platform.")
```

This implementation:

1. Registers handlers for OS termination signals
2. Converts signals to shutdown events
3. Triggers the agent's graceful shutdown sequence
4. Handles platform-specific limitations
5. Prevents double registration errors

During execution, the system uses a race mechanism to handle shutdown:

```python
# Create tasks for the main agent run and for waiting on the stop signal
run_task = asyncio.create_task(agent_loop_instance.run(goal=goal, max_loops=max_loops))
stop_task = asyncio.create_task(stop_event.wait())

# Wait for either the agent run to complete OR the stop signal to be received
```

```
done, pending = await asyncio.wait(
    {run_task, stop_task},
    return_when=asyncio.FIRST_COMPLETED
)
```

This approach ensures the agent responds promptly to shutdown signals without polling.

Driver and Entry Point Functionality

Main Driver Function

```
async def run_agent_process(
    mcp_server_url: str,
    anthropic_key: str,
    goal: str,
    max_loops: int,
    state_file: str,
    config_file: Optional[str],
) -> None:
```

This function manages the complete agent lifecycle:

1. **Setup Phase**:
    - Instantiates MCPClient with server URL and config
    - Configures Anthropic API key
    - Sets up MCP connections and interactions
    - Creates AgentMasterLoop instance
    - Registers signal handlers for clean termination
2. **Agent Initialization**:
    - Calls agent.initialize() to load state and prepare tools
    - Exits early if initialization fails
3. **Execution Phase**:
    - Creates concurrent tasks for agent.run() and stop_event.wait()
    - Races them with asyncio.wait()
    - Handles both normal completion and signal interruption
4. **Termination Phase**:
    - Ensures agent shutdown method is called
    - Closes MCP client connections
    - Sets appropriate exit code based on outcome
    - Cleans up resources before exiting

The function includes comprehensive error handling at each stage, with proper error propagation and logging.

Entry Point

```
if __name__ == "__main__":
    # Load configuration from environment variables or defaults
    MCP_SERVER_URL = os.environ.get("MCP_SERVER_URL", "http://localhost:8013")
    ANTHROPIC_API_KEY = os.environ.get("ANTHROPIC_API_KEY")
    AGENT_GOAL = os.environ.get(
        "AGENT_GOAL",
        "Create workflow 'Tier 3 Test': Research Quantum Computing impact on Cryptography.",
    )
    MAX_ITER = int(os.environ.get("MAX_ITERATIONS", "30"))
    STATE_FILE = os.environ.get("AGENT_STATE_FILE", AGENT_STATE_FILE)
```

```python
    CONFIG_PATH = os.environ.get("MCP_CLIENT_CONFIG")

    # Validate essential configuration
    if not ANTHROPIC_API_KEY:
        print("  ERROR: ANTHROPIC_API_KEY missing in environment variables.")
        sys.exit(1)
    if not MCP_CLIENT_AVAILABLE:
        print("  ERROR: MCPClient dependency missing.")
        sys.exit(1)

    # Display configuration being used before starting
    print(f"--- {AGENT_NAME} ---")
    print(f"Memory System URL: {MCP_SERVER_URL}")
    print(f"Agent Goal: {AGENT_GOAL}")
    print(f"Max Iterations: {MAX_ITER}")
    print(f"State File: {STATE_FILE}")
    print(f"Client Config: {CONFIG_PATH or 'Default internal config'}")
    print(f"Log Level: {logging.getLevelName(log.level)}")
    print("Anthropic API Key: Found")
    print("-----------------------------------------")

    # Define and run the main async function
    async def _main() -> None:
        await run_agent_process(
            MCP_SERVER_URL,
            ANTHROPIC_API_KEY,
            AGENT_GOAL,
            MAX_ITER,
            STATE_FILE,
            CONFIG_PATH,
        )

    # Run with asyncio.run() and handle initialization interrupts
    try:
        asyncio.run(_main())
    except KeyboardInterrupt:
        print("\n[yellow]Initial KeyboardInterrupt detected. Exiting.[/yellow]")
        sys.exit(130)
```

This entry point provides:

1. Configuration via environment variables with sensible defaults
2. Validation of critical requirements
3. Transparent display of runtime configuration
4. Clean asyncio execution pattern
5. Initial interrupt handling before signal handlers are registered

Integration Architecture and Workflow

To fully understand how this agent operates in practice, let's examine the complete workflow:

1. **Startup Sequence**:
    - User calls script with goal (CLI or environment variable)
    - System creates MCPClient connection to UMS
    - AgentMasterLoop is initialized with configuration
    - Prior state is loaded if available
    - Signal handlers established for graceful termination
2. **Initial Workflow Creation**:
    - If no active workflow, create one with the specified goal
    - Create initial thought chain for reasoning
    - Create root goal in goal stack
    - Initialize plan with default first step
3. **Think-Act Cycle**:
    - Run periodic cognitive tasks (reflection, consolidation, etc.)

   - Gather comprehensive context (goals, memories, plans, errors)
   - Call LLM for decision (with detailed prompt)
   - Execute decided action (tool call, thought, or completion)
   - Apply plan updates (explicit or heuristic)
   - Save state for persistence
   - Check termination conditions

4. **Goal Management Flow**:
   - LLM can push new sub-goals (decomposing complex tasks)
   - Focus shifts to sub-goal at top of stack
   - When goal is marked complete/failed, it's popped from stack
   - Focus returns to parent goal (or completes if root)
   - Root goal completion signals the agent to finish
5. **Sub-Workflow Management**:
   - Complex tasks may create sub-workflows
   - Each with their own goal stack, thought chains, etc.
   - Completion of sub-workflow returns to parent
   - Links sub-workflow status to corresponding goal
6. **Error Recovery Path**:
   - Tool errors are categorized and captured
   - Error details fed to LLM with recovery strategies
   - Plan updated to handle error condition
   - Consecutive errors tracked with threshold limit
7. **Termination Sequence**:
   - Goal achieved OR max loops reached OR error limit OR signal
   - Cleanup background tasks
   - Save final state
   - Close connections
   - Exit with appropriate code

Prompt Engineering as Cognitive Scaffolding

The system's prompt construction approach represents a sophisticated cognitive scaffolding technique rather than simple context provision:

```
def _construct_agent_prompt(self, goal: str, context: Dict[str, Any]) -> List[Dict[str, Any]]:
```

The prompting approach functions as cognitive scaffolding rather than simple information provision. Beyond presenting factual context, the prompt:

1. **Guides Analytical Process**: Provides a structured framework for problem analysis:

   ```
   '1. Context Analysis: Deeply analyze 'Current Context'...'
   '2. Error Handling: If `last_error_details` exists, **FIRST** reason about...'
   ```

2. **Identifies Cognitive Integration Points**: Explicitly connects context elements:

   ```
   'Note workflow status, errors (`last_error_details` - *pay attention to error `type`*),
   **goal stack (`current_goal_context` -> `goal_stack_summary`) and the `current_goal`**...'
   ```

3. **Provides Recovery Frameworks**: Offers explicit recovery strategies:

```
'Recovery Strategies based on `last_error_details.type`:'
'*   `InvalidInputError`: Review tool schema, arguments, and context...'
```

4. **Creates Decision Frameworks**: Structures the decision process:

```
   '4. Action Decision: Choose **ONE** action based on the *first planned step*
\end{minted}
\end{pageablecode}
\end{enumerate}

This approach creates a 'cognitive partnership' with the LLM, using the prompt t

\subsection*{Complete Integration Example}

Here's how you'd deploy this agent in a real-world scenario:

\begin{enumerate}[label=\arabic*.]
    \item \textbf{Setup Environment}:
    \begin{pageablecode}
    \begin{minted}[fontsize=\scriptsize, breaklines=true, breakanywhere=true, br
   export MCP_SERVER_URL="http://your-memory-server:8013"
   export ANTHROPIC_API_KEY="sk-ant-your-key-here"
   export AGENT_GOAL="Research and summarize recent developments in quantum comp
   export MAX_ITERATIONS=50
   export AGENT_LOOP_LOG_LEVEL=INFO
    \end{minted}
    \end{pageablecode}
    \item \textbf{Run Script}:
    \begin{pageablecode}
    \begin{minted}[fontsize=\scriptsize, breaklines=true, breakanywhere=true, br
   python agent_master_loop.py
    \end{minted}
    \end{pageablecode}
    \item \textbf{Monitor Progress}:
    \begin{itemize}
        \item Console logs show loop iterations, tool calls, errors
        \item State file updated regularly with persistence
        \item UMS records workflow, memories, actions, artifacts
    \end{itemize}
    \item \textbf{Integration with External Systems}:
    \begin{itemize}
        \item Agent can create artifacts via UMS tools
        \item Can incorporate external data sources via appropriate tools
        \item Can trigger downstream processes via workflow status changes
    \end{itemize}
    \item \textbf{Graceful Termination}:
    \begin{itemize}
        \item Press Ctrl+C to send \code{SIGINT}
        \item Agent completes current operation
        \item Saves state for later resumption
        \item Cleanly disconnects from services
    \end{itemize}
    \item \textbf{Resume from Previous State}:
```

```
    \begin{pageablecode}
    \begin{minted}[fontsize=\scriptsize, breaklines=true, breakanywhere=true, br
   # Same environment but potentially different goal
   export AGENT_GOAL="Continue previous research and focus on post-quantum crypt
   python agent_master_loop.py
    \end{minted}
    \end{pageablecode}
\end{enumerate}

\subsection*{Advanced Integration Capabilities}

This agent design supports sophisticated integration patterns:

\begin{enumerate}[label=\arabic*.]
    \item \textbf{Hierarchical Agent Collaboration}:
    \begin{itemize}
        \item Multiple agent instances can create sub-workflows for each other
        \item Parent agents can monitor and coordinate child agents
        \item Complex task decomposition across specialized agents
    \end{itemize}
    \item \textbf{Long-Running/Persistent Agents}:
    \begin{itemize}
        \item State persistence allows resuming after shutdown
        \item Goal stack preserves hierarchical task context
        \item Meta-cognitive processes consolidate knowledge over time
    \end{itemize}
    \item \textbf{Cognitive Framework Integration}:
    \begin{itemize}
        \item Memory levels model human-like episodic/semantic/procedural memory
        \item Working memory with focus mimics human attention
        \item Reflection/consolidation creates higher-level knowledge
    \end{itemize}
    \item \textbf{Adaptive Performance Tuning}:
    \begin{itemize}
        \item Mental momentum bias favors productive periods
        \item Threshold adaptation responds to memory balance
        \item Error rate monitoring triggers course corrections
    \end{itemize}
    \item \textbf{Process Monitoring and Observability}:
    \begin{itemize}
        \item Detailed logging of all operations
        \item State snapshots for debugging/analysis
        \item Tool usage statistics and performance metrics
    \end{itemize}
\end{enumerate}

This comprehensive architecture provides a solid foundation for reliable, sophis

\subsection*{Unified Architectural Overview and Design Philosophy}

The EideticEngine Agent Master Loop represents a sophisticated cognitive archite
```

```
\subsubsection*{Cognitive Science Foundations}

At its core, the EideticEngine employs a cognitive architecture inspired by huma

\begin{enumerate}[label=\arabic*.]
    \item \textbf{Multi-Level Memory System}: The architecture implements three 
    \begin{itemize}
        \item \textbf{Episodic Memory}: Stores specific experiences and observat
        \item \textbf{Semantic Memory}: Contains generalized knowledge abstracte
        \item \textbf{Procedural Memory}: Encodes how-to knowledge and skills
    \end{itemize}
    \item \textbf{Working Memory and Attention}: The system maintains a limited 
    \item \textbf{Goal-Directed Cognition}: The goal stack implementation models
    \item \textbf{Mental Momentum}: The momentum bias system mirrors human cogni
    \item \textbf{Metacognition}: Reflection and consolidation processes simulat
\end{enumerate}

This cognitive foundation isn't merely metaphorical-it shapes the core data stru

It's important to clarify the distinct purposes of the two hierarchical systems 

\subsubsection*{LLM Integration and Prompting Strategy}

The system's interaction with the LLM (Claude from Anthropic) represents a parti
\begin{pageablecode}
\begin{minted}[fontsize=\scriptsize, breaklines=true, breakanywhere=true, breaks
def _construct_agent_prompt(self, goal: str, context: Dict[str, Any]) -> List[Di
\end{minted}
\end{pageablecode}
This method exemplifies advanced prompt engineering techniques:

\begin{enumerate}[label=\arabic*.]
    \item \textbf{Rich Contextual Grounding}: The prompt provides comprehensive 
    \item \textbf{Process Guidance}: Rather than asking open-ended questions, th
    \begin{verbatim}
   "1. Context Analysis: Deeply analyze 'Current Context'..."
   "2. Error Handling: If `last_error_details` exists, **FIRST** reason about...
   "3. Reasoning & Planning:..."
```

5. **Error Recovery Framework**: The system provides explicit recovery strategies based on error types, creating a structured approach to problem-solving:

```
"Recovery Strategies based on `last_error_details.type`:"
"*   `InvalidInputError`: Review tool schema, arguments, and context..."
```

6. **Tool Rationalization**: The prompt highlights essential cognitive tools and provides their schemas, enabling informed tool selection.
7. **Balance of Autonomy and Guidance**: The prompt provides structure without being prescriptive about specific decisions, maintaining the LLM's reasoning capability.

This approach contrasts with simpler prompting strategies that either provide minimal context or overly constrain the model's reasoning. The AML creates a "cognitive partnership" with the LLM, using structured prompts to provide scaffolding for effective reasoning.

Architectural Integration and Information Flow

When we synthesize the various components, an elegant information flow emerges:

1. **Memory → Context → LLM → Decision → Action → Memory** represents the primary cognitive loop
2. **Goal Stack Workflow Stack**: Bidirectional flow between goal and workflow hierarchies maintains task coherence
3. **Background Tasks → Memory**: Asynchronous processes enrich the memory system without blocking the main loop
4. **Meta-Cognition → Feedback → LLM**: Reflection and consolidation outputs feed back into future prompts
5. **Error → Categorization → Recovery Strategy → LLM → Plan Update**: Structured error handling enables resilient execution

These flows create multiple feedback loops that enable sophisticated adaptive behavior:

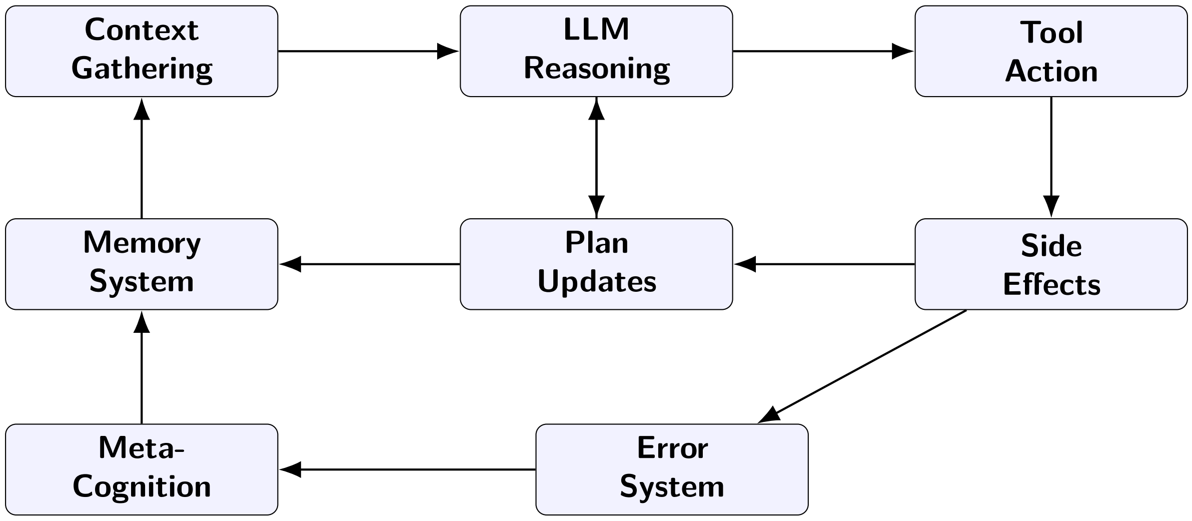


Figure 1: Feedback loops coordinating reasoning, memory, planning, and execution.

This architecture balances synchronous and asynchronous processes, enabling the agent to maintain focus while still performing background cognitive maintenance.

Technical Implementation Excellence

Several aspects of the implementation demonstrate exceptional software engineering practices:

1. **Fault Tolerance and Resilience**:
   - Atomic state persistence with ‘os.replace‘ and ‘os.fsync‘
   - Comprehensive error categorization and recovery
   - Graceful degradation when services are unavailable
   - Retry logic with exponential backoff and jitter
2. **Asynchronous Processing**:
   - Background task management with semaphores
   - Timeout handling to prevent stuck processes
   - Efficient concurrency with asyncio primitives
   - Thread-safe operations for shared state
3. **Resource Management**:
   - Token usage optimization through estimation and compression
   - Memory system maintenance to prevent unbounded growth

    - Adaptive throttling of meta-cognitive processes
    - Efficient context retrieval with fetch/show limits
4. **Extensibility and Modularity**:
    - Clear separation of concerns across methods
    - Consistent error handling patterns
    - Pluggable tool architecture via MCPClient
    - Environment variable configuration for deployment flexibility

These technical qualities enable the system to run reliably in production environments while maintaining adaptability.

The "Cognitive Engine" Metaphor

The EideticEngine name refers to eidetic memory (exceptional recall ability), but the system functions more broadly as a cognitive engine with distinct functional components:

1. **Memory System**: The UMS provides episodic, semantic, and procedural memory storage and retrieval
2. **Attention System**: Working memory optimization and focal point management
3. **Executive Function**: Goal stack and plan management
4. **Metacognitive System**: Reflection and consolidation processes
5. **Reasoning Engine**: LLM integration for decision-making
6. **Action System**: Tool execution framework
7. **Learning System**: Memory consolidation, promotion, and linking
8. **Emotional System**: Mental momentum bias and adaptation thresholds

This metaphor isn't merely aesthetic—it provides a unifying framework for understanding how the components interact. Just as human cognition emerges from the interaction of specialized brain systems, agent intelligence emerges from the interaction of these specialized cognitive components.

Practical Applications and Use Cases

The EideticEngine architecture enables sophisticated applications beyond simple task automation:

1. **Long-Running Research Agents**: The persistence system and goal stack enable extended research projects with complex sub-tasks.
2. **Autonomous Knowledge Workers**: The memory system and metacognitive capabilities support knowledge acquisition, organization, and application.
3. **Adaptive Personal Assistants**: The goal management system enables assistants that maintain context across multiple sessions and adapt to user patterns.
4. **Exploratory Problem Solvers**: The plan-execution cycle with error recovery enables structured exploration of solution spaces.
5. **Multi-Agent Systems**: The sub-workflow capability enables hierarchical collaboration between specialized agents.

These applications leverage the system's distinctive ability to maintain context, learn from experience, and adapt its cognitive processes based on feedback.

Current Limitations and Future Directions

Despite its sophistication, the system has several limitations that suggest future development directions:

1. **LLM Dependence**: The system relies heavily on LLM reasoning quality, inheriting potential biases and limitations.
2. **Tool-Based Action Space**: Actions are limited to available tools, constraining the agent's capabilities.
3. **Single-Agent Focus**: While sub-workflows exist, true multi-agent collaboration isn't fully supported.
4. **Limited Self-Modification**: The agent can't modify its own code or cognitive architecture.
5. **Static Prompt Strategy**: The LLM prompting approach is sophisticated but relatively static.

Future versions might address these limitations through:

- More dynamic prompt engineering based on context
- Enhanced multi-agent coordination protocols
- Greater architectural self-modification capabilities
- Improved hybridization with other AI techniques beyond LLMs

Conclusion: The Agent as a Cognitive System

When we integrate all aspects of our analysis, the EideticEngine Agent Master Loop emerges not just as a technical implementation but as a comprehensive cognitive system. It embodies principles from cognitive science, AI theory, and software engineering to create an agent architecture capable of:

1. **Maintaining Extended Context**: Through its memory systems and state persistence
2. **Learning from Experience**: Via metacognitive feedback loops and memory consolidation
3. **Adaptive Problem Solving**: Through goal decomposition and flexible planning
4. **Resilient Execution**: Via sophisticated error handling and recovery
5. **Self-Reflection**: Through periodic meta-cognitive processes

This cognitive systems approach represents a significant advancement over simpler agent architectures that lack memory, meta-cognition, or goal hierarchies. While still fundamentally powered by an LLM, the EideticEngine creates an execution context that dramatically enhances the LLM's capabilities, enabling more reliable, contextual, and goal-directed behavior.

The system demonstrates how architectural design can complement foundational model capabilities, creating an integrated system greater than the sum of its parts—a true cognitive engine rather than simply an interface to an LLM.

C Appendix C: Unified Memory System (UMS) Code Listing

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

```
"""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.
```

```python
- 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 contextlib
import json
import os
import re
import time
import uuid
from collections import defaultdict
from datetime import datetime, timezone
from enum import Enum
from pathlib import Path
from typing import Any, AsyncIterator, Dict, List, Optional, Tuple

import aiosqlite
import markdown
import numpy as np
from pygments.formatters import HtmlFormatter
from sklearn.metrics.pairwise import cosine_similarity as sk_cosine_similarity

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

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

logger = get_logger("ultimate.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 = 1536  # For the default model
SIMILARITY_THRESHOLD = 0.75

# SQLite optimization pragmas (from cognitive_memory)
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"
]
```

```python
MAX_SEMANTIC_CANDIDATES = int(os.environ.get("MAX_SEMANTIC_CANDIDATES", "500")) # Hard cap for semantic
search candidates

# ======================================================
# 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
# ======================================================

# Note: Using TEXT for IDs (UUIDs) and INTEGER for datetimes (Unix timestamp seconds)

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 ---
CREATE TABLE IF NOT EXISTS workflows (
    workflow_id TEXT PRIMARY KEY,
    title TEXT NOT NULL,
    description TEXT,
    goal TEXT,
    status TEXT NOT NULL,
    created_at INTEGER NOT NULL,
    updated_at INTEGER NOT NULL,
    completed_at INTEGER,
    parent_workflow_id TEXT,
    metadata TEXT,
    last_active INTEGER
);

-- Actions table ---
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 INTEGER NOT NULL,
    completed_at INTEGER,
    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 ---
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 INTEGER 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 INTEGER 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
);

-- Embeddings table (Create before memories which references it) ---
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
    dimension INTEGER NOT NULL,        --  Dimension of the embedding vector
    created_at INTEGER NOT NULL
    -- Cannot add FK to memories yet, as it doesn't exist. Will be added via memories FK.
);

-- Memories table (Create before thoughts which references it) ---
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,
    updated_at INTEGER NOT NULL,
    last_accessed INTEGER,
    access_count INTEGER DEFAULT 0,
    ttl INTEGER DEFAULT 0,             -- TTL in seconds (0 = permanent)
    embedding_id TEXT,                 -- FK to embeddings table
    action_id TEXT,                    -- *** FK: Action associated with this memory ***
    thought_id TEXT,                   -- *** FK: Thought associated with this memory - REMOVED INLINE, ADDED
    VIA ALTER LATER ***
    artifact_id TEXT,                  -- *** 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,
    FOREIGN KEY (artifact_id) REFERENCES artifacts(artifact_id) ON DELETE SET NULL
    -- REMOVED INLINE FK: FOREIGN KEY (thought_id) REFERENCES thoughts(thought_id) ON DELETE SET NULL
    DEFERRABLE INITIALLY DEFERRED;
);

-- Add back reference from embeddings to memories now that memories exists
ALTER TABLE embeddings ADD CONSTRAINT fk_embeddings_memory FOREIGN KEY (memory_id) REFERENCES
memories(memory_id) ON DELETE CASCADE;


-- Thoughts table (Create after memories)
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 INTEGER 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,           -- *** FK: Memory entry this thought relates to - REMOVED INLINE,
    ADDED VIA ALTER LATER ***
    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
    -- REMOVED INLINE FK: FOREIGN KEY (relevant_memory_id) REFERENCES memories(memory_id) ON DELETE SET NULL
    DEFERRABLE INITIALLY DEFERRED;
);


-- Memory links table ---
CREATE TABLE IF NOT EXISTS memory_links (
    link_id TEXT PRIMARY KEY,
    source_memory_id TEXT NOT NULL,
    target_memory_id TEXT NOT NULL,
    link_type TEXT NOT NULL,          -- Uses LinkType enum
    strength REAL DEFAULT 1.0,
    description TEXT,
    created_at INTEGER NOT NULL,
    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)
);

-- Cognitive states table (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 INTEGER NOT NULL,
    is_latest BOOLEAN NOT NULL,
    FOREIGN KEY (workflow_id) REFERENCES workflows(workflow_id) ON DELETE CASCADE
);

-- Reflections table (for meta-cognitive analysis)
CREATE TABLE IF NOT EXISTS reflections (
    reflection_id TEXT PRIMARY KEY,
    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,
    referenced_memories TEXT,          -- JSON array of memory_ids
    FOREIGN KEY (workflow_id) REFERENCES workflows(workflow_id) ON DELETE CASCADE
);

-- Memory operations log (for auditing/debugging)
CREATE TABLE IF NOT EXISTS memory_operations (
    operation_log_id TEXT PRIMARY KEY,
    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 ---
CREATE TABLE IF NOT EXISTS tags (
    tag_id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL UNIQUE,
    description TEXT,
    category TEXT,
    created_at INTEGER NOT NULL
);

-- Junction Tables for Tags ---
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 ---
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 INTEGER 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 ---
-- 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);
-- 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);
CREATE INDEX IF NOT EXISTS idx_thoughts_relevant_memory ON thoughts(relevant_memory_id); -- Index still
useful
-- Memory indices
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);
CREATE INDEX IF NOT EXISTS idx_memories_thought_id ON memories(thought_id); -- Index still useful
CREATE INDEX IF NOT EXISTS idx_memories_artifact_id ON memories(artifact_id);
-- 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);
-- Embedding indices
CREATE INDEX IF NOT EXISTS idx_embeddings_memory_id ON embeddings(memory_id); -- Index the FK
CREATE INDEX IF NOT EXISTS idx_embeddings_dimension ON embeddings(dimension); -- Index for dimension
filtering
-- 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);

-- Virtual table for memories ---
CREATE VIRTUAL TABLE IF NOT EXISTS memory_fts USING fts5(
    content, description, reasoning, tags,
    workflow_id UNINDEXED,
    memory_id UNINDEXED,
    content='memories',
    content_rowid='rowid',
    tokenize='porter unicode61'
);

-- Triggers to keep virtual 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;

-- Deferrable Circular Foreign Key Constraints for thoughts <-> memories
-- Execute after tables are created and within an explicit transaction
BEGIN IMMEDIATE TRANSACTION; -- Use IMMEDIATE for exclusive lock during schema change
PRAGMA defer_foreign_keys = ON; -- Enable deferral specifically for this transaction

ALTER TABLE thoughts ADD CONSTRAINT fk_thoughts_memory
    FOREIGN KEY (relevant_memory_id) REFERENCES memories(memory_id)
    ON DELETE SET NULL DEFERRABLE INITIALLY DEFERRED;

ALTER TABLE memories ADD CONSTRAINT fk_memories_thought
    FOREIGN KEY (thought_id) REFERENCES thoughts(thought_id)
    ON DELETE SET NULL DEFERRABLE INITIALLY DEFERRED;

COMMIT; -- Commit the transaction containing the PRAGMA and ALTERs
"""

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

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

    _instance: Optional[aiosqlite.Connection] = None # Added type hint for clarity
    _lock = asyncio.Lock()
    _db_path_used: Optional[str] = None
    _init_lock_timeout = 15.0 # Configurable timeout in seconds

    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 _initialize_instance(self) -> aiosqlite.Connection:
        """Handles the actual creation and setup of the database connection."""
        logger.info(f"Connecting to database: {self.db_path}", emoji_key="database")
        conn = await aiosqlite.connect(
            self.db_path,
            timeout=CONNECTION_TIMEOUT # Use timeout from cognitive_memory
            # isolation_level=ISOLATION_LEVEL # aiosqlite handles transactions differently
        )
        conn.row_factory = aiosqlite.Row

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

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

        # Initialize schema if needed
        # Check if tables exist before running the full script
        cursor = await conn.execute("SELECT name FROM sqlite_master WHERE type='table' AND name='workflows'")
        table_exists = await cursor.fetchone()
        await cursor.close() # Explicitly close cursor after fetch
        if not table_exists:
            logger.info("Database schema not found. Initializing...", emoji_key="gear")
            # Ensure foreign keys are enabled before schema execution for ALTER TABLE
            await conn.execute("PRAGMA foreign_keys = ON;")
            # Use executescript for potentially multi-statement SCHEMA_SQL
            # NOTE: executescript implicitly commits before and after execution
            # This is generally okay for schema setup but problematic for transactions.
            # The explicit transaction for DEFERRABLE FKs in SCHEMA_SQL is handled separately.
            await conn.executescript(SCHEMA_SQL)
            # No explicit commit needed after executescript typically
            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 conn.execute("PRAGMA foreign_keys = ON;")

        # Set the path used for this instance *after* successful connection
        DBConnection._db_path_used = self.db_path
        return conn

    async def __aenter__(self) -> aiosqlite.Connection:
        """Acquires the singleton database connection instance."""
        # 1. Quick check without lock
        instance = DBConnection._instance
        if instance is not None:
            # Path consistency check for singleton reuse
            if self.db_path != DBConnection._db_path_used:
                logger.error(f"DBConnection singleton mismatch: Already initialized with path
                '{DBConnection._db_path_used}', but requested '{self.db_path}'.")
                raise RuntimeError(f"DBConnection singleton initialized with path
                '{DBConnection._db_path_used}', requested '{self.db_path}'")
            # Ensure foreign keys are enabled for this specific use of the connection
            # Doing this on every enter ensures it's set for the current operation context
            await instance.execute("PRAGMA foreign_keys = ON;")
            return instance

        # 2. Acquire lock with timeout only if instance might need initialization
        try:
            # Use asyncio.timeout for the lock acquisition itself
            async with asyncio.timeout(DBConnection._init_lock_timeout):
                async with DBConnection._lock:
```

```
                # 3. Double-check instance after acquiring lock
                if DBConnection._instance is None:
                    # Call the separate initialization method
                    DBConnection._instance = await self._initialize_instance()
                # Re-check path consistency inside lock to handle race condition if multiple threads tried
                init
                elif self.db_path != DBConnection._db_path_used:
                     logger.error(f"DBConnection singleton mismatch detected inside lock: Already
                     initialized with path '{DBConnection._db_path_used}', but requested
                     '{self.db_path}'.")
                     raise RuntimeError(f"DBConnection singleton initialized with path
                     '{DBConnection._db_path_used}', requested '{self.db_path}'")

    except asyncio.TimeoutError:
          # Log timeout error and raise a ToolError
          logger.error(f"Timeout acquiring DB initialization lock after
          {DBConnection._init_lock_timeout}s. Possible deadlock or hang.", emoji_key="alarm_clock")
          raise ToolError("Database initialization timed out.") from None
    except Exception as init_err:
          # Catch potential errors during _initialize_instance
          logger.error(f"Error during database initialization: {init_err}", exc_info=True, emoji_key="x")
          # Ensure instance is None if initialization failed
          DBConnection._instance = None
          DBConnection._db_path_used = None
          raise ToolError(f"Database initialization failed: {init_err}") from init_err

    # Ensure FKs enabled for the first use after initialization and return the instance
    # Note: _initialize_instance also sets PRAGMA foreign_keys=ON, but setting it again here
    # ensures it's applied for the context manager's immediate use.
    await DBConnection._instance.execute("PRAGMA foreign_keys = ON;")
    return DBConnection._instance

async def __aexit__(self, exc_type, exc_val, exc_tb):
    """Releases the connection context (but doesn't close singleton).

    Propagates exceptions that occurred within the context.
    """
    # Note: This context manager manages access, not the lifecycle of the singleton connection.
    # Closing is handled by the explicit close_connection method.
    # The transaction manager context handles commit/rollback.
    if exc_type is not None:
          # Log the error that occurred *within* the 'async with DBConnection(...)' block
          logger.error(f"Database error occurred within DBConnection context: {exc_val}",
          exc_info=(exc_type, exc_val, exc_tb))
          # Re-raise the exception to notify the caller
          raise exc_val
    pass # If no exception, just pass (commit/rollback handled by transaction manager)

@classmethod
async def close_connection(cls):
    """Closes the singleton database connection if it exists.

    This method should be called explicitly by the application during shutdown
    to ensure resources are released cleanly.
    """
    if cls._instance:
        async with cls._lock: # Ensure exclusive access for closing
            if cls._instance: # Double check after lock
                logger.info("Attempting to close database connection.", emoji_key="lock")
                try:
                    await cls._instance.close()
                    logger.success("Database connection closed successfully.",
                    emoji_key="white_check_mark")
                except Exception as e:
                    logger.error(f"Error closing database connection: {e}", exc_info=True)
                finally:
                    # Ensure the instance reference is cleared even if close fails
                    cls._instance = None
                    cls._db_path_used = None
            else:
                 logger.info("Database connection was closed by another task while waiting for lock.")
    else:
        logger.info("No active database connection instance to close.")

# --- NEW TRANSACTION MANAGER ---
@contextlib.asynccontextmanager
async def transaction(self) -> AsyncIterator[aiosqlite.Connection]:
    """Provides an atomic transaction block using the singleton connection."""
```

```python
        conn = await self.__aenter__() # Acquire the connection instance
        try:
            # Explicitly BEGIN transaction. aiosqlite defaults might differ.
            # Using DEFERRED is generally fine unless immediate locking is needed.
            await conn.execute("BEGIN DEFERRED TRANSACTION")
            logger.debug("DB Transaction Started.")
            yield conn # Provide the connection to the 'async with' block
        except Exception as e:
            logger.error(f"Exception during transaction, rolling back: {e}", exc_info=True)
            await conn.rollback()
            logger.warning("DB Transaction Rolled Back.", emoji_key="rewind")
            raise # Re-raise the exception after rollback
        else:
            await conn.commit()
            logger.debug("DB Transaction Committed.")
        finally:
            # __aexit__ for the base DBConnection doesn't close the connection,
            # so we don't need to call it explicitly here. The transaction is finished.
            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

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


# =====================================================
# Utilities
# =====================================================

def to_iso_z(ts: float) -> str:        # helper   ISO‑8601 with trailing "Z"
    return (
        datetime.fromtimestamp(ts, tz=timezone.utc)
        .isoformat(timespec="seconds")
        .replace("+00:00", "Z")
    )
```

```python
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.

        Args:
            obj: The Python object to serialize.

        Returns:
            A JSON string representation, or None if the input is None.
            Returns a specific error JSON structure if serialization fails or
            if the resulting JSON string exceeds MAX_TEXT_LENGTH.
        """
        if obj is None:
            return None

        json_str = None # Initialize variable

        try:
            # Attempt direct JSON serialization with reasonable defaults
            # Use default=str as a basic fallback for common non-serializable types like datetime
            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 JSON serialization failed for type {type(obj)}: {e}. Trying fallback.")
            try:
                # Attempt a fallback using string representation
                fallback_repr = str(obj)
                # Ensure fallback doesn't exceed limits either, using robust UTF-8 handling
                fallback_bytes = fallback_repr.encode('utf-8')
                if len(fallback_bytes) > MAX_TEXT_LENGTH:
                    # Truncate the byte representation
                    truncated_bytes = fallback_bytes[:MAX_TEXT_LENGTH]
                    # Decode back to string, replacing invalid byte sequences caused by truncation
                    truncated_repr = truncated_bytes.decode('utf-8', errors='replace')

                    # Optional refinement: Check if the last character is the replacement char (U+FFFD)
                    # If so, try truncating one byte less to avoid splitting a multi-byte char right at the
                    end.
                    # This is a heuristic and might not always be perfect but can improve readability.
                    if truncated_repr.endswith('\ufffd') and MAX_TEXT_LENGTH > 1:
                         # Try decoding one byte less
                         shorter_repr = fallback_bytes[:MAX_TEXT_LENGTH-1].decode('utf-8', errors='replace')
                         # If the shorter version *doesn't* end with the replacement character, use it.
                         if not shorter_repr.endswith('\ufffd'):
                              truncated_repr = shorter_repr

                    truncated_repr += "[TRUNCATED]" # Add ellipsis to indicate truncation
                    logger.warning(f"Fallback string representation truncated for type {type(obj)}.")
                else:
                    # No truncation needed for the fallback string itself
                    truncated_repr = fallback_repr

                # Create the JSON string containing the error and the (potentially truncated) fallback
                json_str = json.dumps({
                    "error": f"Serialization failed for type {type(obj)}.",
                    "fallback_repr": truncated_repr # Store the safely truncated string representation
                }, ensure_ascii=False)

            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)
                }, ensure_ascii=False)

        # --- Check final length AFTER serialization attempt (success or fallback) ---
        # Ensure json_str is assigned before checking length
        if json_str is None:
             # This case should theoretically not be reached if the logic above is sound,
             # but added as a safeguard. It implies an unexpected path where serialization
             # didn't succeed but also didn't fall into the error handlers properly.
             logger.error(f"Internal error: json_str is None after serialization attempt for object of type
             {type(obj)}")
             return json.dumps({
                 "error": "Internal serialization error occurred.",
                 "original_type": str(type(obj))
             }, ensure_ascii=False)


        # Check final length against MAX_TEXT_LENGTH (bytes)
        final_bytes = json_str.encode('utf-8')
        if len(final_bytes) > MAX_TEXT_LENGTH:
            # If the generated JSON (even if it's an error JSON from fallback) is too long,
            # return a standard "too long" error marker with a preview.
            logger.warning(f"Serialized JSON string exceeds max length ({MAX_TEXT_LENGTH} bytes). Returning
            truncated indicator.")
            # Provide a preview of the oversized JSON string
            preview_str = json_str[:200] + ("..." if len(json_str) > 200 else "")
            return json.dumps({
                "error": "Serialized content exceeded maximum length.",
                "original_type": str(type(obj)),
                "preview": preview_str # Provide a small preview of the oversized content
            }, ensure_ascii=False)
        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 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
    def _validate_sql_identifier(identifier: str, identifier_type: str = "column/table") -> str:
        """Validates a string intended for use as an SQL table or column name.

        Prevents SQL injection by ensuring the identifier only contains
        alphanumeric characters and underscores. Raises ToolInputError if invalid.

        Args:
            identifier: The string to validate.
            identifier_type: A description of what the identifier represents (for error messages).

        Returns:
            The validated identifier if it's safe.

        Raises:
            ToolInputError: If the identifier is invalid.
        """
        # Simple regex: Allows letters, numbers, and underscores. Must start with a letter or underscore.
        # Adjust regex if more complex identifiers (e.g., quoted) are needed, but keep it strict.
        if not identifier or not re.fullmatch(r"^[a-zA-Z_][a-zA-Z0-9_]*$", identifier):
            logger.error(f"Invalid SQL identifier provided: '{identifier}'")
            raise ToolInputError(f"Invalid {identifier_type} name provided. Must be
            alphanumeric/underscore.", param_name=identifier_type)
        # Optional: Check against a known allowlist of tables/columns if possible
        # known_tables = {"actions", "thoughts", "memories", ...}
        # if identifier_type == "table" and identifier not in known_tables:
        #     raise ToolInputError(f"Unknown table name provided: {identifier}", param_name=identifier_type)
        return identifier

    @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).
        """
        # --- Validate dynamic identifiers to prevent SQL injection ---
        validated_table = MemoryUtils._validate_sql_identifier(table, "table")
        validated_parent_col = MemoryUtils._validate_sql_identifier(parent_col, "parent_col")
        # --- End Validation ---
        #
        # --- Concurrency Note ---
        # This read-then-write operation (SELECT MAX + 1, then INSERT) is
        # generally safe within the transaction managed by the DBConnection
        # context manager wrapping the calling tool function. This makes it
        # atomic relative to other *completed* tool calls.
        # However, a theoretical race condition exists if *multiple concurrent*
        # calls to the *same* tool function attempt to get the sequence number
        # for the *exact same parent_id* before the transaction commits.
        # They might both read the same MAX value.
        # In practice, SQLite's isolation levels (especially WAL mode) and the
        # typical single-threaded nature of agent actions within a workflow
        # make this unlikely to cause issues.
        # If duplicate sequence numbers are observed under very high concurrency,
        # consider implementing an explicit asyncio.Lock per parent_id,
        # managed in a shared dictionary within the MemoryUtils class or a
        # dedicated sequence manager. For now, we rely on transaction isolation.
        # --- End Concurrency Note ---

        # Use validated identifiers in the f-string
        sql = f"SELECT MAX(sequence_number) FROM {validated_table} WHERE {validated_parent_col} = ?"
        # Use execute directly on the connection for context management
        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.
            # Access by index as row might be None or a tuple/row object
            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'). Must form valid SQL
            identifiers when combined with '_tags' or '_id'.
        """
        if not tags:
            return # Nothing to do if no tags are provided

        # Validate entity_type first as it forms part of identifiers
        # Allow only specific expected entity types
        allowed_entity_types = {"workflow", "action", "artifact"}
        if entity_type not in allowed_entity_types:
             raise ToolInputError(f"Invalid entity_type for tagging: {entity_type}",
             param_name="entity_type")

        # Define and validate dynamic identifiers
        junction_table_name = f"{entity_type}_tags"
        id_column_name = f"{entity_type}_id"
        validated_junction_table = MemoryUtils._validate_sql_identifier(junction_table_name,
        "junction_table")
        validated_id_column = MemoryUtils._validate_sql_identifier(id_column_name, "id_column")
        # --- End Validation ---

        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_unix = int(time.time())

        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_unix)
            )
            # Retrieve the tag_id (whether newly inserted or existing)
            cursor = await conn.execute("SELECT tag_id FROM tags WHERE name = ?", (tag_name,))
            row = await cursor.fetchone()
            await cursor.close() # Close cursor

            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
            # Use validated identifiers in the f-string
            await conn.executemany(
                f"INSERT OR IGNORE INTO {validated_junction_table} ({validated_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)
            )
        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_list = await embedding_service.create_embeddings(texts=[text])
        if not embedding_list or not embedding_list[0]: # Extra check for empty embedding
             logger.warning(f"Failed to generate embedding for memory {memory_id}")
             return None
        embedding_array = np.array(embedding_list[0], dtype=np.float32) # Ensure consistent dtype
        if embedding_array.size == 0:
             logger.warning(f"Generated embedding is empty for memory {memory_id}")
             return None

        # Get the embedding dimension
        embedding_dimension = embedding_array.shape[0]

        # 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, dimension, created_at)
            VALUES (?, ?, ?, ?, ?, ?)
            ON CONFLICT(memory_id) DO UPDATE SET
                id = excluded.id,
                model = excluded.model,
                embedding = excluded.embedding,
                dimension = excluded.dimension,
                created_at = excluded.created_at
            """,
            (
                embedding_db_id,
                memory_id,
                model_used,
                embedding_bytes,
                embedding_dimension,
                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} (Dim: {embedding_dimension}) 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
) -> List[Tuple[str, float]]:
    """Finds memories with similar semantic meaning using embeddings stored in SQLite.
       Filters by workflow, level, type, dimension, 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_list = await embedding_service.create_embeddings(texts=[query_text])
        if not query_embedding_list or not query_embedding_list[0]: # Extra check
            logger.warning(f"Failed to generate query embedding for: '{query_text[:50]}...'")
            return []
        query_embedding = np.array(query_embedding_list[0], dtype=np.float32) # Ensure consistent dtype
        if query_embedding.size == 0:
            logger.warning(f"Generated query embedding is empty for: '{query_text[:50]}...'")
            return []

        query_dimension = query_embedding.shape[0]
        query_embedding_2d = query_embedding.reshape(1, -1) # Reshape for scikit-learn

        # 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 e.dimension = ?
        """
        params: List[Any] = [query_dimension]

        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
        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 (only those with matching dimension)
        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(f"No candidate memories found matching filters (including dimension
            {query_dimension}) for semantic search.")
            return []

        # 4. Calculate similarities for candidates
        similarities: List[Tuple[str, float]] = []
        for memory_id, embedding_bytes in candidates:
            try:
                # Deserialize embedding from bytes
                memory_embedding = np.frombuffer(embedding_bytes, dtype=np.float32)
                if memory_embedding.size == 0:
                    logger.warning(f"Skipping empty embedding blob for memory {memory_id}")
                    continue

                # Reshape for scikit-learn compatibility
                memory_embedding_2d = memory_embedding.reshape(1, -1)

                # --- Safety Check: Verify dimensions again (should match due to SQL filter) ---
                # This primarily guards against database corruption or schema inconsistencies.
                if query_embedding_2d.shape[1] != memory_embedding_2d.shape[1]:
                    logger.warning(f"Dimension mismatch detected for memory {memory_id} (Query:
                    {query_embedding_2d.shape[1]}, DB: {memory_embedding_2d.shape[1]}) despite DB filter.
                    Skipping.")
                    continue
                # --- End Safety Check ---

                # Calculate cosine similarity
                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 (Dim: {query_dimension}).
        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. **Raises ToolError if
    the embedding service fails to initialize or is non-functional.**

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

    Returns:
        Initialization status dictionary (only if successful).
        {
            "success": true,
            "message": "Unified Memory System initialized successfully.",
            "db_path": "/path/to/unified_agent_memory.db",
            "embedding_service_functional": true, # Will always be true if function returns successfully
            "embedding_service_warning": null,
            "processing_time": 0.123
        }

    Raises:
        ToolError: If database initialization fails OR if the EmbeddingService
                   cannot be initialized or lacks a functional client (e.g., missing API key).
    """
    start_time = time.time()
    logger.info("Initializing Unified Memory System...", emoji_key="rocket")
    embedding_service_warning = None # This will now likely be part of the error message

    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
            cursor = await conn.execute("SELECT count(*) FROM workflows")
            _ = await cursor.fetchone()
            await cursor.close() # Close cursor
            # No explicit commit needed here if using default aiosqlite behavior or autocommit
        logger.success("Unified Memory System database connection verified.", emoji_key="database")

        # Attempt to initialize/get the EmbeddingService singleton and VERIFY functionality
        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 (e.g., requires API key)
            if embedding_service.client is not None:
                logger.info("EmbeddingService initialized and functional.", emoji_key="brain")
            else:
                embedding_service_warning = "EmbeddingService client not available (check API key).
                Embeddings disabled."
                logger.error(embedding_service_warning, emoji_key="warning") # Log as error
                # Raise explicit error instead of just returning False status
                raise ToolError(embedding_service_warning)
        except Exception as embed_init_err:
             # This includes the explicit ToolError raised above for missing client
             if not isinstance(embed_init_err, ToolError): # Avoid double wrapping if it was the specific
             client error
                 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)
                 raise ToolError(embedding_service_warning) from embed_init_err
             else:
                 # Re-raise the ToolError directly if it was the specific missing client error
                 raise embed_init_err

        # If we reach here, both DB and Embedding Service are functional
        processing_time = time.time() - start_time
        logger.success("Unified Memory System initialized successfully (DB and Embeddings OK).",
        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": True, # Will always be true if this return is reached
            "embedding_service_warning": None, # No warning if successful
            "processing_time": processing_time
        }
    except Exception as e:
        # This catches errors during DB initialization OR the ToolError raised from embedding failure
        processing_time = time.time() - start_time
        # Ensure it's logged as a critical failure
        logger.error(f"Failed to initialize memory system: {str(e)}", emoji_key="x", exc_info=True,
        time=processing_time)
        # Re-raise as ToolError if it wasn't already one
        if isinstance(e, ToolError):
            raise e
        else:
            # Wrap unexpected DB errors
            raise ToolError(f"Memory system 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.
        Timestamps are returned as ISO 8601 strings.
        {
            "workflow_id": "uuid-string",
            "title": "Workflow Title",
            "description": "...",
            "goal": "...",
            "status": "active",
            "created_at": "iso-timestampZ",
            "updated_at": "iso-timestampZ",
            "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_unix = int(time.time())

    try:
        async with DBConnection(db_path) as conn:
            # Check parent workflow existence if provided
            if parent_workflow_id:
                cursor = await conn.execute("SELECT 1 FROM workflows WHERE workflow_id = ?",
                (parent_workflow_id,))
                parent_exists = await cursor.fetchone()
                await cursor.close() # Close cursor
                if not parent_exists:
                    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_unix, now_unix, parent_workflow_id, metadata_json, now_unix) # *** CHANGED: Use 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_unix)
            )

            # 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_unix)
                )

            # Commit the transaction
            await conn.commit()

            # Prepare the result dictionary, formatting timestamps for output

            result = {
                "workflow_id": workflow_id,
                "title": title,
                "description": description,
                "goal": goal,
                "status": WorkflowStatus.ACTIVE.value,
                "created_at": to_iso_z(now_unix),
                "updated_at": to_iso_z(now_unix),
                "tags": tags or [],
                "primary_thought_chain_id": thought_chain_id,  # default chain ID for the agent
                "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(
    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, adapted).
       Timestamps are returned as ISO 8601 strings.
    """
    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_unix = int(time.time())

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

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

            if status_enum in [WorkflowStatus.COMPLETED, WorkflowStatus.FAILED, WorkflowStatus.ABANDONED]:
                set_clauses += ", completed_at = ?"
                update_params.append(now_unix)

            # Add workflow_id to params for WHERE clause
            update_params.append(workflow_id)

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

            # Add completion message as thought
            if completion_message:
                 cursor = await conn.execute("SELECT thought_chain_id FROM thought_chains WHERE workflow_id =
                 ? ORDER BY created_at ASC LIMIT 1", (workflow_id,))
                 row = await cursor.fetchone()
                 await cursor.close()
                 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_unix)
                     )

            # 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": to_iso_z(now_unix),
                "success": True,
            }

            if status_enum in (
                WorkflowStatus.COMPLETED,
                WorkflowStatus.FAILED,
                WorkflowStatus.ABANDONED,
            ):
                result["completed_at"] = to_iso_z(now_unix)
                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
```

```
# --- 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_unix = int(time.time())

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

            if parent_action_id:
                cursor = await conn.execute("SELECT 1 FROM actions WHERE action_id = ? AND workflow_id = ?",
                (parent_action_id, workflow_id))
                parent_exists = await cursor.fetchone()
                await cursor.close()
                if not parent_exists:
                    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:
```

```
        cursor = await 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))
        thought_exists = await cursor.fetchone()
        await cursor.close()
        if not thought_exists:
            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_unix,
         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 ---
    memory_content = f"Started action [{sequence_number}] '{auto_title}' ({action_type_enum.value}).
    Reasoning: {reasoning}"
    if tool_name:
         memory_content += f" Tool: {tool_name}."
    mem_tags = ["action_start", action_type_enum.value] + (tags or [])
    mem_tags_json = json.dumps(list(set(mem_tags)))

    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) # Memories already use Unix timestamps
    )
    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_unix, now_unix, workflow_id))

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

    # --- Prepare Result (Format timestamp for output) ---
    result = {
        "action_id": action_id,
        "workflow_id": workflow_id,
        "action_type": action_type_enum.value,
        "title": auto_title,
        "tool_name": tool_name,
                "status": ActionStatus.IN_PROGRESS.value,
                "started_at": to_iso_z(now_unix),
                "sequence_number": sequence_number,
                "tags": tags or [],
                "linked_memory_id": memory_id,
                "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)
        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]:
            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:
            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_unix = int(time.time())

    try:
        async with DBConnection(db_path) as conn:
            # --- 1. Verify Action and Get Workflow ID ---
            cursor = await conn.execute("SELECT workflow_id, status FROM actions WHERE action_id = ?",
            (action_id,))
            action_row = await cursor.fetchone()
            await cursor.close()
            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"]
            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.")

            # --- 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_unix, tool_result_json, action_id) # *** Use now_unix ***
            )

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

            # --- 4. Add Conclusion Thought (if provided) ---
            conclusion_thought_id = None
            if conclusion_thought and thought_type_enum:
                cursor = await conn.execute("SELECT thought_chain_id FROM thought_chains WHERE workflow_id =
                ? ORDER BY created_at ASC LIMIT 1", (workflow_id,))
                chain_row = await cursor.fetchone()
                await cursor.close()
                if chain_row:
                    thought_chain_id = chain_row["thought_chain_id"]
                    seq_no = await MemoryUtils.get_next_sequence_number(conn, thought_chain_id, "thoughts",
                    "thought_chain_id")
                    conclusion_thought_id = MemoryUtils.generate_id()
                    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_unix, action_id) # *** Use now_unix ***
                    )
                    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 ---
            cursor = await conn.execute("SELECT memory_id, content FROM memories WHERE action_id = ? AND
            memory_type = ?", (action_id, MemoryType.ACTION_LOG.value))
            memory_row = await cursor.fetchone()
            await cursor.close()
            if memory_row:
                memory_id = memory_row["memory_id"]
                original_content = memory_row["content"]
                update_parts = [f"Completed ({status_enum.value})."]
                if summary:
                    update_parts.append(f"Summary: {summary}")
                if tool_result is not None:
                    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.")
                update_text = " ".join(update_parts)
                new_content = original_content + " " + update_text
                importance_mult = 1.0
                if status_enum == ActionStatus.FAILED:
                    importance_mult = 1.2
                elif status_enum == ActionStatus.SKIPPED:
                    importance_mult = 0.8
                await conn.execute(
                    """
                    UPDATE memories
                    SET content = ?,
                        importance = importance * ?,
                        updated_at = ?
                    WHERE memory_id = ?
                    """,
                    (new_content, importance_mult, now_unix, memory_id)
                )
                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 (Format timestamp for output) ---

            result = {
                "action_id": action_id,
                "workflow_id": workflow_id,
                "status": status_enum.value,
                "completed_at": to_iso_z(now_unix),
                "conclusion_thought_id": conclusion_thought_id,
                "success": True,
                "processing_time": time.time() - start_time,
            }

            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
@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": "action-id-1", "type": "requires"}],
                        "dependent_actions": [{"action_id": "action-id-3", "type": "informs"}]
                    }
                },
                ... 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")

    # Ensure target_action_ids is a list
    target_action_ids = []
    if action_ids:
        if isinstance(action_ids, list):
            target_action_ids = action_ids
        else:
            # Handle potential non-list input gracefully
            logger.warning(f"action_ids provided was not a list ({type(action_ids)}). Attempting to use
            action_id.")
            if action_id:
                target_action_ids = [action_id]
            else:
                raise ToolInputError("action_ids must be a list or action_id must be provided.",
                param_name="action_ids")
    elif action_id:
        target_action_ids = [action_id]

    if not target_action_ids: # Should not happen due to initial check, but safeguard
         raise ToolInputError("No valid action IDs specified.", param_name="action_id")


    try:
        async with DBConnection(db_path) as conn:
            placeholders = ', '.join(['?'] * len(target_action_ids))
            # Ensure the query correctly joins tags and groups
            select_query = f"""
                SELECT a.*, GROUP_CONCAT(DISTINCT t.name) as tags_str
                FROM actions a
                LEFT JOIN action_tags at ON a.action_id = at.action_id
                LEFT JOIN tags t ON at.tag_id = t.tag_id
                WHERE a.action_id IN ({placeholders})
                GROUP BY a.action_id
            """

            actions_result = []
            cursor = await conn.execute(select_query, target_action_ids)
            # Iterate using async for
            async for row in cursor:
                # Convert row to dict for easier manipulation
                action_data = dict(row)

                # Format timestamps
                if action_data.get("started_at"):
                    action_data["started_at"] = to_iso_z(action_data["started_at"])
                if action_data.get("completed_at"):
                    action_data["completed_at"] = to_iso_z(action_data["completed_at"])

                # Process tags
                if action_data.get("tags_str"):
                    action_data["tags"] = action_data["tags_str"].split(',')
                else:
                    action_data["tags"] = []
                action_data.pop("tags_str", None) # Remove the intermediate column

                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"])

                # Include dependencies if requested
                if include_dependencies:
                    action_data["dependencies"] = {"depends_on": [], "dependent_actions": []}
                    # Fetch actions this one depends ON (target_action_id is the dependency)
                    dep_cursor_on = await conn.execute(
                        "SELECT target_action_id, dependency_type FROM dependencies WHERE source_action_id =
                        ?",
                        (action_data["action_id"],)
                    )
                    depends_on_rows = await dep_cursor_on.fetchall()
                    await dep_cursor_on.close()
                    action_data["dependencies"]["depends_on"] = [{"action_id": r["target_action_id"], "type":
                    r["dependency_type"]} for r in depends_on_rows]

                    # Fetch actions that depend ON this one (source_action_id depends on this)
                    dep_cursor_by = await conn.execute(
                        "SELECT source_action_id, dependency_type FROM dependencies WHERE target_action_id =
                        ?",
                        (action_data["action_id"],)
                    )
                    dependent_rows = await dep_cursor_by.fetchall()
                    await dep_cursor_by.close()
                    action_data["dependencies"]["dependent_actions"] = [{"action_id": r["source_action_id"],
                    "type": r["dependency_type"]} for r in dependent_rows]


                actions_result.append(action_data)
            await cursor.close() # Close the main cursor

            if not actions_result:
                action_ids_str = ", ".join(target_action_ids[:5]) + ("..." if len(target_action_ids) > 5 else
                "")
                raise ToolInputError(f"No actions found with IDs: {action_ids_str}", param_name="action_id"
                if action_id else "action_ids")

            processing_time = time.time() - start_time
            logger.info(f"Retrieved details for {len(actions_result)} actions", emoji_key="search",
            time=processing_time)

            result = {
                "actions": actions_result,
                "success": True,
                "processing_time": processing_time
            }
            return result

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

# ======================================================
# Contextual Summarization (Used in Agent Context Compression)
# ======================================================
@with_tool_metrics
@with_error_handling
async def summarize_context_block(
    text_to_summarize: str,
    target_tokens: int = 500,
    context_type: str = "actions",  # "actions", "memories", "thoughts", etc.
    workflow_id: Optional[str] = None,
    provider: str = LLMGatewayProvider.ANTHROPIC.value, # Use enum/constant for default
    model: Optional[str] = "claude-3-5-haiku-20241022", # Default model
    db_path: str = DEFAULT_DB_PATH
) -> Dict[str, Any]:
    """Summarizes a specific block of context for an agent, optimized for preserving key information.

    A specialized version of summarize_text designed specifically for compressing agent context
    blocks like action histories, memory sets, or thought chains. Uses optimized prompting
    based on context_type to preserve the most relevant information for agent decision-making.

    Args:
        text_to_summarize: Context block text to summarize
        target_tokens: Desired length of summary (default 500)
        context_type: Type of context being summarized (affects prompting)
        workflow_id: Optional workflow ID for logging
        provider: (Optional) LLM provider to use (e.g., 'openai', 'anthropic').
                  Default 'anthropic'.
        model: (Optional) Specific LLM model name (e.g., 'gpt-4.1-mini',
               'claude-3-5-haiku-20241022'). If None, uses provider's default.
               Default 'claude-3-5-haiku-20241022'.
        db_path: Path to the SQLite database file

    Returns:
        Dictionary containing the generated summary:
        {
            "summary": "Concise context summary...",
            "context_type": "actions",
            "compression_ratio": 0.25,  # ratio of summary length to original length
            "success": true,
            "processing_time": 0.123
        }

    Raises:
        ToolInputError: If text_to_summarize is empty
        ToolError: If summarization fails or provider is invalid
    """
    start_time = time.time()

    if not text_to_summarize:
        raise ToolInputError("Text to summarize cannot be empty", param_name="text_to_summarize")

    # Select appropriate prompt template based on context type
    if context_type == "actions":
        prompt_template = """
You are an expert context summarizer for an AI agent. Your task is to summarize the following ACTION HISTORY
logs
while preserving the most important information for the agent to maintain situational awareness.

For actions, focus on:
1. Key actions that changed state or produced important outputs
2. Failed actions and their error reasons
3. The most recent 2-3 actions regardless of importance
4. Any actions that created artifacts or memories
5. Sequential relationships between actions

Produce a VERY CONCISE summary that maintains the chronological flow and preserves action IDs
when referring to specific actions. Aim for approximately {target_tokens} tokens.

ACTION HISTORY TO SUMMARIZE:
{text_to_summarize}

CONCISE ACTION HISTORY SUMMARY:
"""
    elif context_type == "memories":
        prompt_template = """
You are an expert context summarizer for an AI agent. Your task is to summarize the following MEMORY ENTRIES
while preserving the most important information for the agent to maintain understanding.

For memories, focus on:
1. High importance memories (importance > 7)
```

```
2. High confidence memories (confidence > 0.8)
3. Insights and facts over observations
4. Memory IDs should be preserved when referring to specific memories
5. Connected memories that form knowledge networks

Produce a VERY CONCISE summary that preserves the key information, high-value insights, and
critical relationships. Aim for approximately {target_tokens} tokens.

MEMORY ENTRIES TO SUMMARIZE:
{text_to_summarize}

CONCISE MEMORY SUMMARY:
"""
    elif context_type == "thoughts":
        prompt_template = """
You are an expert context summarizer for an AI agent. Your task is to summarize the following THOUGHT CHAINS
while preserving the reasoning, decisions, and insights.

For thoughts, focus on:
1. Goals, decisions, and conclusions
2. Key hypotheses and critical reflections
3. The most recent thoughts that may affect current reasoning
4. Thought IDs should be preserved when referring to specific thoughts

Produce a VERY CONCISE summary that captures the agent's reasoning process and main insights.
Aim for approximately {target_tokens} tokens.

THOUGHT CHAINS TO SUMMARIZE:
{text_to_summarize}

CONCISE THOUGHT SUMMARY:
"""
    else:
        # Generic template for other context types
        prompt_template = """
You are an expert context summarizer for an AI agent. Your task is to create a concise summary of the
following text
while preserving the most important information for the agent to maintain awareness and functionality.

Focus on information that is:
1. Recent and relevant to current goals
2. Critical for understanding the current state
3. Containing unique identifiers that need to be preserved
4. Representing significant events, insights, or patterns

Produce a VERY CONCISE summary that maximizes the agent's ability to operate with this reduced context.
Aim for approximately {target_tokens} tokens.

TEXT TO SUMMARIZE:
{text_to_summarize}

CONCISE SUMMARY:
"""

    try:
        # Get provider instance using the function argument
        provider_instance = await get_provider(provider)
        if not provider_instance:
            raise ToolError(f"Failed to initialize provider '{provider}'. Check configuration.")

        # Prepare prompt
        prompt = prompt_template.format(
            text_to_summarize=text_to_summarize,
            target_tokens=target_tokens
        )

        # Use the model parameter passed to the function.
        # The get_provider instance might handle None model by using its default,
        # or we rely on the default value set in the function signature if None is passed.
        model_to_use = model # Pass the model argument (which defaults if None wasn't explicitly passed)

        # Generate summary
        generation_result = await provider_instance.generate_completion(
            prompt=prompt,
            model=model_to_use, # Use the variable holding the desired model
            max_tokens=target_tokens + 50,  # Add some buffer for prompt tokens
            temperature=0.2  # Lower temperature for more deterministic summaries
        )
    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 on app shutdown
    await DBConnection.close_connection()

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

```
        summary_text = generation_result.text.strip()
        if not summary_text:
            raise ToolError("LLM returned empty context summary.")

        # Calculate compression ratio
        # Avoid division by zero if text_to_summarize is empty (although checked earlier)
        original_length = max(1, len(text_to_summarize))
        compression_ratio = len(summary_text) / original_length

        # Log the operation if workflow_id provided
        if workflow_id:
            async with DBConnection(db_path) as conn:
                await MemoryUtils._log_memory_operation(
                    conn, workflow_id, "compress_context", None, None,
                    {
                        "context_type": context_type,
                        "original_length": len(text_to_summarize),
                        "summary_length": len(summary_text),
                        "compression_ratio": compression_ratio,
                        "provider": provider, # Log the provider used
                        "model": model_to_use # Log the model used
                    }
                )
                await conn.commit()

        processing_time = time.time() - start_time
        logger.info(
            f"Compressed {context_type} context: {len(text_to_summarize)} -> {len(summary_text)} chars
(Ratio: {compression_ratio:.2f}, LLM: {provider}/{model_to_use or 'default'})",
            emoji_key="compression", time=processing_time
        )

        return {
            "summary": summary_text,
            "context_type": context_type,
            "compression_ratio": compression_ratio,
            "success": True,
```

D Appendix D: Agent Master Loop (AML) Code Listing

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

```
"""
"""
EideticEngine Agent Master Loop (AML) - v4.1 P1 - GOAL STACK + MOMENTUM
======================================================================

This module implements the core orchestration logic for the EideticEngine
AI agent. It manages the primary think-act cycle, interacts with the
Unified Memory System (UMS) via MCPClient, leverages an LLM (Anthropic Claude)
for decision-making and planning, and incorporates several cognitive functions
inspired by human memory and reasoning.

** V4.1 P1 implements Phase 1 improvements: refined context, adaptive
thresholds, plan validation/repair, structured error handling, robust
background task management, AND adds explicit Goal Stack management and
a "Mental Momentum" bias. **

Key Functionalities:
--------------------
*   **Workflow & Context Management:**
    - Creates, manages, and tracks progress within structured workflows.
    - Supports sub-workflow execution via a workflow stack.
    - **Manages an explicit Goal Stack for hierarchical task decomposition.**
    - Gathers rich, multi-faceted context for the LLM decision-making process, including:
        *   **Current Goal Stack information.**
        *   Current working memory and focal points **(Prioritized)**.
        *   Core workflow context (recent actions, important memories, key thoughts).
        *   Proactively searched memories relevant to the current goal/plan step **(Limited Fetch)**.
        *   Relevant procedural memories (how-to knowledge) **(Limited Fetch)**.
        *   Summaries of memories linked to the current focus **(Limited Fetch)**.
        *   **Freshness indicators** for context components.
    - Implements structure-aware context truncation and optional LLM-based compression.

*   **Planning & Execution:**
    - Maintains an explicit, modifiable plan consisting of sequential steps with dependencies.
    - Allows the LLM to propose plan updates via a dedicated tool or text parsing.
    - Includes a heuristic fallback mechanism to update plan steps based on action outcomes if the LLM
doesn't explicitly replan.
    - **Validates plan steps and detects dependency cycles.**
    - Checks action prerequisites (dependencies) before execution.
```

```
            "dependency_id": 123, # Auto-incremented ID
            "created_at": "iso-timestamp",
            "success": true,
            "processing_time": 0.04
        }
```

```
    - Executes tools via the MCPClient, handling server lookup and argument injection.
    - Records detailed action history (start, completion, arguments, results, dependencies).

*   **LLM Interaction & Reasoning:**
    - Constructs detailed prompts for the LLM, providing comprehensive context, tool schemas, and cognitive
    instructions.
    - **Prompts explicitly guide analysis of working memory, goal stack, and provide error recovery
    strategies.**
    - Parses LLM responses to identify tool calls, textual reasoning (recorded as thoughts), or goal
    completion signals.
    - Manages dedicated thought chains for recording the agent's reasoning process.

*   **Cognitive & Meta-Cognitive Processes:**
    - **Memory Interaction:** Stores, updates, searches (semantic/hybrid/keyword), and links memories in the
    UMS.
    - **Working Memory Management:** Retrieves, optimizes (based on relevance/diversity), and automatically
    focuses working memory via UMS tools.
    - **Goal Management:** Uses UMS tools to push new sub-goals onto the stack and mark goals as
    completed/failed.
    - **Background Cognitive Tasks:** Initiates asynchronous tasks **with timeouts and concurrency limits
    (semaphore)** for:
        *   Automatic semantic linking of newly created/updated memories.
        *   Checking and potentially promoting memories to higher cognitive levels (e.g., Episodic ->
        Semantic) based on usage/confidence.
    - **Periodic Meta-cognition:** Runs scheduled tasks based on loop intervals or success counters:
        *   **Reflection:** Generates analysis of progress, gaps, strengths, or plans using an LLM.
        *   **Consolidation:** Synthesizes information from multiple memories into summaries, insights, or
        procedures using an LLM.
        *   **Adaptive Thresholds:** Dynamically adjusts the frequency of reflection/consolidation based on
        agent performance (e.g., error rates, **memory statistics, trends, goal progress stability**) **with
        enhanced heuristics and dampening**. **Includes "Mental Momentum" bias.**
    - **Maintenance:** Periodically deletes expired memories.

*   **State & Error Handling:**
    - Persists the complete agent runtime state (workflow, **goal stack**, plan, counters, thresholds)
    atomically to a JSON file for resumption.
    - Implements retry logic with backoff for potentially transient tool failures (especially for idempotent
    operations).
    - Tracks consecutive errors and halts execution if a limit is reached.
    - Provides detailed, **categorized** error information back to the LLM for recovery attempts.
    - Handles graceful shutdown via system signals (SIGINT, SIGTERM).

"""
from __future__ import annotations

import asyncio
import copy
import dataclasses
import json
import logging
import math
import os
import random
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

import aiofiles
from anthropic import APIConnectionError, APIStatusError, AsyncAnthropic, RateLimitError  # Correct imports
from pydantic import BaseModel, Field, ValidationError

try:
    # Note: Import all potentially used enums/classes from MCPClient for clarity
    from mcp_client import (
        ActionStatus,
        ActionType,
        LinkType,
        MCPClient,
        MemoryLevel,
        MemoryType,
        MemoryUtils,
        ThoughtType,
        ToolError,
        ToolInputError,
        WorkflowStatus,
    )

    MCP_CLIENT_AVAILABLE = True
    log = logging.getLogger("AgentMasterLoop")
    # Bootstrap logger if MCPClient didn't configure it
    if not logging.root.handlers and not log.handlers:
        logging.basicConfig(
            level=logging.INFO,
            format="%(asctime)s - %(name)s - %(levelname)s - %(message)s",
        )
        log = logging.getLogger("AgentMasterLoop")
        log.warning("MCPClient did not configure logger - falling back.")
    log.info("Successfully imported MCPClient and required components.")
except ImportError as import_err:
    # Critical error if MCPClient cannot be imported
    print(f" CRITICAL ERROR: Could not import MCPClient: {import_err}")
    sys.exit(1)

# ---------------------------------------------------------------------------
# Runtime log-level can be raised via AGENT_LOOP_LOG_LEVEL=DEBUG, etc.
# ---------------------------------------------------------------------------
LOG_LEVEL_ENV = os.environ.get("AGENT_LOOP_LOG_LEVEL", "INFO").upper()
log.setLevel(getattr(logging, LOG_LEVEL_ENV, logging.INFO))
if log.level <= logging.DEBUG:
    log.info("Verbose logging enabled for Agent loop.")

# ===========================================================================
# CONSTANTS
# ===========================================================================
# File for saving/loading agent state, versioned for this implementation phase
AGENT_STATE_FILE = "agent_loop_state_v4.1_p1_goalstack_momentum.json" # Updated filename
# Agent identifier used in prompts and logging
AGENT_NAME = "EidenticEngine4.1-P1-GoalStackMomentum" # Updated agent name
# Default LLM model string (can be overridden by environment or config)
MASTER_LEVEL_AGENT_LLM_MODEL_STRING = "claude-3-7-sonnet-20250219" # Use the confirmed model

# ---------------- meta-cognition thresholds ----------------
# Base thresholds for triggering reflection and consolidation, adjustable via environment
BASE_REFLECTION_THRESHOLD = int(os.environ.get("BASE_REFLECTION_THRESHOLD", "7"))
BASE_CONSOLIDATION_THRESHOLD = int(os.environ.get("BASE_CONSOLIDATION_THRESHOLD", "12"))
# Minimum and maximum bounds for adaptive thresholds to prevent extreme values
MIN_REFLECTION_THRESHOLD = 3
MAX_REFLECTION_THRESHOLD = 15
MIN_CONSOLIDATION_THRESHOLD = 5
MAX_CONSOLIDATION_THRESHOLD = 25
# Dampening factor for threshold adjustments (e.g., 0.75 means apply 75% of calculated change)
THRESHOLD_ADAPTATION_DAMPENING = float(os.environ.get("THRESHOLD_DAMPENING", "0.75"))
# Positive bias added to thresholds during "Mental Momentum" (low errors)
MOMENTUM_THRESHOLD_BIAS_FACTOR = 1.2 # e.g., multiply base adjustment by this if momentum is high

# ---------------- interval constants (in loop iterations) ----------------
# How often to run working memory optimization and auto-focus checks
OPTIMIZATION_LOOP_INTERVAL = int(os.environ.get("OPTIMIZATION_INTERVAL", "8"))
# How often to check recently accessed memories for potential level promotion
MEMORY_PROMOTION_LOOP_INTERVAL = int(os.environ.get("PROMOTION_INTERVAL", "15"))
# How often to compute memory statistics and adapt thresholds
STATS_ADAPTATION_INTERVAL = int(os.environ.get("STATS_ADAPTATION_INTERVAL", "10"))
# How often to run maintenance tasks like deleting expired memories
MAINTENANCE_INTERVAL = int(os.environ.get("MAINTENANCE_INTERVAL", "50"))

# ---------------- context / token sizing ----------------
# Delay range (seconds) before running background auto-linking task
AUTO_LINKING_DELAY_SECS: Tuple[float, float] = (1.5, 3.0)
# Default description for the initial plan step if none exists
DEFAULT_PLAN_STEP = "Assess goal, gather context, formulate initial plan."

# Limits for various context components included in the prompt (PRE-FETCH LIMITS)
CONTEXT_RECENT_ACTIONS_FETCH_LIMIT = 10 # Fetch slightly more than shown
CONTEXT_IMPORTANT_MEMORIES_FETCH_LIMIT = 7
CONTEXT_KEY_THOUGHTS_FETCH_LIMIT = 7
CONTEXT_PROCEDURAL_MEMORIES_FETCH_LIMIT = 3 # Fetch limit for procedural
CONTEXT_PROACTIVE_MEMORIES_FETCH_LIMIT = 5 # Fetch limit for proactive goal-relevant
CONTEXT_LINK_TRAVERSAL_FETCH_LIMIT = 5 # Fetch limit for link traversal per direction
CONTEXT_GOAL_DETAILS_FETCH_LIMIT = 3 # How many parent goals to fetch details for in context
```

```
# Limits for items SHOWN in final prompt context (after potential truncation/summarization)
CONTEXT_RECENT_ACTIONS_SHOW_LIMIT = 7
CONTEXT_IMPORTANT_MEMORIES_SHOW_LIMIT = 5
CONTEXT_KEY_THOUGHTS_SHOW_LIMIT = 5
CONTEXT_PROCEDURAL_MEMORIES_SHOW_LIMIT = 2 # Limit procedural memories included
CONTEXT_PROACTIVE_MEMORIES_SHOW_LIMIT = 3 # Limit goal-relevant memories included
CONTEXT_WORKING_MEMORY_SHOW_LIMIT = 10 # Max working memory items shown in context
CONTEXT_LINK_TRAVERSAL_SHOW_LIMIT = 3 # Max links shown per direction in link summary
CONTEXT_GOAL_STACK_SHOW_LIMIT = 5 # Max goals shown from the stack in context

# Token limits triggering context compression
CONTEXT_MAX_TOKENS_COMPRESS_THRESHOLD = 15_000
CONTEXT_COMPRESSION_TARGET_TOKENS = 5_000 # Target size after compression

# Maximum number of consecutive tool execution errors before aborting
MAX_CONSECUTIVE_ERRORS = 3

# ---------------- unified-memory tool constants ----------------
# Define constants for all UMS tool names for consistency and easy updates
TOOL_GET_WORKFLOW_DETAILS = "unified_memory:get_workflow_details"
TOOL_GET_CONTEXT = "unified_memory:get_workflow_context" # Core context retrieval tool
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"
TOOL_ADD_ACTION_DEPENDENCY = "unified_memory:add_action_dependency"
TOOL_GET_ACTION_DEPENDENCIES = "unified_memory:get_action_dependencies"
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"
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"
TOOL_SEMANTIC_SEARCH = "unified_memory:search_semantic_memories"
TOOL_CREATE_THOUGHT_CHAIN = "unified_memory:create_thought_chain"
TOOL_GET_THOUGHT_CHAIN = "unified_memory:get_thought_chain"
TOOL_DELETE_EXPIRED_MEMORIES = "unified_memory:delete_expired_memories"
TOOL_COMPUTE_STATS = "unified_memory:compute_memory_statistics"
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" # Keyword/filter-based search
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"
# --- New UMS Tool Constants for Goal Stack ---
TOOL_PUSH_SUB_GOAL = "unified_memory:push_sub_goal" # Assumed UMS tool
TOOL_MARK_GOAL_STATUS = "unified_memory:mark_goal_status" # Assumed UMS tool
TOOL_GET_GOAL_DETAILS = "unified_memory:get_goal_details" # Assumed UMS tool
# --- Agent-internal tool name constant ---
AGENT_TOOL_UPDATE_PLAN = "agent:update_plan"

# --- Background Task Management ---
BACKGROUND_TASK_TIMEOUT_SECONDS = 60.0 # Timeout for individual background tasks
MAX_CONCURRENT_BG_TASKS = 10 # Limit concurrent background tasks (linking, promotion)

# ==========================================================================
# Utility helpers
# ==========================================================================

def _fmt_id(val: Any, length: int = 8) -> str:
    """Return a short id string safe for logs."""
    s = str(val) if val is not None else "?"
    return s[:length] if len(s) >= length else s


def _utf8_safe_slice(s: str, max_len: int) -> str:
    """Return a UTF-8 boundary-safe slice ▯ max_len bytes."""
    # Ensure input is a string
    if not isinstance(s, str):
        s = str(s)
    return s.encode("utf-8")[:max_len].decode("utf-8", "ignore")


def _truncate_context(context: Dict[str, Any], max_len: int = 25_000) -> str:
    """
    Structure-aware context truncation with UTF-8 safe fallback.

    Attempts to intelligently reduce context size while preserving structure
    and indicating where truncation occurred. Honors CONTEXT_*_SHOW_LIMIT constants.

    1. Serialise full context - if within limit, return.
    2. Iteratively truncate known large lists (e.g., recent actions, memories, goal stack)
       based on SHOW_LIMIT constants, adding a note about omissions.
    3. Remove lowest-priority top-level keys (e.g., procedures, links)
       until size fits.
    4. Final fallback: utf-8 safe byte slice of the full JSON, attempting to
       make it syntactically valid-ish and adding a clear truncation marker.
    """
    try:
        # Serialize the context to JSON. Use default=str for non-serializable types like datetime.
        full = json.dumps(context, indent=2, default=str, ensure_ascii=False)
    except TypeError:
        # Handle potential non-serializable types even before dumping if necessary
        context = json.loads(json.dumps(context, default=str)) # Pre-process
        full = json.dumps(context, indent=2, default=str, ensure_ascii=False)

    # If already within the limit, return the full JSON string
    if len(full) <= max_len:
        return full

    # Log that truncation is starting
    log.debug(f"Context length {len(full)} exceeds max {max_len}. Applying structured truncation.")
    ctx_copy = copy.deepcopy(context) # Work on a copy
    ctx_copy["_truncation_applied"] = "structure-aware" # Add metadata about truncation
    original_length = len(full)

    # Define paths to lists that can be truncated and the number of items to keep (using SHOW_LIMIT constants)
    list_paths_to_truncate = [ # (parent_key or None, key_of_list, items_to_keep_constant)
        ("core_context", "recent_actions", CONTEXT_RECENT_ACTIONS_SHOW_LIMIT),
        ("core_context", "important_memories", CONTEXT_IMPORTANT_MEMORIES_SHOW_LIMIT),
        ("core_context", "key_thoughts", CONTEXT_KEY_THOUGHTS_SHOW_LIMIT),
        (None, "proactive_memories", CONTEXT_PROACTIVE_MEMORIES_SHOW_LIMIT),
        # Apply limit to the list *within* the working memory dict
        ("current_working_memory", "working_memories", CONTEXT_WORKING_MEMORY_SHOW_LIMIT),
        (None, "relevant_procedures", CONTEXT_PROCEDURAL_MEMORIES_SHOW_LIMIT),
        # --- Add goal stack truncation ---
        ("current_goal_context", "goal_stack_summary", CONTEXT_GOAL_STACK_SHOW_LIMIT),
    ]
    # Define keys to remove entirely if context is still too large, in order of least importance
    keys_to_remove_low_priority = [
        "relevant_procedures",
        "proactive_memories",
        "contextual_links", # Link summary is lower priority than core items
        # Removing items from core_context is more impactful
        ("core_context", "key_thoughts"), # Check nested key first
        ("core_context", "important_memories"),
        ("core_context", "recent_actions"),
        "core_context", # Remove the entire core context dict last
        "current_working_memory", # Remove working memory context before byte slice
        "current_goal_context", # Remove goal context before byte slice (lower priority than plan?)
    ]

    # 1. Truncate specified lists based on SHOW_LIMIT constants
    for parent, key, keep_count in list_paths_to_truncate:
        try:
            container = ctx_copy
            # Navigate to the parent dictionary if specified
            if parent:
                if parent not in container or not isinstance(container[parent], dict):
                    continue # Parent doesn't exist or isn't a dict
                container = container[parent]

            # Check if the key exists, is a list, and needs truncation
            if key in container and isinstance(container[key], list) and len(container[key]) > keep_count:
                original_count = len(container[key])
                # Add a note indicating truncation within the list itself
                note = {"truncated_note": f"{original_count - keep_count} items omitted from '{key}'"}
                # Slice to keep the desired number of items
                container[key] = container[key][:keep_count]
                # Append the note if there's space or if it's crucial context
                if keep_count > 0: # Only add note if we kept some items
                    container[key].append(note)
                log.debug(f"Truncated list '{key}' (under '{parent or 'root'}') to {keep_count} items (+
                note).")

            # Check size after each list truncation
            serial = json.dumps(ctx_copy, indent=2, default=str, ensure_ascii=False)
            if len(serial) <= max_len:
                log.info(f"Context truncated successfully after list reduction (Length: {len(serial)}).")
                return serial
        except (KeyError, TypeError, IndexError) as e:
            # Log errors during this specific truncation but continue with others
            log.warning(f"Error during list truncation for key '{key}' (under '{parent}'): {e}")
            continue

    # 2. Remove low-priority keys if still too large
    for key_info in keys_to_remove_low_priority:
        removed = False
        key_to_remove = key_info
        parent_key_to_remove = None
        if isinstance(key_info, tuple): # Handle nested keys like ("core_context", "key_thoughts")
            parent_key_to_remove, key_to_remove = key_info

        # Try removing from the parent or root level
        container_to_remove = ctx_copy
        if parent_key_to_remove:
            if parent_key_to_remove in container_to_remove and
            isinstance(container_to_remove[parent_key_to_remove], dict):
                container_to_remove = container_to_remove[parent_key_to_remove]
            else:
                continue # Skip if parent doesn't exist or isn't a dict

        # Remove the key if it exists in the target container
        if key_to_remove in container_to_remove:
            container_to_remove.pop(key_to_remove)
            removed = True
            log.debug(f"Removed low-priority key '{key_to_remove}' from {'root' if parent_key_to_remove is
            None else parent_key_to_remove} for truncation.")

        # Check size after each key removal
        if removed:
            serial = json.dumps(ctx_copy, indent=2, default=str, ensure_ascii=False)
            if len(serial) <= max_len:
                log.info(f"Context truncated successfully after key removal (Length: {len(serial)}).")
                return serial

    # 3. Ultimate fallback: UTF-8 safe byte slice
    log.warning(f"Structured truncation insufficient (Length still {len(serial)}). Applying final
    byte-slice.")
    # Slice the original full JSON string, leaving some buffer for the truncation note/closing chars
    clipped_json_str = _utf8_safe_slice(full, max_len - 50)
    # Attempt to make it look somewhat like valid JSON by finding the last closing brace/bracket
    try:
        last_brace = clipped_json_str.rfind('}')
        last_bracket = clipped_json_str.rfind(']')
        cutoff = max(last_brace, last_bracket)
        if cutoff > 0:
            # Truncate after the last complete element and add a note + closing brace
            final_str = clipped_json_str[:cutoff+1] + '\n// ... (CONTEXT TRUNCATED BY BYTE LIMIT) ...\n}'
        else:
            # If no closing elements found, just add a basic note
            final_str = clipped_json_str + '... (CONTEXT TRUNCATED)'
    except Exception:
        # Fallback if string manipulation fails
        final_str = clipped_json_str + '... (CONTEXT TRUNCATED)'

    log.error(f"Context severely truncated from {original_length} to {len(final_str)} bytes using fallback.")
    return final_str

# =============================================================================
# Dataclass & pydantic models
# =============================================================================

class PlanStep(BaseModel):
    """Represents a single step in the agent's plan."""
    id: str = Field(default_factory=lambda: f"step-{MemoryUtils.generate_id()[:8]}", description="Unique
    identifier for the plan step.")
    description: str = Field(..., description="Clear, concise description of the action or goal for this
    step.")
    status: str = Field(
        default="planned",
        description="Current status of the step: planned, in_progress, completed, failed, skipped.",
    )
    depends_on: List[str] = Field(default_factory=list, description="List of other plan step IDs that must be
    completed before this step can start.")
    assigned_tool: Optional[str] = Field(default=None, description="Specific tool designated for executing
    this step, if applicable.")
    tool_args: Optional[Dict[str, Any]] = Field(default=None, description="Arguments to be passed to the
    assigned tool.")
    result_summary: Optional[str] = Field(default=None, description="A brief summary of the outcome after the
    step is completed or failed.")
    is_parallel_group: Optional[str] = Field(default=None, description="Optional tag to group steps that can
    potentially run in parallel.")


def _default_tool_stats() -> Dict[str, Dict[str, Union[int, float]]]:
    """Factory function for initializing tool usage statistics dictionary."""
    # Use defaultdict for convenience: accessing a non-existent tool key will create its default stats entry
    return defaultdict(lambda: {"success": 0, "failure": 0, "latency_ms_total": 0.0})


@dataclass
class AgentState:
    """
    Represents the complete persisted runtime state of the Agent Master Loop.

    This dataclass holds all information necessary to resume the agent's operation,
    including workflow context, **goal hierarchy**, planning state, error tracking,
    meta-cognition metrics, and adaptive thresholds.

    Attributes:
        workflow_id: The ID of the primary workflow the agent is currently focused on.
        context_id: The specific context ID for memory operations (often matches workflow_id).
        workflow_stack: A list maintaining the hierarchy of active workflows (e.g., for sub-workflows).
        goal_stack: List of goal dictionaries tracking hierarchical goals. **(NEW)**
        current_goal_id: ID of the goal at the top of the `goal_stack` (the agent's current focus).
        **(RENAMED/REPURPOSED)**
        current_plan: A list of `PlanStep` objects representing the agent's current plan.
        current_thought_chain_id: ID of the active thought chain for recording reasoning.
        last_action_summary: A brief string summarizing the outcome of the last action taken.
        current_loop: The current iteration number of the main agent loop.
        goal_achieved_flag: Boolean flag indicating if the *overall root goal* has been marked as achieved.
        consecutive_error_count: Counter for consecutive failed actions, used for error limiting.
        needs_replan: Boolean flag indicating if the agent needs to revise its plan in the next cycle.
        last_error_details: A dictionary holding structured information about the last error encountered
        **(Enhanced with category)**.
        successful_actions_since_reflection: Counter for successful *agent-level* actions since the last
        reflection.
        successful_actions_since_consolidation: Counter for successful *agent-level* actions since the last
        consolidation.
        loops_since_optimization: Counter for loops since the last working memory optimization/focus update.
        loops_since_promotion_check: Counter for loops since the last memory promotion check cycle.
        loops_since_stats_adaptation: Counter for loops since the last statistics check and threshold
        adaptation.
        loops_since_maintenance: Counter for loops since the last maintenance task (e.g., deleting expired
        memories).
        reflection_cycle_index: Index to cycle through different reflection types.
        last_meta_feedback: Stores the summary of the last reflection/consolidation output for the next
        prompt.
        current_reflection_threshold: The current dynamic threshold for triggering reflection **(Adaptive +
        Momentum)**.
        current_consolidation_threshold: The current dynamic threshold for triggering consolidation
        **(Adaptive + Momentum)**.
        tool_usage_stats: Dictionary tracking success/failure counts and latency for each tool used.
        background_tasks: (Transient) Set holding currently running asyncio background tasks (not saved to
        state file).
    """

    # --- workflow stack ---
    workflow_id: Optional[str] = None
    context_id: Optional[str] = None
    workflow_stack: List[str] = field(default_factory=list)
    # --- Goal Management --- (NEW/MODIFIED)
    goal_stack: List[Dict[str, Any]] = field(default_factory=list) # Stores {goal_id, description, status,
    ...}
    current_goal_id: Optional[str] = None # ID of the active goal (top of the stack)

    # --- planning & reasoning ---
    current_plan: List[PlanStep] = field(
        default_factory=lambda: [PlanStep(description=DEFAULT_PLAN_STEP)]
    )
    current_thought_chain_id: Optional[str] = None # Tracks the current reasoning thread
    last_action_summary: str = "Loop initialized."
    current_loop: int = 0
    goal_achieved_flag: bool = False # Flag to signal loop termination (based on root goal)

    # --- error/replan ---
    consecutive_error_count: int = 0
    needs_replan: bool = False # Flag to force replanning cycle
    last_error_details: Optional[Dict[str, Any]] = None # Stores info about the last error **(Enhanced)**

    # --- meta-cognition metrics ---
    # Counters reset when corresponding meta-task runs or on error
    successful_actions_since_reflection: float = 0.0 # Use float for potential fractional counting
    successful_actions_since_consolidation: float = 0.0
    # Loop counters reset when corresponding periodic task runs
    loops_since_optimization: int = 0
    loops_since_promotion_check: int = 0
    loops_since_stats_adaptation: int = 0
    loops_since_maintenance: int = 0
    reflection_cycle_index: int = 0 # Used to cycle through reflection types
    last_meta_feedback: Optional[str] = None # Feedback from last meta-task for next prompt

    # adaptive thresholds (dynamic) - Initialized from constants, adapted based on stats and momentum
    current_reflection_threshold: int = BASE_REFLECTION_THRESHOLD
    current_consolidation_threshold: int = BASE_CONSOLIDATION_THRESHOLD

    # tool statistics - tracks usage counts and latency
    tool_usage_stats: Dict[str, Dict[str, Union[int, float]]] = field(
        default_factory=_default_tool_stats
    )

    # background tasks (transient) - Not saved/loaded, managed at runtime
    background_tasks: Set[asyncio.Task] = field(
        default_factory=set, init=False, repr=False
    )

# =====================================================================
# Agent Master Loop
# =====================================================================
class AgentMasterLoop:
    """
    Agent Master Loop Orchestrator.

    This class orchestrates the primary think-act cycle of the AI agent.
    It manages state, interacts with the Unified Memory System via MCPClient,
    calls the LLM for decision-making, handles plan execution, manages goal hierarchies,
    and runs periodic meta-cognitive tasks (reflection, consolidation, optimization).
    This version integrates rich context gathering, goal stack management, mental momentum bias,
    and detailed prompting as defined in Phase 1 of the v4.1 plan.
    """

    # Set of tool names considered internal or meta-cognitive,
    # which typically shouldn't be recorded as primary agent actions.
    _INTERNAL_OR_META_TOOLS: Set[str] = {
        # Action recording itself is meta
        TOOL_RECORD_ACTION_START,
        TOOL_RECORD_ACTION_COMPLETION,
        # Information retrieval is usually part of the agent's thought process, not a world-altering action
        TOOL_GET_CONTEXT,
        TOOL_GET_WORKING_MEMORY,
        TOOL_SEMANTIC_SEARCH,
        TOOL_HYBRID_SEARCH,
        TOOL_QUERY_MEMORIES,
        TOOL_GET_MEMORY_BY_ID,
        TOOL_GET_LINKED_MEMORIES,
        TOOL_GET_ACTION_DETAILS,
        TOOL_GET_ARTIFACTS,
        TOOL_GET_ARTIFACT_BY_ID,
        TOOL_GET_ACTION_DEPENDENCIES,
        TOOL_GET_THOUGHT_CHAIN,
        TOOL_GET_WORKFLOW_DETAILS, # Getting details is informational
        TOOL_GET_GOAL_DETAILS, # Getting goal details is informational (NEW)
        # Managing relationships is meta
        TOOL_ADD_ACTION_DEPENDENCY,
        TOOL_CREATE_LINK,
        # Goal stack management is meta
        TOOL_PUSH_SUB_GOAL, # (NEW)
        TOOL_MARK_GOAL_STATUS, # (NEW)
        # Admin/Utility tasks are not primary actions
        TOOL_LIST_WORKFLOWS,
        TOOL_COMPUTE_STATS,
        TOOL_SUMMARIZE_TEXT, # Summarization is a utility
        # Periodic cognitive maintenance and enhancement tasks
        TOOL_OPTIMIZE_WM,
        TOOL_AUTO_FOCUS,
        TOOL_PROMOTE_MEM,
        TOOL_REFLECTION,
        TOOL_CONSOLIDATION,
        TOOL_DELETE_EXPIRED_MEMORIES,
        # Agent's internal mechanism for plan updates
        AGENT_TOOL_UPDATE_PLAN,
    }

    # -------------------------------------------------------------- ctor --
    def __init__(
        self, mcp_client_instance: MCPClient, agent_state_file: str = AGENT_STATE_FILE
    ):
        """
        Initializes the AgentMasterLoop.

        Args:
            mcp_client_instance: An initialized instance of the MCPClient.
            agent_state_file: Path to the file for saving/loading agent state.
        """
        # Ensure MCPClient dependency is met
        if not MCP_CLIENT_AVAILABLE:
            raise RuntimeError("MCPClient unavailable. Cannot initialize AgentMasterLoop.")

        self.mcp_client = mcp_client_instance
        # Ensure the Anthropic client is available via MCPClient
        if not hasattr(mcp_client_instance, 'anthropic') or not isinstance(mcp_client_instance.anthropic,
        AsyncAnthropic):
             self.logger.critical("Anthropic client not found within provided MCPClient instance.")
             raise ValueError("Anthropic client required via MCPClient.")
        self.anthropic_client: AsyncAnthropic = self.mcp_client.anthropic # type: ignore
        self.logger = log
        self.agent_state_file = Path(agent_state_file)

        # Configuration parameters for cognitive processes
        self.consolidation_memory_level = MemoryLevel.EPISODIC.value
        self.consolidation_max_sources = 10 # Max memories to feed into consolidation
        self.auto_linking_threshold = 0.7 # Similarity threshold for auto-linking
        self.auto_linking_max_links = 3 # Max links to create per auto-link trigger

        # Sequence of reflection types to cycle through
        self.reflection_type_sequence = [
            "summary", "progress", "gaps", "strengths", "plan",
        ]

        # Initialize agent state (will be overwritten by load if file exists)
        self.state = AgentState()
        # Event to signal graceful shutdown
        self._shutdown_event = asyncio.Event()
        # Lock for safely managing the background tasks set
        self._bg_tasks_lock = asyncio.Lock()
        # Semaphore to limit concurrent background tasks
        self._bg_task_semaphore = asyncio.Semaphore(MAX_CONCURRENT_BG_TASKS)
        # Placeholder for loaded tool schemas
        self.tool_schemas: List[Dict[str, Any]] = []

    # ---------------------------------------------------------- shutdown --
    async def shutdown(self) -> None:
        """
        Initiates graceful shutdown of the agent loop.

        Sets the shutdown event, cancels pending background tasks,
        and saves the final agent state.
        """
        self.logger.info("Shutdown requested.")
        self._shutdown_event.set() # Signal loops and tasks to stop
        await self._cleanup_background_tasks() # Wait for background tasks
        await self._save_agent_state() # Save final state
        self.logger.info("Agent loop shutdown complete.")

    # ----------------------------------------------------------- prompt --
    def _construct_agent_prompt(
        self, goal: str, context: Dict[str, Any]
    ) -> List[Dict[str, Any]]:
        """
        Builds the system and user prompts for the agent LLM.

        Integrates the overall goal, available tools with schemas, detailed
        process instructions, cognitive guidance, and the current runtime context
        (plan, errors, feedback, **goal stack**, memories, etc.) into the prompt structure
        expected by the Anthropic API. Includes robust context truncation.
        **Enhanced with goal stack guidance, mental momentum bias, and error recovery strategies.**
        """
        # <<< Start Integration Block: Enhance _construct_agent_prompt (Goal Stack + Momentum) >>>
        # ---------- system ----------
        # Initialize the list for system prompt blocks
        system_blocks: List[str] = [
            f"You are '{AGENT_NAME}', an AI agent orchestrator using a Unified Memory System.",
            "",
            # --- GOAL CONTEXT (Logic moved outside list literal) ---
            f"Overall Goal: {goal}", # This remains the root goal
        ]

        # Extract current goal details from context and append the appropriate string
        current_goal_info = context.get("current_goal_context", {}).get("current_goal")
        if current_goal_info:
            system_blocks.append(f"Current Goal: {current_goal_info.get('description', 'N/A')} (ID:
            {_fmt_id(current_goal_info.get('goal_id'))}, Status: {current_goal_info.get('status', 'N/A')})")
        else:
            system_blocks.append("Current Goal: None specified (Focus on Overall Goal or plan step)")

        # Add the separator and the next section header
        system_blocks.append("") # Separator
        system_blocks.append("Available Unified Memory & Agent Tools (Use ONLY these):")

        # --- Continue appending other system blocks as before ---
        if not self.tool_schemas:
            system_blocks.append("- CRITICAL WARNING: No tools loaded. Cannot function.")
        else:
            # Define key tools to highlight in the prompt (adding Goal tools)
            essential_cognitive_tools = {
                TOOL_ADD_ACTION_DEPENDENCY, TOOL_RECORD_ARTIFACT, TOOL_HYBRID_SEARCH,
                TOOL_STORE_MEMORY, TOOL_UPDATE_MEMORY, TOOL_CREATE_LINK,
                TOOL_CREATE_THOUGHT_CHAIN, TOOL_GET_THOUGHT_CHAIN, TOOL_RECORD_THOUGHT,
                TOOL_REFLECTION, TOOL_CONSOLIDATION, TOOL_PROMOTE_MEM,
                TOOL_OPTIMIZE_WM, TOOL_AUTO_FOCUS, TOOL_GET_WORKING_MEMORY,
                TOOL_QUERY_MEMORIES, TOOL_SEMANTIC_SEARCH,
                AGENT_TOOL_UPDATE_PLAN, # Also highlight the agent's own tool
                # --- Goal Stack Tools ---
                TOOL_PUSH_SUB_GOAL, TOOL_MARK_GOAL_STATUS, TOOL_GET_GOAL_DETAILS,
            }
            # Format each tool schema for the prompt
            for schema in self.tool_schemas:
                sanitized = schema["name"]
                # Ensure mapping exists before using it
                original = self.mcp_client.server_manager.sanitized_to_original.get(sanitized, sanitized)
                desc = schema.get("description", "No description.")
                is_essential = original in essential_cognitive_tools
                prefix = "**" if is_essential else "" # Highlight key tools
                # Provide tool name, mapping, description, and schema
                # Handle different schema formats ('input_schema' vs 'parameters')
                input_schema_str = json.dumps(schema.get('input_schema', schema.get('parameters', {})))
                system_blocks.append(
                    f"\n- {prefix}Name: `{sanitized}` (Represents: `{original}`){prefix}\n"
                    f"  Desc: {desc}\n"
                    f"  Schema: {input_schema_str}"
                )
        system_blocks.append("")
        # --- Detailed Process Instructions (Enhanced for Goals & Momentum) ---
        system_blocks.extend([
            "Your Process at each step:",
            "1.  Context Analysis: Deeply analyze 'Current Context'. Note workflow status, errors
            (`last_error_details` - *pay attention to error `type`*), **goal stack (`current_goal_context` ->
            `goal_stack_summary`) and the `current_goal`**, recent actions, memories (`core_context`,
            `proactive_memories`), thoughts, `current_plan`, `relevant_procedures`,
            **`current_working_memory` (use this for immediate relevance, note `focal_memory_id` if
            present)**, `current_thought_chain_id`, and `meta_feedback`. Pay attention to memory
            `importance`/`confidence` and context component `retrieved_at` timestamps.",
            "2.  Error Handling: If `last_error_details` exists, **FIRST** reason about the error `type` and
            `message`. Propose a recovery strategy in your reasoning. Refer to 'Recovery Strategies' below.",
            "3.  Reasoning & Planning:",
            "    a. State step-by-step reasoning towards the **Current Goal**, integrating context and
            feedback. Consider `current_working_memory` for immediate context. Record key thoughts using
            `record_thought` and specify the `thought_chain_id` if different from
            `current_thought_chain_id`.",
            "    b. Evaluate `current_plan`. Is it aligned with the **Current Goal**? Is it valid? Does it
            address errors? Are dependencies (`depends_on`) likely met? Check for cycles.",
            "    c. **Goal Management:** If the Current Goal is too complex, break it down by using
            `push_sub_goal` with a clear `description`. When a sub-goal is achieved or fails, use
            `mark_goal_status` with the `goal_id` and appropriate `status` ('completed' or 'failed').", #
            Goal instructions
            "    d. **Action Dependencies:** If planning Step B requires output from Step A (action ID
            'a123'), include `\"depends_on\": [\"a123\"]` in Step B's plan object.",
            "    e. **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.",
            "    f. **Direct Memory Management:** If you synthesize a critical new fact, insight, or
            procedure, 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`.",
            "    g. **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`.",
            "    h. **Linking:** Identify potential memory relationships (causal, supportive, contradictory).
            Plan to use `create_memory_link` with specific `link_type`s (e.g., `SUPPORTS`, `CONTRADICTS`,
            `CAUSAL`, `REFERENCES`).",
            "    i. **Search:** Prefer `hybrid_search_memories` for mixed queries needing relevance and
            keyword matching. Use `search_semantic_memories` for pure conceptual similarity.",
            "    j. **Update Plan Tool / Repair:** Use `agent:update_plan` ONLY for significant changes
            (error recovery, major strategy shift, multi-step insertion/removal, fixing validation issues
            like cycles). **If `needs_replan` is true in context, prioritize using this tool to fix the
            plan.** Do NOT use it for simply marking a step complete.",
            "4.  Action Decision: Choose **ONE** action based on the *first planned step* in your current
            plan:",
            "    *   Call Memory/Other Tool: Select the most precise tool (including goal tools like
            `push_sub_goal` or `mark_goal_status`). Provide args per schema. **Mandatory:** Call
            `create_workflow` if context shows 'No Active Workflow'.", # Added goal tools example
            "    *   Record Thought: Use `record_thought` for logging reasoning, questions, hypotheses etc.
            Specify `thought_chain_id` if not the current one.",
            "    *   Update Plan Tool: Call `agent:update_plan` with the **complete, repaired** plan
            structure if major changes are needed or validation failed.",
            "    *   Signal Completion: If the **Current Goal** is MET (use `mark_goal_status`) OR the
            Overall Goal is MET (respond ONLY with the text \"Goal Achieved:\" followed by a brief
            summary).", # Clarified goal completion
            "5.  Output Format: Respond **ONLY** with the valid JSON for the chosen tool call OR the \"Goal
            Achieved:\" text."
        ])
        # --- Key Considerations (Enhanced for Goals & Momentum) ---
        system_blocks.extend([
            "\nKey Considerations:",
            "*   Goal Focus: Always work towards the **Current Goal** (top of the stack). Use `push_sub_goal`
            and `mark_goal_status` to manage the hierarchy.",
            "*   **Mental Momentum:** Prioritize completing the current plan steps if progress is steady.
            Justify significant deviations or replanning efforts.", # Momentum instruction
            "*   Dependencies & Cycles: Ensure `depends_on` actions are likely complete. Avoid circular
            dependencies. Use `get_action_details` if unsure.",
            "*   Artifacts: Track outputs (`record_artifact`), retrieve inputs
            (`get_artifacts`/`get_artifact_by_id`).",
            "*   Memory: Store important learned info (`store_memory`). Update incorrect info
            (`update_memory`). Use confidence scores.",
            "*   Thought Chains: Use `create_thought_chain` for complex sub-problems. Use the correct
            `thought_chain_id` when recording thoughts.",
            "*   Linking: Use specific `link_type`s to build the knowledge graph.",
            "*   Focus: Leverage `current_working_memory` for immediate context. Note the
            `focal_memory_id`.",
            "*   Errors: Prioritize error analysis and recovery based on `last_error_details.type`."
        ])
        # --- Recovery Strategies Guidance ---
        system_blocks.extend([
            "\nRecovery Strategies based on `last_error_details.type`:",
            "*   `InvalidInputError`: Review tool schema, arguments, and context. Correct the arguments and
            retry OR choose a different tool/step.",
            "*   `DependencyNotMetError`: Use `get_action_details` on dependency IDs to check status. Adjust
            plan order using `agent:update_plan` or wait.",
            "*   `ServerUnavailable` / `NetworkError`: The tool's server might be down. Try a different tool,
            wait, or adjust the plan.",
            "*   `APILimitError` / `RateLimitError`: The external API (e.g., LLM) is busy. Plan to wait
            (record a thought) before retrying the step.",
            "*   `ToolExecutionError` / `ToolInternalError`: The tool failed internally. Analyze the error
            message. Maybe try different arguments, use an alternative tool, or adjust the plan.",
            "*   `PlanUpdateError`: The plan structure you proposed was invalid. Re-examine the plan steps
            and dependencies, then try `agent:update_plan` again with a valid list.",
            "*   `PlanValidationError`: The proposed plan has logical issues (e.g., cycles). Debug
            dependencies and propose a corrected plan structure using `agent:update_plan`.",
            "*   `CancelledError`: The previous action was cancelled. Re-evaluate the current step.",
            "*   `GoalManagementError`: An error occurred managing the goal stack (e.g., trying to mark a
            non-existent goal). Review `current_goal_context` and the goal stack logic.", # Added Goal error
            "*   `UnknownError` / `UnexpectedExecutionError`: Analyze the error message carefully. Try to
            understand the cause. You might need to simplify the step, use a different approach, or ask for
            clarification via `record_thought` if stuck."
        ])
        system_prompt = "\n".join(system_blocks)
        # <<< End Integration Block: Enhance _construct_agent_prompt >>>

        # ---------- user ----------
        # Construct the user part of the prompt, including truncated context
        context_json = _truncate_context(context) # Apply robust truncation
        user_blocks = [
            "Current Context:",
            "```json",
            context_json,
            "```",
            "",
            "Current Plan:",
            "```json",
            # Serialize current plan steps (ensure model_dump handles exclude_none)
            json.dumps(
                [step.model_dump(exclude_none=True) for step in self.state.current_plan],
                indent=2,
                ensure_ascii=False,
            ),
            "```",
            "",
            # Include summary of the last action taken
            f"Last Action Summary:\n{self.state.last_action_summary}\n",
        ]
        # If there was an error in the previous step, include details prominently
        if self.state.last_error_details:
            user_blocks += [
                "**CRITICAL: Address Last Error Details**:", # Highlight error
                "```json",
                # Use default=str for safe serialization of potential complex error objects
                json.dumps(self.state.last_error_details, indent=2, default=str),
                "```",
                "",
            ]
        # If there's feedback from meta-cognitive tasks, include it
        if self.state.last_meta_feedback:
            user_blocks += [
                "**Meta-Cognitive Feedback**:", # Highlight feedback
                self.state.last_meta_feedback,
                "",
            ]
        # Reiterate the overall goal and the final instruction
        user_blocks += [
            # Reiterate current goal from context for emphasis
            f"Current Goal Reminder: {context.get('current_goal_context', {}).get('current_goal',
            {}).get('description', 'Overall Goal')}",
            "",
            # Updated instruction emphasizing error/plan repair AND goal management
            "Instruction: Analyze context & errors (use recovery strategies if needed). Reason step-by-step
            towards the Current Goal. Evaluate and **REPAIR** the plan if `needs_replan` is true or errors
            indicate plan issues (use `agent:update_plan`). Manage goals using `push_sub_goal` or
            `mark_goal_status` if needed. Otherwise, decide ONE action based on the *first planned step*:
            call a tool (output tool_use JSON), record a thought (`record_thought`), or signal completion
            (use `mark_goal_status` for sub-goals or output 'Goal Achieved: ...' for overall goal).",
        ]
```

```
        user_prompt = "\n".join(user_blocks)

        # Return structure for Anthropic API (user prompt combines system instructions and current state)
        # Note: Anthropic recommends placing system prompts outside the 'messages' list if using their client
        directly.
        # Here, we combine them into the user message content as per the original structure.
        # >>>>> PRESERVED ORIGINAL PROMPT STRUCTURE <<<<<
        return [{"role": "user", "content": system_prompt + "\n---\n" + user_prompt}]


    # ---------------------------------------------------------- bg-task utils --
    def _background_task_done(self, task: asyncio.Task) -> None:
        """Callback attached to background tasks upon completion."""
        # Schedule the safe cleanup coroutine to avoid blocking the callback
        asyncio.create_task(self._background_task_done_safe(task))

    async def _background_task_done_safe(self, task: asyncio.Task) -> None:
        """
        Safely removes a completed task from the tracking set, releases the semaphore,
        and logs any exceptions. Ensures thread-safety using an asyncio Lock.
        """
        # <<< Start Integration Block: Enhance _background_task_done_safe (Phase 1, Step 5) >>>
        was_present = False
        async with self._bg_tasks_lock: # Acquire lock before modifying the set
            if task in self.state.background_tasks:
                 self.state.background_tasks.discard(task)
                 was_present = True

        # Release the semaphore ONLY if the task was successfully removed from the set
        # This prevents releasing the semaphore multiple times if the callback somehow fires twice
        if was_present:
            try:
                self._bg_task_semaphore.release()
                log.debug(f"Released semaphore. Count: {self._bg_task_semaphore._value}. Task:
                {task.get_name()}")
            except ValueError:
                 # This can happen if release is called more times than acquire (should not normally occur)
                 log.warning(f"Semaphore release attempt failed for task {task.get_name()}. Already fully
                 released?")
            except Exception as sem_err:
                 log.error(f"Unexpected error releasing semaphore for task {task.get_name()}: {sem_err}")

        # Log cancellation or exceptions after releasing the lock and semaphore
        if task.cancelled():
            self.logger.debug(f"Background task {task.get_name()} was cancelled.")
            return
        # Check if the task encountered an exception
        exc = task.exception()
        if exc:
            # Log the exception details
            self.logger.error(
                # Provide task name/info for better debugging
                f"Background task {task.get_name()} failed: {type(exc).__name__}",
                exc_info=(type(exc), exc, exc.__traceback__), # Provide full traceback info
            )
        # <<< End Integration Block: Enhance _background_task_done_safe >>>

    def _start_background_task(self, coro_fn, *args, **kwargs) -> asyncio.Task:
        """
        Creates and starts an asyncio task for a background operation.

        Acquires a semaphore slot before starting. Includes timeout handling.
        Captures essential state (workflow_id, context_id) at the time of creation
        to ensure the background task operates on the correct context, even if the
        main agent state changes before the task runs. Adds the task to the
        tracking set for cleanup.

        Args:
            coro_fn: The async function (coroutine) to run in the background.
                     Must accept `self` as the first argument if it's an instance method.
            *args: Positional arguments to pass to `coro_fn`.
            **kwargs: Keyword arguments to pass to `coro_fn`.

        Returns:
            The created asyncio.Task object.
        """
        # <<< Start Integration Block: Enhance _start_background_task (Phase 1, Step 5) >>>
        # Snapshot critical state needed by the background task at the moment of creation
        snapshot_wf_id = self.state.workflow_id
        snapshot_ctx_id = self.state.context_id
        # Add other state variables here if specific background tasks need them

        # Define an async wrapper function to execute the coroutine with snapshotted state
        async def _wrapper():
            # Acquire semaphore before running the actual work
            log.debug(f"Waiting for semaphore... Task: {asyncio.current_task().get_name()}. Current count:
            {self._bg_task_semaphore._value}")
            await self._bg_task_semaphore.acquire()
            log.debug(f"Acquired semaphore. Task: {asyncio.current_task().get_name()}. New count:
            {self._bg_task_semaphore._value}")
            try:
                # Run the actual coroutine with timeout
                await asyncio.wait_for(
                    coro_fn(
                        self, # Pass the agent instance
                        *args,
                        workflow_id=snapshot_wf_id, # Pass snapshotted workflow_id
                        context_id=snapshot_ctx_id, # Pass snapshotted context_id
                        **kwargs,
                    ),
                    timeout=BACKGROUND_TASK_TIMEOUT_SECONDS
                )
            except asyncio.TimeoutError:
                # Log timeout specifically
                self.logger.warning(f"Background task {asyncio.current_task().get_name()} timed out after
                {BACKGROUND_TASK_TIMEOUT_SECONDS}s.")
            except Exception:
                # Log other exceptions, they will also be logged by the done callback
                self.logger.debug(f"Exception caught within background task wrapper
                {asyncio.current_task().get_name()}. Will be logged by done callback.")
            # finally:
            #     # Ensure semaphore is released *within the task itself*
            #     # This is handled by the done callback now to simplify wrapper logic
            #     # self._bg_task_semaphore.release()
            #     # log.debug(f"Released semaphore via FINALLY. Task: {asyncio.current_task().get_name()}")
            #     pass

        # Create the asyncio task
        # Naming the task helps with debugging
        task_name = f"bg_{coro_fn.__name__}_{_fmt_id(snapshot_wf_id)}_{random.randint(100,999)}"
        task = asyncio.create_task(_wrapper(), name=task_name)

        # Schedule adding the task to the tracking set safely using another task
        # This avoids potential blocking if the lock is held
        asyncio.create_task(self._add_bg_task(task))

        # Add the completion callback to handle cleanup and logging (this callback now also releases
        semaphore)
        task.add_done_callback(self._background_task_done)
        self.logger.debug(f"Started background task: {task.get_name()} for WF {_fmt_id(snapshot_wf_id)}")
        return task
        # <<< End Integration Block: Enhance _start_background_task >>>


    async def _add_bg_task(self, task: asyncio.Task) -> None:
        """Safely add a task to the background task set using the lock."""
        async with self._bg_tasks_lock:
            self.state.background_tasks.add(task)

    async def _cleanup_background_tasks(self) -> None:
        """
        Cancels all pending background tasks and awaits their completion.
        Called during graceful shutdown. Ensures semaphores are released.
        """
        # <<< Start Integration Block: Enhance _cleanup_background_tasks (Phase 1, Step 5) >>>
        tasks_to_cleanup: List[asyncio.Task] = []
        async with self._bg_tasks_lock: # Acquire lock to safely get the list
            # Create a copy to iterate over, as the set might be modified by callbacks
            tasks_to_cleanup = list(self.state.background_tasks)

        if not tasks_to_cleanup:
            self.logger.debug("No background tasks to clean up.")
            return

        self.logger.info(f"Cleaning up {len(tasks_to_cleanup)} background tasks…")
        cancelled_tasks = []
        already_done_tasks = []

        # Cancel running tasks
        for t in tasks_to_cleanup:
            if not t.done():
                # Cancel any task that hasn't finished yet
                t.cancel()
                cancelled_tasks.append(t)
            else:
                 already_done_tasks.append(t)

        # Wait for all tasks (including those just cancelled) to finish
        # return_exceptions=True prevents gather from stopping on the first exception
        results = await asyncio.gather(*tasks_to_cleanup, return_exceptions=True)

        # Log the outcome of each task cleanup
        for i, res in enumerate(results):
            task = tasks_to_cleanup[i]
            task_name = task.get_name()
            if isinstance(res, asyncio.CancelledError):
                # This is expected for tasks that were cancelled above
                self.logger.debug(f"Task {task_name} successfully cancelled during cleanup.")
            elif isinstance(res, Exception):
                # Log any unexpected errors that occurred during task execution/cleanup
                self.logger.error(f"Task {task_name} raised an exception during cleanup: {res}")
            else:
                # Task completed normally before/during cleanup or was already done
                self.logger.debug(f"Task {task_name} finalized during cleanup (completed normally or already
                done).")

            # --- Ensure semaphore is released ---
            # The done callback should handle release, but we add a check here as a failsafe
            # during shutdown, especially if the callback didn't run or failed itself.
            # This is tricky because we don't know if the task acquired the semaphore before being cancelled.
            # A potentially safer approach is *not* to release here unless we are certain the task
            # acquired it and didn't release it. Releasing without acquiring increments the semaphore count.
            # Given the complexity, we rely on the robust done_callback for release.
            # Adding a log message if the semaphore count seems wrong at the end might be useful.

        # Clear the tracking set after all tasks are handled
        async with self._bg_tasks_lock:
            self.state.background_tasks.clear()

        # Final check on semaphore count after cleanup
        final_sem_count = self._bg_task_semaphore._value
        if final_sem_count != MAX_CONCURRENT_BG_TASKS:
             self.logger.warning(f"Semaphore count is {final_sem_count} after cleanup, expected
             {MAX_CONCURRENT_BG_TASKS}. Some tasks might not have released.")

        self.logger.info("Background tasks cleanup finished.")
        # <<< End Integration Block: Enhance _cleanup_background_tasks >>>


    # -------------------------------------------------------- token estimator --
    async def _estimate_tokens_anthropic(self, data: Any) -> int:
        """
        Estimates token count for given data using the Anthropic client.

        Handles serialization of non-string data and provides a fallback
        heuristic (chars/4) if the API call fails.
        """
        if data is None:
            return 0
        try:
            if not self.anthropic_client:
                # This should ideally be caught during initialization
                raise RuntimeError("Anthropic client unavailable for token estimation")

            # Convert data to string if it's not already (e.g., dict, list)
            text_to_count = data if isinstance(data, str) else json.dumps(data, default=str,
            ensure_ascii=False)

            # Use the actual count_tokens method from the anthropic client
            token_count = await self.anthropic_client.count_tokens(text_to_count)
            return int(token_count) # Ensure result is an integer

        except Exception as e:
            # Log the specific error from the API call
            self.logger.warning(f"Token estimation via Anthropic API failed: {e}. Using fallback.")
            # Fallback heuristic: Estimate based on character count
            try:
                text_representation = data if isinstance(data, str) else json.dumps(data, default=str,
                ensure_ascii=False)
                return len(text_representation) // 4 # Rough approximation
            except Exception as fallback_e:
                # Log error if even the fallback fails
                self.logger.error(f"Token estimation fallback failed: {fallback_e}")
                return 0 # Return 0 if all estimation methods fail

    # --------------------------------------------------------------- retry util --
    async def _with_retries(
        self,
        coro_fun, # The async function to call
        *args,
        max_retries: int = 3,
        # Exceptions to retry on (can be customized per call)
        retry_exceptions: Tuple[type[BaseException], ...] = (
            ToolError, ToolInputError, # Specific MCP errors
            asyncio.TimeoutError, ConnectionError, # Common network issues
            APIConnectionError, RateLimitError, # Anthropic network issues
            APIStatusError, # Treat potentially transient API status errors as retryable
        ),
        retry_backoff: float = 2.0, # Exponential backoff factor
        jitter: Tuple[float, float] = (0.1, 0.5), # Random jitter range (min_sec, max_sec)
        **kwargs,
    ):
        """
        Generic retry wrapper for coroutine functions with exponential backoff and jitter.

        Args:
            coro_fun: The async function to execute and potentially retry.
            *args: Positional arguments for `coro_fun`.
            max_retries: Maximum number of total attempts (1 initial + max_retries-1 retries).
            retry_exceptions: Tuple of exception types that trigger a retry.
            retry_backoff: Multiplier for exponential backoff calculation.
            jitter: Tuple (min, max) defining the range for random delay added to backoff.
            **kwargs: Keyword arguments for `coro_fun`.

        Returns:
            The result of `coro_fun` upon successful execution.

        Raises:
            The last exception encountered if all retry attempts fail.
            asyncio.CancelledError: If cancellation occurs during the retry loop or wait.
        """
        attempt = 0
        last_exception = None # Store the last exception for re-raising
        while True:
            try:
                # Attempt to execute the coroutine
                return await coro_fun(*args, **kwargs)
            except retry_exceptions as e:
                attempt += 1
                last_exception = e # Store the exception
                # Check if max retries have been reached
                if attempt >= max_retries:
                    self.logger.error(f"{coro_fun.__name__} failed after {max_retries} attempts. Last error:
                    {e}")
                    raise # Re-raise the last encountered exception
                # Calculate delay with exponential backoff and random jitter
                delay = (retry_backoff ** (attempt - 1)) + random.uniform(*jitter)
                self.logger.warning(
                    f"{coro_fun.__name__} failed ({type(e).__name__}: {str(e)[:100]}...); retry
                    {attempt}/{max_retries} in {delay:.2f}s"
                )
                # Check for shutdown signal *before* sleeping
                if self._shutdown_event.is_set():
                    self.logger.warning(f"Shutdown signaled during retry wait for {coro_fun.__name__}.
                    Aborting retry.")
                    # Raise CancelledError to stop the process cleanly if shutdown occurs during wait
                    raise asyncio.CancelledError(f"Shutdown during retry for {coro_fun.__name__}") from
                    last_exception
                # Wait for the calculated delay
                await asyncio.sleep(delay)
            except asyncio.CancelledError:
                 # Propagate cancellation immediately if caught
                self.logger.info(f"Coroutine {coro_fun.__name__} was cancelled during retry loop.")
                raise


    # ---------------------------------------------------------------- state I/O --
    async def _save_agent_state(self) -> None:
        """
        Saves the current agent state to a JSON file atomically.

        Uses a temporary file and `os.replace` for atomicity. Includes fsync
        for robustness against crashes. Serializes the AgentState dataclass,
        handling nested structures like the plan, goal stack, and tool stats. Excludes
        transient fields like `background_tasks`.
        """
        # Create a dictionary from the dataclass state
        state_dict = dataclasses.asdict(self.state)
        # Add a timestamp for when the state was saved
        state_dict["timestamp"] = datetime.now(timezone.utc).isoformat()
        # Ensure background_tasks (non-serializable Set[Task]) is removed
        state_dict.pop("background_tasks", None)
        # Convert defaultdict to regular dict for saving tool stats
        state_dict["tool_usage_stats"] = {
            k: dict(v) for k, v in self.state.tool_usage_stats.items()
        }
        # Convert PlanStep Pydantic objects to dictionaries for saving
        state_dict["current_plan"] = [
            step.model_dump(exclude_none=True) for step in self.state.current_plan
        ]
        # --- ADDED: Ensure goal_stack (list of dicts) is saved correctly ---
        state_dict["goal_stack"] = self.state.goal_stack # Already a list of dicts

        try:
            # Ensure the directory for the state file exists
            self.agent_state_file.parent.mkdir(parents=True, exist_ok=True)
            # Define a temporary file path for atomic write
            tmp_file = self.agent_state_file.with_suffix(f".tmp_{os.getpid()}") # Process-specific avoids
            collisions
            # Write to the temporary file asynchronously
            async with aiofiles.open(tmp_file, "w", encoding='utf-8') as f:
                # Dump the state dictionary to JSON with indentation
                await f.write(json.dumps(state_dict, indent=2, ensure_ascii=False))
                # Ensure data is written to the OS buffer
                await f.flush()
                # Ensure data is physically written to disk (crucial for crash recovery)
                try:
                    os.fsync(f.fileno())
                except OSError as e:
                    # fsync might fail on some systems/filesystems (e.g., network drives)
                    self.logger.warning(f"os.fsync failed during state save: {e} (Continuing, but save might
                    not be fully durable)")

            # Atomically replace the old state file with the new temporary file
            os.replace(tmp_file, self.agent_state_file)
            self.logger.debug(f"State saved atomically → {self.agent_state_file}")
        except Exception as e:
            # Log any errors during the save process
            self.logger.error(f"Failed to save agent state: {e}", exc_info=True)
            # Attempt to clean up the temporary file if it exists after an error
            if 'tmp_file' in locals() and tmp_file.exists():
                try:
                    os.remove(tmp_file)
                except OSError as rm_err:
                     self.logger.error(f"Failed to remove temporary state file {tmp_file}: {rm_err}")


    async def _load_agent_state(self) -> None:
        """
        Loads agent state from the JSON file.

        Handles file not found, JSON decoding errors, and potential mismatches
        between the saved state structure and the current `AgentState` dataclass
        (missing keys use defaults, extra keys are ignored with a warning).
        Ensures critical fields like thresholds and the **goal stack** are initialized
        even if loading fails.
        """
        # Check if the state file exists
        if not self.agent_state_file.exists():
            # If no file, initialize with defaults, ensuring thresholds are set
            self.state = AgentState(
                current_reflection_threshold=BASE_REFLECTION_THRESHOLD,
                current_consolidation_threshold=BASE_CONSOLIDATION_THRESHOLD
                # Other fields will use their dataclass defaults
            )
            self.logger.info("No prior state file found. Starting fresh with default state.")
            return
        # Try loading the state file
        try:
            async with aiofiles.open(self.agent_state_file, "r", encoding='utf-8') as f:
                # Read and parse the JSON data
                data = json.loads(await f.read())

            # Prepare keyword arguments for AgentState constructor
            kwargs: Dict[str, Any] = {}
            processed_keys = set() # Track keys successfully processed from the file

            # Iterate through the fields defined in the AgentState dataclass
            for fld in dataclasses.fields(AgentState):
                # Skip fields that are not meant to be initialized (like background_tasks)
                if not fld.init:
                    continue

                name = fld.name
                processed_keys.add(name)

                # Check if the field exists in the loaded data
                if name in data:
                    value = data[name]
                    # Handle specific fields needing type conversion or validation
                    if name == "current_plan":
                        try:
                            # Validate and convert saved plan steps back to PlanStep objects
                            if isinstance(value, list):
                                kwargs["current_plan"] = [PlanStep(**d) for d in value]
                            else:
                                raise TypeError("Saved plan is not a list")
                        except (ValidationError, TypeError) as e:
                            # If plan loading fails, reset to default plan
                            self.logger.warning(f"Plan reload failed: {e}. Resetting plan.")
                            kwargs["current_plan"] = [PlanStep(description=DEFAULT_PLAN_STEP)]
                    elif name == "tool_usage_stats":
                        # Reconstruct defaultdict structure for tool stats
                        dd = _default_tool_stats()
                        if isinstance(value, dict):
                            for k, v_dict in value.items():
                                if isinstance(v_dict, dict):
                                    # Ensure required keys exist with correct types
                                    dd[k]["success"] = int(v_dict.get("success", 0))
                                    dd[k]["failure"] = int(v_dict.get("failure", 0))
                                    dd[k]["latency_ms_total"] = float(v_dict.get("latency_ms_total", 0.0))
                        kwargs["tool_usage_stats"] = dd
                    # --- ADDED: Handle loading goal_stack ---
                    elif name == "goal_stack":
                        if isinstance(value, list):
                             # Basic validation: ensure items are dicts (more complex validation could be
                             added)
                             if all(isinstance(item, dict) for item in value):
                                 kwargs[name] = value
                             else:
                                 self.logger.warning("Invalid goal_stack format in saved state. Resetting.")
                                 kwargs[name] = [] # Reset to empty list
                        else:
                             self.logger.warning("goal_stack in saved state is not a list. Resetting.")
                             kwargs[name] = []
                    # Add handling for other complex types here if necessary in the future
                    else:
                        # Directly assign the loaded value if no special handling is needed
                        kwargs[name] = value
                else:
                    # Field defined in AgentState but missing in saved data
                    self.logger.debug(f"Field '{name}' not found in saved state. Using default.")
                    # Use the dataclass default factory or default value if defined
                    if fld.default_factory is not dataclasses.MISSING:
                        kwargs[name] = fld.default_factory()
                    elif fld.default is not dataclasses.MISSING:
                        kwargs[name] = fld.default
                    # Explicitly handle potentially missing thresholds if they didn't have defaults
                    elif name == "current_reflection_threshold":
                        kwargs[name] = BASE_REFLECTION_THRESHOLD
                    elif name == "current_consolidation_threshold":
                        kwargs[name] = BASE_CONSOLIDATION_THRESHOLD
                    # Handle missing goal stack/ID explicitly
                    elif name == "goal_stack":
                         kwargs[name] = [] # Default to empty list
                    elif name == "current_goal_id":
                         kwargs[name] = None # Default to None
                    # Otherwise, the field will be missing if it had no default and wasn't saved

            # Warn about extra keys found in the file but not defined in the current AgentState
            # This helps detect state format drift or old fields
            extra_keys = set(data.keys()) - processed_keys - {"timestamp"} # Exclude meta timestamp key
            if extra_keys:
                self.logger.warning(f"Ignoring unknown keys found in state file: {extra_keys}")

            # Create the AgentState instance using the processed keyword arguments
            # >>>>> PRESERVED ORIGINAL LOADING LOGIC FOR UNHANDLED FIELDS <<<<<
            # Note: This implicitly handles fields not explicitly checked above,
            # relying on the dataclass constructor to handle types if possible.
            # It's generally safer to handle complex types explicitly as done for plan/stats.
            temp_state = AgentState(**kwargs)

            # --- Validation and Correction after loading ---
            # Ensure mandatory fields (like thresholds) have values AFTER construction,
            # using defaults if somehow missed or loading failed for them.
            if not isinstance(temp_state.current_reflection_threshold, int):
                 self.logger.warning(f"Invalid loaded reflection threshold
                 ({temp_state.current_reflection_threshold}). Resetting to base.")
                 temp_state.current_reflection_threshold = BASE_REFLECTION_THRESHOLD
            else:
                 # Ensure loaded threshold is within bounds
                 temp_state.current_reflection_threshold = max(MIN_REFLECTION_THRESHOLD,
                 min(MAX_REFLECTION_THRESHOLD, temp_state.current_reflection_threshold))

            if not isinstance(temp_state.current_consolidation_threshold, int):
                 self.logger.warning(f"Invalid loaded consolidation threshold
                 ({temp_state.current_consolidation_threshold}). Resetting to base.")
                 temp_state.current_consolidation_threshold = BASE_CONSOLIDATION_THRESHOLD
            else:
                 # Ensure loaded threshold is within bounds
                 temp_state.current_consolidation_threshold = max(MIN_CONSOLIDATION_THRESHOLD,
                 min(MAX_CONSOLIDATION_THRESHOLD, temp_state.current_consolidation_threshold))

            # Ensure goal stack consistency
            if not isinstance(temp_state.goal_stack, list):
                self.logger.warning("Loaded goal_stack is not a list. Resetting.")
                temp_state.goal_stack = []
            # Ensure current_goal_id points to a goal actually in the stack (or is None)
            if temp_state.current_goal_id and not any(g.get('goal_id') == temp_state.current_goal_id for g in
            temp_state.goal_stack):
                 self.logger.warning(f"Loaded current_goal_id {_fmt_id(temp_state.current_goal_id)} not found
                 in loaded goal_stack. Resetting.")
                 # Set current_goal_id to the top of the loaded stack, or None if stack is empty
                 temp_state.current_goal_id = temp_state.goal_stack[-1].get('goal_id') if
                 temp_state.goal_stack else None

            # Assign the potentially corrected state
            self.state = temp_state
            self.logger.info(f"Loaded state from {self.agent_state_file}; current loop
            {self.state.current_loop}")

        except (json.JSONDecodeError, TypeError, FileNotFoundError) as e:
            # Handle common file loading or parsing errors
            self.logger.error(f"State load failed: {e}. Resetting to default state.", exc_info=True)
            # Reset to a clean default state on failure
            self.state = AgentState(
                current_reflection_threshold=BASE_REFLECTION_THRESHOLD,
                current_consolidation_threshold=BASE_CONSOLIDATION_THRESHOLD
            )
        except Exception as e:
            # Catch any other unexpected errors during state loading
            self.logger.critical(f"Unexpected error loading state: {e}. Resetting to default state.",
            exc_info=True)
            self.state = AgentState(
                current_reflection_threshold=BASE_REFLECTION_THRESHOLD,
                current_consolidation_threshold=BASE_CONSOLIDATION_THRESHOLD
            )


    # ---------------------------------------------------- tool-lookup helper --
    def _find_tool_server(self, tool_name: str) -> Optional[str]:
        """
        Finds an active server providing the specified tool via MCPClient's Server Manager.

        Args:
            tool_name: The original, potentially sanitized, tool name (e.g., "unified_memory:store_memory").

        Returns:
            The name of the active server providing the tool, "AGENT_INTERNAL" for the plan update tool,
            or None if the tool is not found on any active server.
        """
        # Ensure MCP Client and its server manager are available
        if not self.mcp_client or not self.mcp_client.server_manager:
            self.logger.error("MCP Client or Server Manager not available for tool lookup.")
            return None

        sm = self.mcp_client.server_manager
        # Check registered tools first (uses original tool names)
        if tool_name in sm.tools:
            server_name = sm.tools[tool_name].server_name
            # Verify the server is currently connected and considered active
            if server_name in sm.active_sessions:
                self.logger.debug(f"Found tool '{tool_name}' on active server '{server_name}'.")
                return server_name
            else:
                # Tool is known but its server is not currently active
                self.logger.debug(f"Server '{server_name}' for tool '{tool_name}' is registered but not
                active.")
                return None

        # Handle core tools if a server named "CORE" is active
        # (Assuming core tools follow a "core:" prefix convention)
        if tool_name.startswith("core:") and "CORE" in sm.active_sessions:
            self.logger.debug(f"Found core tool '{tool_name}' on active CORE server.")
            return "CORE"

        # Handle the agent's internal plan update tool
        if tool_name == AGENT_TOOL_UPDATE_PLAN:
            self.logger.debug(f"Internal tool '{tool_name}' does not require a server.")
            return "AGENT_INTERNAL" # Return a special marker

        # If the tool is not found in registered tools or core tools
        self.logger.debug(f"Tool '{tool_name}' not found on any active server.")
        return None


    # ------------------------------------------------------------ initialization --
    async def initialize(self) -> bool:
        """
        Initializes the Agent Master Loop.

        Loads prior agent state, fetches available tool schemas from MCPClient,
        validates the presence of essential tools (including goal tools),
        checks the validity of any loaded workflow and goal state,
        and sets the initial thought chain ID.

        Returns:
            True if initialization is successful, False otherwise.
        """
        self.logger.info("Initializing Agent loop …")
        # Load state from file first
        await self._load_agent_state()

        # Ensure context_id matches workflow_id if context_id was missing after load
        # This maintains consistency, assuming context usually maps 1:1 with workflow initially
        if self.state.workflow_id and not self.state.context_id:
            self.state.context_id = self.state.workflow_id
            self.logger.info(f"Initialized context_id from loaded workflow_id:
            {_fmt_id(self.state.workflow_id)}")

        try:
            # Check if MCPClient's server manager is ready
            if not self.mcp_client.server_manager:
                self.logger.error("MCP Client server manager not initialized.")
                return False

            # Fetch all available tool schemas formatted for the LLM (e.g., Anthropic format)
            all_tools = self.mcp_client.server_manager.format_tools_for_anthropic()

            # Manually inject the schema for the internal agent plan-update tool
            plan_step_schema = PlanStep.model_json_schema()
            # Remove 'title' if Pydantic adds it automatically, as it's not part of PlanStep definition
            plan_step_schema.pop('title', None)
            all_tools.append(
                {
                    "name": AGENT_TOOL_UPDATE_PLAN, # Use sanitized name expected by LLM
                    "description": "Replace the agent's current plan with a new list of plan steps. Use this
                    for significant replanning or error recovery.",
                    "input_schema": { # Anthropic uses 'input_schema'
                        "type": "object",
                        "properties": {
                            "plan": {
                                "type": "array",
                                "description": "Complete new plan as a list of PlanStep objects.",
                                "items": plan_step_schema, # Embed the generated PlanStep schema
                            }
                        },
                        "required": ["plan"],
                    },
                }
            )

            # Filter the fetched schemas to keep only those relevant to this agent
            # (Unified Memory tools and the internal agent tool)
            self.tool_schemas = []
            loaded_tool_names = set()
            for sc in all_tools:
                # Map the sanitized name back to the original for filtering
                original_name = self.mcp_client.server_manager.sanitized_to_original.get(sc["name"],
                sc["name"])
                # Keep if it's a unified_memory tool or the internal agent plan update tool
                if original_name.startswith("unified_memory:") or sc["name"] == AGENT_TOOL_UPDATE_PLAN:
                    self.tool_schemas.append(sc)
                    loaded_tool_names.add(original_name)

            self.logger.info(f"Loaded {len(self.tool_schemas)} relevant tool schemas: {loaded_tool_names}")

            # Verify that essential tools for core functionality are available
            essential = [
                TOOL_CREATE_WORKFLOW, TOOL_RECORD_ACTION_START, TOOL_RECORD_ACTION_COMPLETION,
                TOOL_RECORD_THOUGHT, TOOL_STORE_MEMORY, TOOL_GET_WORKING_MEMORY,
                TOOL_HYBRID_SEARCH, # Essential for enhanced context gathering
                TOOL_GET_CONTEXT,   # Essential for core context
                TOOL_REFLECTION,    # Essential for meta-cognition loop
                TOOL_CONSOLIDATION, # Essential for meta-cognition loop
                TOOL_GET_WORKFLOW_DETAILS, # Needed for setting default chain ID on load
                # --- ADDED: Check essential Goal Stack tools ---
                TOOL_PUSH_SUB_GOAL,
                TOOL_MARK_GOAL_STATUS,
                # TOOL_GET_GOAL_DETAILS, # Maybe not essential to *start*, but needed for context
            ]
            # Check availability using the server lookup helper
            missing = [t for t in essential if not self._find_tool_server(t)]
            if missing:
                # Log as error because agent functionality will be significantly impaired
                self.logger.error(f"Missing essential tools: {missing}. Agent functionality WILL BE
                impaired.")
                # Depending on desired strictness, could return False here to halt initialization

            # Check the validity of the workflow ID loaded from the state file
            # Determine the top workflow ID from the stack or the primary ID
            top_wf = (self.state.workflow_stack[-1] if self.state.workflow_stack else None) or
            self.state.workflow_id
            if top_wf and not await self._check_workflow_exists(top_wf):
                # If the loaded workflow doesn't exist anymore, reset workflow-specific state
                self.logger.warning(
                    f"Stored workflow '{_fmt_id(top_wf)}' not found in UMS; resetting workflow-specific
                    state."
                )
                # Preserve non-workflow specific state like stats and dynamic thresholds
                preserved_stats = self.state.tool_usage_stats
                pres_ref_thresh = self.state.current_reflection_threshold
                pres_con_thresh = self.state.current_consolidation_threshold
                # Reset state, keeping only preserved items
                self.state = AgentState(
                    tool_usage_stats=preserved_stats,
                    current_reflection_threshold=pres_ref_thresh,
                    current_consolidation_threshold=pres_con_thresh
                    # All other fields reset to defaults (including goal_stack, current_goal_id)
                )
                # Save the reset state immediately
                await self._save_agent_state()

            # --- ADDED: Validate loaded goal stack consistency ---
            await self._validate_goal_stack_on_load()

            # Initialize the current thought chain ID if a workflow exists but the chain ID is missing
            # This ensures thoughts are recorded correctly after loading state
            if self.state.workflow_id and not self.state.current_thought_chain_id:
                await self._set_default_thought_chain_id() # Attempt to find/set the primary chain

            self.logger.info("Agent loop initialization complete.")
            return True # Initialization successful
        except Exception as e:
            # Catch any unexpected errors during initialization
            self.logger.critical(f"Agent loop initialization failed: {e}", exc_info=True)
            return False # Initialization failed


    async def _set_default_thought_chain_id(self):
        """
        Sets the `current_thought_chain_id` in the agent state to the primary
        (usually first created) thought chain associated with the current workflow.
        """
        # Determine the current workflow ID from the stack or primary state
        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.debug("Cannot set default thought chain ID: No active workflow.")
            return # Cannot set if no workflow is active

        get_details_tool = TOOL_GET_WORKFLOW_DETAILS # Tool needed to fetch chains

        # Check if the necessary tool is available
        if self._find_tool_server(get_details_tool):
            try:
                # Call the tool internally to get workflow details, including thought chains
                details = await self._execute_tool_call_internal(
                    get_details_tool,
                    {
                        "workflow_id": current_wf_id,
                        "include_thoughts": True, # Must include thoughts to get chain info
                        "include_actions": False, # Not needed for this task
                        "include_artifacts": False,
                        "include_memories": False
                    },
                    record_action=False # Internal setup action, don't log as agent action
                )
                # Check if the tool call was successful and returned data
                if details.get("success"):
                    thought_chains = details.get("thought_chains")
                    # Check if thought_chains is a non-empty list
                    if isinstance(thought_chains, list) and thought_chains:
                        # Assume the first chain in the list is the primary one (usually ordered by creation
                        time)
                        first_chain = thought_chains[0]
                        chain_id = first_chain.get("thought_chain_id")
                        if chain_id:
                            # Set the state variable
                            self.state.current_thought_chain_id = chain_id
                            self.logger.info(f"Set current_thought_chain_id to primary chain:
                            {_fmt_id(self.state.current_thought_chain_id)} for workflow
                            {_fmt_id(current_wf_id)}")
                            return # Successfully set
                        else:
                             # Log warning if the found chain object is missing the ID
                             self.logger.warning(f"Primary thought chain found for workflow {current_wf_id},
                             but it lacks an ID in the details.")
                    else:
                         # Log warning if no chains were found for the workflow
```

```
                        self.logger.warning(f"Could not find any thought chains in details for workflow
                        {current_wf_id}.")
                else:
                    # Log the error message returned by the tool if it failed
                    self.logger.error(f"Tool '{get_details_tool}' failed while trying to get default thought
                    chain: {details.get('error')}")

            except Exception as e:
                # Log any exceptions encountered during the tool call itself
                self.logger.error(f"Error fetching workflow details for default chain: {e}", exc_info=False)
        else:
            # Log warning if the required tool is unavailable
            self.logger.warning(f"Cannot set default thought chain ID: Tool '{get_details_tool}'
            unavailable.")

        # Fallback message if the chain ID couldn't be set for any reason
        self.logger.info(f"Could not determine primary thought chain ID for WF {_fmt_id(current_wf_id)}. Will
        use default on first thought.")


    async def _check_workflow_exists(self, workflow_id: str) -> bool:
        """
        Efficiently checks if a given workflow ID exists using the UMS.

        Args:
            workflow_id: The workflow ID to check.

        Returns:
            True if the workflow exists, False otherwise.
        """
        self.logger.debug(f"Checking existence of workflow {_fmt_id(workflow_id)} using
        {TOOL_GET_WORKFLOW_DETAILS}.")
        tool_name = TOOL_GET_WORKFLOW_DETAILS
        # Check if the required tool is available
        if not self._find_tool_server(tool_name):
            self.logger.error(f"Cannot check workflow existence: Tool {tool_name} unavailable.")
            # If tool is unavailable, we cannot confirm existence, assume False for safety
            return False
        try:
            # Call get_workflow_details with minimal includes for efficiency
            result = await self._execute_tool_call_internal(
                tool_name,
                {
                    "workflow_id": workflow_id,
                    "include_actions": False,
                    "include_artifacts": False,
                    "include_thoughts": False,
                    "include_memories": False
                },
                record_action=False # This is an internal check
            )
            # If the tool call returns success=True, the workflow exists
            return isinstance(result, dict) and result.get("success", False)
        except ToolInputError as e:
            # A ToolInputError often indicates the ID was not found
            self.logger.debug(f"Workflow {_fmt_id(workflow_id)} likely not found (ToolInputError: {e}).")
            return False
        except Exception as e:
            # Log other errors encountered during the check
            self.logger.error(f"Error checking workflow {_fmt_id(workflow_id)} existence: {e}",
            exc_info=False)
            # Assume not found if an error occurs
            return False

    # <<< Start Integration Block: Goal Stack Validation Helper >>>
    async def _validate_goal_stack_on_load(self):
        """
        Validates the loaded goal stack against the UMS.

        Checks if the goals in the stack still exist and belong to the correct
        workflow context. Removes invalid goals from the stack.
        This assumes a UMS tool `get_goal_details` exists.
        """
        if not self.state.goal_stack:
            return # Nothing to validate if stack is empty

        tool_name = TOOL_GET_GOAL_DETAILS # Assumed UMS tool
        if not self._find_tool_server(tool_name):
            self.logger.warning(f"Cannot validate goal stack: Tool {tool_name} unavailable. Loaded stack may
            be invalid.")
            return

        current_wf_id = self.state.workflow_id
        if not current_wf_id:
            self.logger.warning("Cannot validate goal stack: No active workflow ID.")
            # Clear the stack if there's no workflow context
            self.state.goal_stack = []
            self.state.current_goal_id = None
            return

        valid_goals = []
        needs_update = False
        original_stack_ids = [g.get('goal_id') for g in self.state.goal_stack if g.get('goal_id')]

        for goal_dict in self.state.goal_stack:
            goal_id = goal_dict.get('goal_id')
            if not goal_id:
                self.logger.warning(f"Found goal with missing ID in loaded stack: {goal_dict}. Removing.")
                needs_update = True
                continue

            try:
                # Check goal existence and workflow association
                goal_details = await self._execute_tool_call_internal(
                    tool_name, {"goal_id": goal_id}, record_action=False
                )
                # Assuming the tool returns 'success' and 'workflow_id' if found
                if goal_details.get("success") and goal_details.get("workflow_id") == current_wf_id:
                    valid_goals.append(goal_dict) # Keep the goal if valid
                else:
                    self.logger.warning(f"Removing goal {_fmt_id(goal_id)} from stack: Not found or wrong
                    workflow ({goal_details.get('workflow_id')}) in UMS.")
                    needs_update = True
            except Exception as e:
                 self.logger.error(f"Error validating goal {_fmt_id(goal_id)}: {e}. Removing from stack.")
                 needs_update = True

        if needs_update:
            self.logger.info(f"Goal stack updated after validation. Original IDs: {[_fmt_id(g) for g in
            original_stack_ids]}, Valid IDs: {[_fmt_id(g.get('goal_id')) for g in valid_goals]}")
            self.state.goal_stack = valid_goals
            # Update current_goal_id to the top of the validated stack
            self.state.current_goal_id = self.state.goal_stack[-1].get('goal_id') if self.state.goal_stack
            else None
            self.logger.info(f"Reset current_goal_id after stack validation:
            {_fmt_id(self.state.current_goal_id)}")

        # If stack became empty after validation, ensure current_goal_id is None
        elif not self.state.goal_stack:
             self.state.current_goal_id = None

    # <<< End Integration Block: Goal Stack Validation Helper >>>

    # <<< Start Integration Block: Plan Cycle Detection Helper (Phase 1, Step 3) >>>
    def _detect_plan_cycle(self, plan: List[PlanStep]) -> bool:
        """
        Detects cyclic dependencies in the agent's plan using Depth First Search.

        Args:
            plan: The list of PlanStep objects representing the current plan.

        Returns:
            True if a cycle is detected, False otherwise.
        """
        if not plan: return False # Empty plan has no cycles

        adj: Dict[str, Set[str]] = defaultdict(set) # Adjacency list: step_id -> set(dependency_step_ids)
        plan_step_ids = {step.id for step in plan} # Set of all valid step IDs in the current plan

        # Build the adjacency list from depends_on relationships
        for step in plan:
            for dep_id in step.depends_on:
                # Only add dependency if the target step actually exists in the current plan
                if dep_id in plan_step_ids:
                    adj[step.id].add(dep_id)
                else:
```

```
                # Log if a dependency points to a non-existent step (potential issue)
                self.logger.warning(f"Plan step {_fmt_id(step.id)} depends on non-existent step
                {_fmt_id(dep_id)} in current plan.")

    # DFS state tracking:
    # path: nodes currently in the recursion stack for the current DFS path
    # visited: nodes that have been completely explored (all descendants visited)
    path: Set[str] = set()
    visited: Set[str] = set()

    def dfs(node_id: str) -> bool:
        """Recursive DFS function. Returns True if a cycle is detected."""
        path.add(node_id) # Mark node as currently visiting
        visited.add(node_id) # Mark node as visited

        # Explore neighbors (dependencies)
        for neighbor_id in adj[node_id]:
            if neighbor_id in path: # Cycle detected! Neighbor is already in the current path.
                self.logger.warning(f"Dependency cycle detected involving steps: {_fmt_id(node_id)} ->
                {_fmt_id(neighbor_id)}")
                return True
            if neighbor_id not in visited: # If neighbor not visited yet, recurse
                if dfs(neighbor_id):
                    return True # Propagate cycle detection signal up

        # Finished exploring node_id's descendants, remove from current path
        path.remove(node_id)
        return False # No cycle found starting from this node

    # Run DFS from each node in the plan to check all potential cycles
    for step_id in plan_step_ids:
        if step_id not in visited:
            if dfs(step_id):
                return True # Cycle found

    # If DFS completes for all nodes without finding a cycle
    return False
# <<< End Integration Block: Plan Cycle Detection Helper >>>

# ------------------------------------------------ dependency check --
async def _check_prerequisites(self, ids: List[str]) -> Tuple[bool, str]:
    """
    Checks if all specified prerequisite action IDs have status 'completed'.

    Args:
        ids: A list of action IDs to check.

    Returns:
        A tuple: (bool: True if all completed, False otherwise,
                  str: Reason for failure or "All dependencies completed.")
    """
    # If no IDs are provided, prerequisites are met by default
    if not ids:
        return True, "No dependencies listed."

    tool_name = TOOL_GET_ACTION_DETAILS # Tool needed to get action status
    # Check if the tool is available
    if not self._find_tool_server(tool_name):
        self.logger.error(f"Cannot check prerequisites: Tool {tool_name} unavailable.")
        return False, f"Tool {tool_name} unavailable."

    self.logger.debug(f"Checking prerequisites: {[_fmt_id(item_id) for item_id in ids]}")
    try:
        # Call the tool internally to get details for the specified action IDs
        res = await self._execute_tool_call_internal(
            tool_name,
            {"action_ids": ids, "include_dependencies": False}, # Don't need nested dependencies for this
            check
            record_action=False # Internal check
        )

        # Check if the tool call itself failed
        if not res.get("success"):
            error_msg = res.get("error", "Unknown error during dependency check.")
            self.logger.warning(f"Dependency check failed: {error_msg}")
            return False, f"Failed to check dependencies: {error_msg}"

        # Process the returned action details
```

```
            actions_found = res.get("actions", [])
            found_ids = {a.get("action_id") for a in actions_found}
            # Check if any requested dependency IDs were not found
            missing_ids = list(set(ids) - found_ids)
            if missing_ids:
                self.logger.warning(f"Dependency actions not found: {[_fmt_id(item_id) for item_id in
                missing_ids]}")
                return False, f"Dependency actions not found: {[_fmt_id(item_id) for item_id in
                missing_ids]}"

            # Check the status of each found dependency action
            incomplete_actions = []
            for action in actions_found:
                if action.get("status") != ActionStatus.COMPLETED.value:
                    # Collect details of incomplete actions for the reason message
                    incomplete_actions.append(
                        f"'{action.get('title', _fmt_id(action.get('action_id')))}' (Status:
                        {action.get('status', 'UNKNOWN')})"
                    )

            # If any actions are not completed, prerequisites are not met
            if incomplete_actions:
                reason = f"Dependencies not completed: {', '.join(incomplete_actions)}"
                self.logger.warning(reason)
                return False, reason

            # If all checks passed, prerequisites are met
            self.logger.debug("All dependencies completed.")
            return True, "All dependencies completed."

        except Exception as e:
            # Log any exceptions during the prerequisite check
            self.logger.error(f"Error during prerequisite check: {e}", exc_info=True)
            return False, f"Exception checking prerequisites: {str(e)}"


    # ---------------------------------------------------- action recording --
    async def _record_action_start_internal(
        self,
        tool_name: str,
        tool_args: Dict[str, Any],
        planned_dependencies: Optional[List[str]] = None,
    ) -> Optional[str]:
        """
        Records the start of an action using the UMS tool.

        Optionally records dependencies declared for this action.

        Args:
            tool_name: The name of the tool being executed.
            tool_args: The arguments passed to the tool.
            planned_dependencies: Optional list of action IDs this action depends on.

        Returns:
            The generated `action_id` if successful, otherwise None.
        """
        start_tool = TOOL_RECORD_ACTION_START # Tool for recording action start
        # Check if the recording tool is available
        if not self._find_tool_server(start_tool):
            self.logger.error(f"Cannot record action start: Tool '{start_tool}' unavailable.")
            return None

        # Get the current workflow ID from the state
        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 action start: No active workflow ID in state.")
            return None

        # Prepare the payload for the recording tool
        payload = {
            "workflow_id": current_wf_id,
            # Generate a basic title based on the tool name
            "title": f"Execute: {tool_name.split(':')[-1]}", # Use only the tool name part
            "action_type": ActionType.TOOL_USE.value, # Assume it's a tool use action
            "tool_name": tool_name,
            "tool_args": tool_args, # Pass the arguments being used
            "reasoning": f"Agent initiated tool call: {tool_name}", # Basic reasoning
            # Status will likely be set to IN_PROGRESS by the tool itself
        }

        action_id: Optional[str] = None
        try:
            # Call the recording tool internally (don't record this recording action)
            res = await self._execute_tool_call_internal(
                start_tool, payload, record_action=False
            )
            # Check if the recording was successful and returned an action ID
            if res.get("success"):
                action_id = res.get("action_id")
                if action_id:
                    self.logger.debug(f"Action started: {_fmt_id(action_id)} for tool {tool_name}")
                    # If dependencies were provided, record them *after* getting the action ID
                    if planned_dependencies:
                        await self._record_action_dependencies_internal(action_id, planned_dependencies)
                else:
                    # Log if the tool succeeded but didn't return an ID
                    self.logger.warning(f"Tool {start_tool} succeeded but returned no action_id.")
            else:
                # Log if the recording tool itself failed
                self.logger.error(f"Failed to record action start for {tool_name}: {res.get('error')}")

        except Exception as e:
            # Log any exceptions during the recording process
            self.logger.error(f"Exception recording action start for {tool_name}: {e}", exc_info=True)

        return action_id # Return the action ID or None


    async def _record_action_dependencies_internal(
        self,
        source_id: str, # The action being started
        target_ids: List[str], # The actions it depends on
    ) -> None:
        """
        Records dependencies (source_id REQUIRES target_id) in the UMS.

        Args:
            source_id: The ID of the action that has dependencies.
            target_ids: A list of action IDs that `source_id` depends on.
        """
        # Basic validation
        if not source_id or not target_ids:
            self.logger.debug("Skipping dependency recording: Missing source or target IDs.")
            return
        # Filter out empty/invalid target IDs and self-references
        valid_target_ids = {tid for tid in target_ids if tid and tid != source_id}
        if not valid_target_ids:
            self.logger.debug(f"No valid dependencies to record for source action {_fmt_id(source_id)}.")
            return

        dep_tool = TOOL_ADD_ACTION_DEPENDENCY # Tool for adding dependencies
        # Check tool availability
        if not self._find_tool_server(dep_tool):
            self.logger.error(f"Cannot record dependencies: Tool '{dep_tool}' unavailable.")
            return

        # Get current workflow ID (should match the source action's workflow)
        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(f"Cannot record dependencies for action {_fmt_id(source_id)}: No active
            workflow ID.")
            return

        self.logger.debug(f"Recording {len(valid_target_ids)} dependencies for action {_fmt_id(source_id)}:
        depends on {[_fmt_id(tid) for tid in valid_target_ids]}")

        # Create tasks to record each dependency concurrently
        tasks = []
        for target_id in valid_target_ids:
            args = {
                # workflow_id might be inferred by the tool, but pass for robustness
                # "workflow_id": current_wf_id, # Removed as UMS tool infers from action IDs
                "source_action_id": source_id,
                "target_action_id": target_id,
                "dependency_type": "requires", # Assuming 'requires' is the default/most common type here
            }
            # Call the dependency tool internally for each target ID
            task = asyncio.create_task(
                self._execute_tool_call_internal(dep_tool, args, record_action=False)
            )
            tasks.append(task)

        # Wait for all dependency recording tasks to complete
        results = await asyncio.gather(*tasks, return_exceptions=True)
        target_list = list(valid_target_ids) # Ensure consistent ordering for results
        # Log any failures encountered during dependency recording
        for i, res in enumerate(results):
            target_id = target_list[i] # Get corresponding target ID
            if isinstance(res, Exception):
                self.logger.error(f"Error recording dependency {_fmt_id(source_id)} -> {_fmt_id(target_id)}:
                {res}", exc_info=False)
            elif isinstance(res, dict) and not res.get("success"):
                # Log if the tool call itself failed
                self.logger.warning(f"Failed recording dependency {_fmt_id(source_id)} ->
                {_fmt_id(target_id)}: {res.get('error')}")
            # else: Successfully recorded


    async def _record_action_completion_internal(
        self,
        action_id: str,
        result: Dict[str, Any], # The final result dict from the tool execution
    ) -> None:
        """
        Records the completion or failure status and result for a given action ID.

        Args:
            action_id: The ID of the action to mark as completed/failed.
            result: The result dictionary returned by `_execute_tool_call_internal`.
        """
        completion_tool = TOOL_RECORD_ACTION_COMPLETION # Tool for recording completion
        # Check tool availability
        if not self._find_tool_server(completion_tool):
            self.logger.error(f"Cannot record action completion: Tool '{completion_tool}' unavailable.")
            return

        # Determine the final status based on the 'success' key in the result dict
        status = (
            ActionStatus.COMPLETED.value
            if isinstance(result, dict) and result.get("success")
            else ActionStatus.FAILED.value
        )

        # Get current workflow ID (should match the action's workflow)
        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(f"Cannot record completion for action {_fmt_id(action_id)}: No active
            workflow ID.")
            return

        # Prepare payload for the completion tool
        payload = {
            # Pass workflow_id for context, though tool might infer from action_id
            # "workflow_id": current_wf_id, # Removed as UMS tool infers from action ID
            "action_id": action_id,
            "status": status,
            # Pass the entire result dictionary to be stored/summarized by the UMS tool
            "tool_result": result,
            # Optionally, extract a summary here if needed by the tool, but UMS likely handles this
            # "summary": result.get("summary") or result.get("message") or str(result.get("data"))[:100]
        }

        try:
            # Call the completion tool internally
            completion_result = await self._execute_tool_call_internal(
                completion_tool, payload, record_action=False # Don't record this meta-action
            )
            # Log success or failure of the recording itself
            if completion_result.get("success"):
                self.logger.debug(f"Action completion recorded for {_fmt_id(action_id)} (Status: {status})")
            else:
                self.logger.error(f"Failed to record action completion for {_fmt_id(action_id)}:
                {completion_result.get('error')}")
        except Exception as e:
            # Log any exceptions during the completion recording call
            self.logger.error(f"Exception recording action completion for {_fmt_id(action_id)}: {e}",
            exc_info=True)

    # ---------------------------------------------------- auto-link helper --
    async def _run_auto_linking(
        self,
        memory_id: str, # The ID of the newly created/updated memory
        *,
        workflow_id: Optional[str], # Snapshotted workflow ID
        context_id: Optional[str], # Snapshotted context ID (unused here but passed)
    ) -> None:
        """
        Background task to find semantically similar memories and automatically
        create links to them. Uses richer link types based on memory types.
        """
        # (Keep the integrated _run_auto_linking method logic)
        # Check if the agent's current workflow matches the one snapshotted when the task was created
        # Also check if shutdown has been requested
        if workflow_id != self.state.workflow_id or self._shutdown_event.is_set():
            self.logger.debug(f"Skipping auto-linking for {_fmt_id(memory_id)}: Workflow changed
            ({_fmt_id(self.state.workflow_id)} vs {_fmt_id(workflow_id)}) or shutdown signaled.")
            return

        try:
            # Validate inputs
            if not memory_id or not workflow_id:
                self.logger.debug(f"Skipping auto-linking: Missing memory_id ({_fmt_id(memory_id)}) or
                workflow_id ({_fmt_id(workflow_id)}).")
                return

            # Introduce a small random delay to distribute load
            await asyncio.sleep(random.uniform(*AUTO_LINKING_DELAY_SECS))
            # Check shutdown again after sleep
            if self._shutdown_event.is_set(): return

            self.logger.debug(f"Attempting auto-linking for memory {_fmt_id(memory_id)} in workflow
            {_fmt_id(workflow_id)}...")

            # 1. Get details of the source memory (the one just created/updated)
            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
            )
            # Ensure retrieval succeeded and the memory still belongs to the expected workflow
            if not source_mem_details_result.get("success") or source_mem_details_result.get("workflow_id")
            != workflow_id:
                self.logger.warning(f"Auto-linking failed for {_fmt_id(memory_id)}: Couldn't retrieve source
                memory or workflow mismatch.")
                return
            source_mem = source_mem_details_result # Result is the memory dict

            # 2. Determine text for similarity search (use description or truncated content)
            query_text = source_mem.get("description", "") or source_mem.get("content", "")[:200] # Limit
            content length
            if not query_text:
                self.logger.debug(f"Skipping auto-linking for {_fmt_id(memory_id)}: No description or content
                for query.")
                return

            # 3. Perform semantic search for similar memories within the same workflow
            # Prefer hybrid search if available, fall back to semantic
            search_tool = TOOL_HYBRID_SEARCH if self._find_tool_server(TOOL_HYBRID_SEARCH) else
            TOOL_SEMANTIC_SEARCH
            if not self._find_tool_server(search_tool):
                self.logger.warning(f"Skipping auto-linking: Tool {search_tool} unavailable.")
                return

            search_args = {
                "workflow_id": workflow_id, # Search within the snapshotted workflow
                "query": query_text,
                "limit": self.auto_linking_max_links + 1, # Fetch one extra to filter self
                "threshold": self.auto_linking_threshold, # Use configured threshold
                "include_content": False # Don't need full content of similar items
            }
            # Adjust weights if using hybrid search (prioritize semantic similarity)
            if search_tool == TOOL_HYBRID_SEARCH:
                search_args.update({"semantic_weight": 0.8, "keyword_weight": 0.2})

            similar_results = await self._execute_tool_call_internal(
                search_tool, search_args, record_action=False
            )
            if not similar_results.get("success"):
                self.logger.warning(f"Auto-linking search failed for {_fmt_id(memory_id)}:
                {similar_results.get('error')}")
                return

            # 4. Process results and create links
            link_count = 0
            # Determine which key holds the similarity score based on the tool used
            score_key = "hybrid_score" if search_tool == TOOL_HYBRID_SEARCH else "similarity"

            for similar_mem_summary in similar_results.get("memories", []):
                # Check shutdown flag frequently during loop
                if self._shutdown_event.is_set(): break

                target_id = similar_mem_summary.get("memory_id")
                similarity_score = similar_mem_summary.get(score_key, 0.0)

                # Skip linking to self
                if not target_id or target_id == memory_id: continue

                # 5. Get details of the potential target memory for richer link type inference
                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
                )
                # Ensure target retrieval succeeded and it's in the same workflow
                if not target_mem_details_result.get("success") or
                target_mem_details_result.get("workflow_id") != workflow_id:
                    self.logger.debug(f"Skipping link target {_fmt_id(target_id)}: Not found or workflow
                    mismatch.")
                    continue
                target_mem = target_mem_details_result

                # 6. Infer a more specific link type based on memory types
                inferred_link_type = LinkType.RELATED.value # Default link type
                source_type = source_mem.get("memory_type")
                target_type = target_mem.get("memory_type")

                # Example inference rules (can be expanded)
                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 # Or maybe GENERALIZES/SPECIALIZES?
                elif source_type == MemoryType.QUESTION.value and target_type == MemoryType.FACT.value:
                inferred_link_type = LinkType.REFERENCES.value
                # >>>>> PRESERVED ORIGINAL LINK TYPE RULES <<<<<
                # (Can add more rules here based on analysis)
                # elif source_type == MemoryType.HYPOTHESIS.value and target_type ==
                MemoryType.EVIDENCE.value: inferred_link_type = LinkType.SUPPORTS.value # Assuming evidence
                supports hypothesis
                # elif source_type == MemoryType.EVIDENCE.value and target_type ==
                MemoryType.HYPOTHESIS.value: inferred_link_type = LinkType.SUPPORTS.value
                # ... add other rules ...

                # 7. Create the link using the UMS tool
                link_tool_name = TOOL_CREATE_LINK
                if not self._find_tool_server(link_tool_name):
                    self.logger.warning(f"Cannot create link: Tool {link_tool_name} unavailable.")
                    break # Stop trying if link tool is missing

                link_args = {
                    # workflow_id is usually inferred by the tool from memory IDs
                    "source_memory_id": memory_id,
                    "target_memory_id": target_id,
                    "link_type": inferred_link_type,
                    "strength": round(similarity_score, 3), # Use similarity score as strength
                    "description": f"Auto-link ({inferred_link_type}) based on similarity ({score_key})"
                }
                link_result = await self._execute_tool_call_internal(
                    link_tool_name, link_args, record_action=False # Don't record link creation as primary
                    action
                )
```

```
                # Log success or failure of link creation
                if link_result.get("success"):
                    link_count += 1
                    self.logger.debug(f"Auto-linked memory {_fmt_id(memory_id)} to {_fmt_id(target_id)}
                    ({inferred_link_type}, score: {similarity_score:.2f})")
                else:
                    # Log failure, but continue trying other potential links
                    self.logger.warning(f"Failed to auto-create link
                    {_fmt_id(memory_id)}->{_fmt_id(target_id)}: {link_result.get('error')}")

                # Stop if max links reached for this source memory
                if link_count >= self.auto_linking_max_links:
                    self.logger.debug(f"Reached auto-linking limit ({self.auto_linking_max_links}) for memory
                    {_fmt_id(memory_id)}.")
                    break
                # Small delay between creating links if multiple are found
                await asyncio.sleep(0.1)

        except Exception as e:
            # Catch and log any errors occurring within the background task itself
            self.logger.warning(f"Error in auto-linking task for {_fmt_id(memory_id)}: {e}", exc_info=False)


    # ---------------------------------------------------- promotion helper --
    async def _check_and_trigger_promotion(
        self,
        memory_id: str, # The ID of the memory to check
        *,
        workflow_id: Optional[str], # Snapshotted workflow ID
        context_id: Optional[str], # Snapshotted context ID (unused but passed)
    ):
        """
        Checks if a specific memory meets criteria for promotion to a higher
        cognitive level and calls the UMS tool to perform the promotion if eligible.
        Intended to be run as a background task.
        """
        # (Keep the integrated _check_and_trigger_promotion method logic)
        # Abort if workflow context has changed or shutdown is signaled
        if workflow_id != self.state.workflow_id or self._shutdown_event.is_set():
            self.logger.debug(f"Skipping promotion check for {_fmt_id(memory_id)}: Workflow changed
            ({_fmt_id(self.state.workflow_id)} vs {_fmt_id(workflow_id)}) or shutdown.")
            return

        promotion_tool_name = TOOL_PROMOTE_MEM # Tool for checking/promoting
        # Basic validation
        if not memory_id or not self._find_tool_server(promotion_tool_name):
            self.logger.debug(f"Skipping promotion check for {_fmt_id(memory_id)}: Invalid ID or tool
            '{promotion_tool_name}' unavailable.")
            return

        try:
            # Optional slight delay
            await asyncio.sleep(random.uniform(0.1, 0.4))
            # Check shutdown again after sleep
            if self._shutdown_event.is_set(): return

            self.logger.debug(f"Checking promotion potential for memory {_fmt_id(memory_id)} in workflow
            {_fmt_id(workflow_id)}...")
            # Execute the promotion check tool internally
            # Workflow ID is likely inferred by the tool from memory_id
            promotion_result = await self._execute_tool_call_internal(
                promotion_tool_name, {"memory_id": memory_id}, record_action=False # Don't record check as
                action
            )

            # Log the outcome of the promotion check
            if promotion_result.get("success"):
                if promotion_result.get("promoted"):
                    # Log successful promotion clearly
                    self.logger.info(f"Memory {_fmt_id(memory_id)} promoted from
                    {promotion_result.get('previous_level')} to {promotion_result.get('new_level')}.",
                    emoji_key="arrow_up")
                else:
                    # Log the reason if promotion didn't occur but check was successful
                    self.logger.debug(f"Memory {_fmt_id(memory_id)} not promoted:
                    {promotion_result.get('reason')}")
            else:
                # Log if the promotion check tool itself failed
                self.logger.warning(f"Promotion check tool failed for {_fmt_id(memory_id)}:
                {promotion_result.get('error')}")

        except Exception as e:
            # Log any exceptions within the promotion check task
            self.logger.warning(f"Error in memory promotion check task for {_fmt_id(memory_id)}: {e}",
            exc_info=False)


    # ---------------------------------------------------- execute tool call --
    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]:
        """
        Central handler for executing tool calls via MCPClient.

        Includes:
        - Server lookup.
        - Automatic injection of workflow/context IDs.
        - Prerequisite dependency checking.
        - Handling of the internal AGENT_TOOL_UPDATE_PLAN.
        - Optional recording of action start/completion/dependencies.
        - Retry logic for idempotent tools.
        - Result parsing and standardization.
        - **Enhanced error handling/categorization** and state updates (last_error_details).
        - Triggering relevant background tasks (auto-linking, promotion check).
        - Updating last_action_summary state.
        - Handling workflow and **goal stack** side effects.
        """
        # <<< Start Integration Block: Enhance _execute_tool_call_internal (Goal Stack Side Effects) >>>
        # --- Step 1: Server Lookup ---
        target_server = self._find_tool_server(tool_name)
        # Handle case where tool server is not found, except for the internal agent tool
        if not target_server and tool_name != AGENT_TOOL_UPDATE_PLAN:
            err = f"Tool server unavailable for {tool_name}"
            self.logger.error(err)
            # Set error details for the main loop to see - Enhanced Category
            self.state.last_error_details = {"tool": tool_name, "error": err, "type": "ServerUnavailable",
            "status_code": 503}
            # Return a failure dictionary consistent with other error returns
            return {"success": False, "error": err, "status_code": 503} # 503 Service Unavailable

        # --- Step 2: Context Injection ---
        # Get current workflow context IDs from state
        current_wf_id = (self.state.workflow_stack[-1] if self.state.workflow_stack else
        self.state.workflow_id)
        current_ctx_id = self.state.context_id
        current_goal_id = self.state.current_goal_id # Get current goal ID

        # Make a copy to avoid modifying the original arguments dict passed in
        final_arguments = arguments.copy()
        # Inject workflow_id if missing and relevant for the tool
        if (
            final_arguments.get("workflow_id") is None # Only if not already provided
            and current_wf_id # Only if a workflow is active
            and tool_name # Check if tool_name is valid
            not in { # Exclude tools that don't operate on a specific workflow
                TOOL_CREATE_WORKFLOW, # Creates a new one
                TOOL_LIST_WORKFLOWS, # Lists across potentially many
                "core:list_servers", # Core MCP tool
                "core:get_tool_schema", # Core MCP tool
                AGENT_TOOL_UPDATE_PLAN, # Internal agent tool
                TOOL_PUSH_SUB_GOAL, # Assumes UMS handles workflow association
                TOOL_MARK_GOAL_STATUS, # Uses goal_id, workflow inferred
                TOOL_GET_GOAL_DETAILS, # Uses goal_id
            }
        ):
            final_arguments["workflow_id"] = current_wf_id
        # Inject context_id if missing and relevant for the tool
        if (
            final_arguments.get("context_id") is None # Only if not already provided
            and current_ctx_id # Only if a context ID is set
            and tool_name # Check if tool_name is valid
            in { # Tools known to operate on a specific cognitive context ID
                TOOL_GET_WORKING_MEMORY,
                TOOL_OPTIMIZE_WM,
                TOOL_AUTO_FOCUS,
                # Add others here if they accept/require context_id
            }
        ):
            final_arguments["context_id"] = current_ctx_id
        # Inject current thought chain ID for thought recording if not specified
        if (
            final_arguments.get("thought_chain_id") is None # Only if not already provided
            and self.state.current_thought_chain_id # Only if a chain ID is set
            and tool_name == TOOL_RECORD_THOUGHT # Only for the record_thought tool
        ):
            final_arguments["thought_chain_id"] = self.state.current_thought_chain_id
        # --- ADDED: Inject parent goal ID for push_sub_goal if not provided ---
        if (
             final_arguments.get("parent_goal_id") is None # Only if not already provided
             and current_goal_id # Only if a goal is currently active
             and tool_name == TOOL_PUSH_SUB_GOAL # Only for this tool
        ):
             final_arguments["parent_goal_id"] = current_goal_id

        # --- Step 3: Dependency Check ---
        # If dependencies were declared for this action, check them first
        if planned_dependencies:
            ok, reason = await self._check_prerequisites(planned_dependencies)
            if not ok:
                # If dependencies not met, log warning, set error state, and return failure
                err_msg = f"Prerequisites not met for {tool_name}: {reason}"
                self.logger.warning(err_msg)
                # Store detailed error info for the LLM - Enhanced Category
                self.state.last_error_details = {"tool": tool_name, "error": err_msg, "type":
                "DependencyNotMetError", "dependencies": planned_dependencies, "status_code": 412}
                # Signal that replanning is needed due to dependency failure
                self.state.needs_replan = True
                return {"success": False, "error": err_msg, "status_code": 412} # 412 Precondition Failed
            else:
                # Log if dependencies were checked and met
                self.logger.info(f"Prerequisites {[_fmt_id(dep) for dep in planned_dependencies]} met for
                {tool_name}.")

        # --- Step 4: Handle Internal Agent Tool ---
        # Directly handle the AGENT_TOOL_UPDATE_PLAN without calling MCPClient
        if tool_name == AGENT_TOOL_UPDATE_PLAN:
            try:
                new_plan_data = final_arguments.get("plan", [])
                # Validate the structure of the provided plan data
                if not isinstance(new_plan_data, list):
                    raise ValueError("`plan` argument must be a list of step objects.")
                # Convert list of dicts to list of PlanStep objects (validates structure)
                validated_plan = [PlanStep(**p) for p in new_plan_data]

                # --- Plan Cycle Detection ---
                if self._detect_plan_cycle(validated_plan):
                    err_msg = "Proposed plan contains a dependency cycle."
                    self.logger.error(err_msg)
                    self.state.last_error_details = {"tool": tool_name, "error": err_msg, "type":
                    "PlanValidationError", "proposed_plan": new_plan_data}
                    self.state.needs_replan = True # Force replan again
                    return {"success": False, "error": err_msg}

                # Replace the agent's current plan
                self.state.current_plan = validated_plan
                # Plan was explicitly updated, so replan flag can be cleared
                self.state.needs_replan = False
                self.logger.info(f"Internal plan update successful. New plan has {len(validated_plan)}
                steps.")
                # Clear any previous errors after a successful plan update
                self.state.last_error_details = None
                self.state.consecutive_error_count = 0
                return {"success": True, "message": f"Plan updated with {len(validated_plan)} steps."}
            except (ValidationError, TypeError, ValueError) as e:
                # Handle errors during plan validation or application
                err_msg = f"Failed to validate/apply new plan: {e}"
                self.logger.error(err_msg)
                # Store error details for the LLM - Enhanced Category
                self.state.last_error_details = {"tool": tool_name, "error": err_msg, "type":
                "PlanUpdateError", "proposed_plan": final_arguments.get("plan")}
                # Increment error count for internal failures too? Decide policy. Yes, for now.
                self.state.consecutive_error_count += 1
                # Failed plan update requires another attempt at planning
                self.state.needs_replan = True
                return {"success": False, "error": err_msg}

        # --- Step 5: Record Action Start (Optional) ---
        action_id: Optional[str] = None
        # Determine if this tool call should be recorded as a primary agent action
        # Exclude internal/meta tools and calls where record_action is explicitly False
        should_record = record_action and tool_name not in self._INTERNAL_OR_META_TOOLS
        if should_record:
            # Call internal helper to record the action start and dependencies
            action_id = await self._record_action_start_internal(
                 tool_name, final_arguments, planned_dependencies # Pass potentially modified args and
                 dependencies
            )
            # Note: _record_action_start_internal now handles calling _record_action_dependencies_internal

        # --- Step 6: Execute Tool Call (with Retries) ---
        # Define the actual async function to call the tool via MCPClient
        async def _do_call():
            # Ensure None values are stripped *before* sending to MCPClient execute_tool
            # Although MCPClient likely handles this, this adds robustness.
            call_args = {k: v for k, v in final_arguments.items() if v is not None}
            # Target server must be valid here because AGENT_INTERNAL was handled earlier
            return await self.mcp_client.execute_tool(target_server, tool_name, call_args)

        # Get the stats dictionary for this specific tool
        record_stats = self.state.tool_usage_stats[tool_name]
        # Decide if the tool is safe to automatically retry on failure
        idempotent = tool_name in {
            # Read-only operations are generally safe to retry
            TOOL_GET_CONTEXT, TOOL_GET_MEMORY_BY_ID, TOOL_SEMANTIC_SEARCH,
            TOOL_HYBRID_SEARCH, TOOL_GET_ACTION_DETAILS, TOOL_LIST_WORKFLOWS,
            TOOL_COMPUTE_STATS, TOOL_GET_WORKING_MEMORY, TOOL_GET_LINKED_MEMORIES,
            TOOL_GET_ARTIFACTS, TOOL_GET_ARTIFACT_BY_ID, TOOL_GET_ACTION_DEPENDENCIES,
            TOOL_GET_THOUGHT_CHAIN, TOOL_GET_WORKFLOW_DETAILS,
            TOOL_GET_GOAL_DETAILS, # Getting goal details is idempotent (NEW)
            # Some meta operations might be considered retry-safe
            TOOL_SUMMARIZE_TEXT,
        }

        start_ts = time.time() # Record start time for latency calculation
        res = {} # Initialize result dictionary

        try:
            # Execute the tool call using the retry wrapper
            raw = await self._with_retries(
                _do_call,
                max_retries=3 if idempotent else 1, # Retry only idempotent tools (3 attempts total)
                # Specify exceptions that should trigger a retry attempt
                retry_exceptions=(
                    ToolError, ToolInputError, # Specific MCP errors
                    asyncio.TimeoutError, ConnectionError, # Common network issues
                    APIConnectionError, RateLimitError, APIStatusError, # Anthropic/LLM network/API issues
                ),
            )
            # Calculate execution latency
            latency_ms = (time.time() - start_ts) * 1000
            record_stats["latency_ms_total"] += latency_ms

            # --- Step 7: Process and Standardize Result ---
            # Handle different result formats returned by MCPClient/tools
            if isinstance(raw, dict) and ("success" in raw or "isError" in raw):
                # Assume standard MCP result format with success/isError flag
                is_error = raw.get("isError", not raw.get("success", True))
                # Extract content or error message
                content = raw.get("content", raw.get("error", raw.get("data")))
                if is_error:
                    res = {"success": False, "error": str(content), "status_code": raw.get("status_code")}
                else:
                    # If content itself has a standard structure, use it directly
                    if isinstance(content, dict) and "success" in content:
                        res = content
                    else: # Otherwise, wrap the content under a 'data' key for consistency
                        res = {"success": True, "data": content}
            elif isinstance(raw, dict): # Handle plain dictionaries without explicit success/isError
                 # Assume success if no error indicators present
                 res = {"success": True, "data": raw}
            else:
                # Handle non-dict results (e.g., simple strings, numbers, booleans)
                res = {"success": True, "data": raw}

            # --- Step 8: State Updates and Background Triggers on SUCCESS ---
            if res.get("success"):
                # Update success stats for the tool
                record_stats["success"] += 1

                # --- Background Triggers Integration ---
                # Snapshot the workflow ID *before* potentially starting background tasks
                # (This ensures tasks operate on the workflow active during the trigger event)
                current_wf_id_snapshot = self.state.workflow_stack[-1] if self.state.workflow_stack else
                self.state.workflow_id  # noqa: F841

                # Trigger auto-linking after storing/updating a memory or recording an artifact with a linked
                memory
                if tool_name in [TOOL_STORE_MEMORY, TOOL_UPDATE_MEMORY] and res.get("memory_id"):
                    mem_id = res["memory_id"]
                    self.logger.debug(f"Queueing auto-link check for memory {_fmt_id(mem_id)}")
                    # Start background task, passing the memory ID
                    self._start_background_task(AgentMasterLoop._run_auto_linking, memory_id=mem_id)
                    # Note: _start_background_task automatically snapshots workflow_id/context_id
                if tool_name == TOOL_RECORD_ARTIFACT and res.get("linked_memory_id"):
                    linked_mem_id = res["linked_memory_id"]
                    self.logger.debug(f"Queueing auto-link check for memory linked to artifact:
                    {_fmt_id(linked_mem_id)}")
                    self._start_background_task(AgentMasterLoop._run_auto_linking, memory_id=linked_mem_id)

                # Trigger promotion check after retrieving memories
                if tool_name in [TOOL_GET_MEMORY_BY_ID, TOOL_QUERY_MEMORIES, TOOL_HYBRID_SEARCH,
                TOOL_SEMANTIC_SEARCH, TOOL_GET_WORKING_MEMORY]:
                    mem_ids_to_check = set() # Use set to avoid duplicate checks
                    potential_mems = []
                    # Extract memory IDs from various possible result structures
                    if tool_name == TOOL_GET_MEMORY_BY_ID:
                        # Result might be the memory dict directly or nested under 'data'
                        mem_data = res if "memory_id" in res else res.get("data", {})
                        if isinstance(mem_data, dict): potential_mems = [mem_data]
                    elif tool_name == TOOL_GET_WORKING_MEMORY:
                        potential_mems = res.get("working_memories", [])
                        # Also check the focal memory if present
                        focus_id = res.get("focal_memory_id")
                        if focus_id: mem_ids_to_check.add(focus_id)
                    else: # Query/Search results typically under 'memories' key
                         potential_mems = res.get("memories", [])

                    # Add IDs from the list/dict structures found
                    if isinstance(potential_mems, list):
                        # Limit checks to the top few most relevant results to avoid overload
                        mem_ids_to_check.update(
                            m.get("memory_id") for m in potential_mems[:3] # Check top 3 retrieved
                            if isinstance(m, dict) and m.get("memory_id") # Ensure it's a dict with an ID
                        )

                    # Start background tasks for each unique, valid memory ID found
                    for mem_id in filter(None, mem_ids_to_check): # Filter out any None IDs
                         self.logger.debug(f"Queueing promotion check for retrieved memory
                         {_fmt_id(mem_id)}")
                         self._start_background_task(AgentMasterLoop._check_and_trigger_promotion,
                         memory_id=mem_id)
                # --- Background Triggers Integration End ---

                # Update current thought chain ID if a new one was just created successfully
                if tool_name == TOOL_CREATE_THOUGHT_CHAIN and res.get("success"):
                    # Find the chain ID in the result (might be root or under 'data')
                    chain_data = res if "thought_chain_id" in res else res.get("data", {})
                    if isinstance(chain_data, dict):
                        new_chain_id = chain_data.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:
                            {_fmt_id(new_chain_id)}")
            else: # Tool failed
                # Update failure stats
                record_stats["failure"] += 1
                # Ensure error details are captured from the result for the LLM context
                # Enhance error details with categorization
                error_type = "ToolExecutionError" # Default category
                status_code = res.get("status_code")
                error_message = res.get("error", "Unknown failure")
                if status_code == 412: error_type = "DependencyNotMetError"
                elif status_code == 503: error_type = "ServerUnavailable"
                elif "input" in str(error_message).lower() or "validation" in str(error_message).lower():
                error_type = "InvalidInputError" # Basic keyword check
                elif "timeout" in str(error_message).lower(): error_type = "NetworkError" # Assuming timeout
                implies network
                # --- ADDED: Categorize goal management errors ---
                elif tool_name in [TOOL_PUSH_SUB_GOAL, TOOL_MARK_GOAL_STATUS] and ("not found" in
                str(error_message).lower() or "invalid" in str(error_message).lower()):
                     error_type = "GoalManagementError"

                self.state.last_error_details = {
                    "tool": tool_name,
                    "args": arguments, # Log the arguments that caused failure
                    "error": error_message,
                    "status_code": status_code,
                    "type": error_type # Store the categorized error type
                }
                # Log the categorized error
                self.logger.warning(f"Tool {tool_name} failed. Type: {error_type}, Error: {error_message}")


            # --- Step 9: Update Last Action Summary ---
            # Create a concise summary of the action's outcome for the next prompt
            summary = ""
            if res.get("success"):
                # Try to find a meaningful summary field in the result or its 'data' payload
                summary_keys = ["summary", "message", "memory_id", "action_id", "artifact_id", "link_id",
                "chain_id", "state_id", "report", "visualization", "goal_id"] # Added goal_id
                data_payload = res.get("data", res) # Check 'data' key or root level
                if isinstance(data_payload, dict):
                    for k in summary_keys:
                        if k in data_payload and data_payload[k]:
                            # Format IDs concisely, use string value for others
                            summary = f"{k}: {_fmt_id(data_payload[k]) if 'id' in k else
                            str(data_payload[k])}"
                            break
                    else: # Fallback if no specific key found in dict
                        data_str = str(data_payload)[:70] # Preview the dict
                        summary = f"Success. Data: {data_str}..." if len(str(data_payload)) > 70 else
                        f"Success. Data: {data_str}"
                else: # Handle non-dict data payload
                     data_str = str(data_payload)[:70] # Preview the data
                     summary = f"Success. Data: {data_str}..." if len(str(data_payload)) > 70 else f"Success.
                     Data: {data_str}"
            else: # If failed
                # Use the structured error details for a more informative summary
                err_type = self.state.last_error_details.get("type", "Unknown") if
                self.state.last_error_details else "Unknown"
                err_msg = str(res.get('error', 'Unknown Error'))[:100]
                summary = f"Failed ({err_type}): {err_msg}" # Include error type
                if res.get('status_code'): summary += f" (Code: {res['status_code']})" # Add status code if
                available

            # Update the state variable
            self.state.last_action_summary = f"{tool_name} -> {summary}"
            # Log the outcome
            self.logger.info(self.state.last_action_summary, emoji_key="checkered_flag" if res.get('success')
            else "warning")


        # --- Step 10: Exception Handling for Tool Call/Retries ---
        # Updated Exception handling to add error categorization
        except (ToolError, ToolInputError) as e:
            # Handle specific MCP exceptions caught during execution or retries
            err_str = str(e); status_code = getattr(e, 'status_code', None)
            # Determine category based on type and status code
            error_type = "InvalidInputError" if isinstance(e, ToolInputError) else "ToolInternalError"
            if status_code == 412: error_type = "DependencyNotMetError"
            self.logger.error(f"Tool Error executing {tool_name}: {err_str}", exc_info=False) # Don't need
            full trace for these
            res = {"success": False, "error": err_str, "status_code": status_code}
            record_stats["failure"] += 1 # Record failure
            # Store categorized error details for the LLM
            self.state.last_error_details = {"tool": tool_name, "args": arguments, "error": err_str, "type":
            error_type, "status_code": status_code}
            self.state.last_action_summary = f"{tool_name} -> Failed ({error_type}): {err_str[:100]}"

        except APIConnectionError as e:
            err_str = f"LLM API Connection Error: {e}"
            self.logger.error(err_str, exc_info=False)
            res = {"success": False, "error": err_str}
            record_stats["failure"] += 1
            self.state.last_error_details = {"tool": tool_name, "args": arguments, "error": err_str, "type":
            "NetworkError"}
            self.state.last_action_summary = f"{tool_name} -> Failed: NetworkError"
        except RateLimitError as e:
            err_str = f"LLM Rate Limit Error: {e}"
            self.logger.error(err_str, exc_info=False)
            res = {"success": False, "error": err_str}
            record_stats["failure"] += 1
            self.state.last_error_details = {"tool": tool_name, "args": arguments, "error": err_str, "type":
            "APILimitError"}
            self.state.last_action_summary = f"{tool_name} -> Failed: APILimitError"
        except APIStatusError as e:
            err_str = f"LLM API Error {e.status_code}: {e.message}"
            self.logger.error(f"Anthropic API status error: {e.status_code} - {e.response}", exc_info=False)
            res = {"success": False, "error": err_str, "status_code": e.status_code}
            record_stats["failure"] += 1
            self.state.last_error_details = {"tool": tool_name, "args": arguments, "error": err_str, "type":
            "APIError", "status_code": e.status_code}
            self.state.last_action_summary = f"{tool_name} -> Failed: APIError ({e.status_code})"
        except asyncio.TimeoutError as e: # Catch timeouts from retry wrapper or internal calls
            err_str = f"Operation timed out: {e}"
            self.logger.error(f"Timeout executing {tool_name}: {err_str}", exc_info=False)
            res = {"success": False, "error": err_str}
            record_stats["failure"] += 1
            self.state.last_error_details = {"tool": tool_name, "args": arguments, "error": err_str, "type":
            "TimeoutError"}
            self.state.last_action_summary = f"{tool_name} -> Failed: Timeout"

        except asyncio.CancelledError:
             # Handle task cancellation gracefully (e.g., due to shutdown signal)
             err_str = "Tool execution cancelled."
             self.logger.warning(f"{tool_name} execution was cancelled.")
             res = {"success": False, "error": err_str, "status_code": 499} # Use 499 Client Closed Request
             record_stats["failure"] += 1 # Count cancellation as failure for stats
             self.state.last_error_details = {"tool": tool_name, "args": arguments, "error": err_str, "type":
             "CancelledError"}
             self.state.last_action_summary = f"{tool_name} -> Cancelled"
             # Re-raise cancellation to potentially stop the loop if needed
             raise

        except Exception as e:
            # Catch any other unexpected errors during execution or retries
            err_str = str(e)
            self.logger.error(f"Unexpected Error executing {tool_name}: {err_str}", exc_info=True) # Log full
            traceback
            res = {"success": False, "error": f"Unexpected error: {err_str}"}
            record_stats["failure"] += 1 # Record failure
            # Store categorized error
            self.state.last_error_details = {"tool": tool_name, "args": arguments, "error": err_str, "type":
            "UnexpectedExecutionError"}
            self.state.last_action_summary = f"{tool_name} -> Failed: Unexpected error."

        # --- Step 11: Record Action Completion (if start was recorded) ---
        if action_id:
            # Pass the final result 'res' (success or failure dict) to the completion recorder
            await self._record_action_completion_internal(action_id, res)

        # --- Step 12: Handle Workflow & Goal Side Effects ---
        # Call this *after* execution and completion recording, using the final 'res'
        await self._handle_workflow_and_goal_side_effects(tool_name, final_arguments, res)

        # Return the final processed result dictionary
        return res
        # <<< End Integration Block: Enhance _execute_tool_call_internal >>>
```

```python
async def _handle_workflow_and_goal_side_effects(self, tool_name: str, arguments: Dict, result_content:
Dict):
    """
    Handles agent state changes triggered by specific tool outcomes,
    including workflow creation/termination and goal stack updates.
    """
    # <<< Start Integration Block: Goal Stack Side Effects >>>
    # --- Side effects for Workflow Creation ---
    if tool_name == TOOL_CREATE_WORKFLOW and result_content.get("success"):
        new_wf_id = result_content.get("workflow_id")
        primary_chain_id = result_content.get("primary_thought_chain_id")
        parent_wf_id = arguments.get("parent_workflow_id") # Parent WF from original args
        wf_title = result_content.get("title", "Untitled Workflow")
        wf_goal_desc = result_content.get("goal", "Achieve objectives")

        if new_wf_id:
            # Set the agent's primary workflow ID and context ID to the new one
            self.state.workflow_id = new_wf_id
            self.state.context_id = new_wf_id # Align context ID

            # Manage the workflow stack
            is_sub_workflow = parent_wf_id and parent_wf_id in self.state.workflow_stack
            if is_sub_workflow:
                self.state.workflow_stack.append(new_wf_id)
                log_prefix = "sub-"
            else: # New root workflow or parent not on stack
                self.state.workflow_stack = [new_wf_id]
                log_prefix = "new "

            # Set the current thought chain ID
            self.state.current_thought_chain_id = primary_chain_id

            # --- Goal Stack Update for New Workflow ---
            # If it's a new root workflow, reset the goal stack and create a root goal
            if not is_sub_workflow:
                self.state.goal_stack = []
                self.state.current_goal_id = None
                # Try to create a root goal in the UMS for this new workflow
                if self._find_tool_server(TOOL_PUSH_SUB_GOAL):
                    try:
                        goal_res = await self._execute_tool_call_internal(
                            TOOL_PUSH_SUB_GOAL,
                            {
                                "workflow_id": new_wf_id,
                                "description": wf_goal_desc,
                                # Parent is None for the root goal
                            },
                            record_action=False
                        )
                        if goal_res.get("success") and goal_res.get("goal_id"):
                            new_goal = {"goal_id": goal_res["goal_id"], "description": wf_goal_desc,
                            "status": "active"}
                            self.state.goal_stack.append(new_goal)
                            self.state.current_goal_id = new_goal["goal_id"]
                            self.logger.info(f"Created root goal {_fmt_id(self.state.current_goal_id)}
                            for {log_prefix}workflow {_fmt_id(new_wf_id)}.")
                        else:
                            self.logger.warning(f"Failed to create root goal via UMS for new workflow
                            {_fmt_id(new_wf_id)}: {goal_res.get('error')}")
                    except Exception as goal_err:
                         self.logger.error(f"Error creating root goal for new workflow
                         {_fmt_id(new_wf_id)}: {goal_err}")
                else:
                    self.logger.warning(f"Cannot create root goal for new workflow: Tool
                    {TOOL_PUSH_SUB_GOAL} unavailable.")
            # If it's a sub-workflow, the goal that triggered its creation should already be on the stack.
            # We don't automatically create a new goal here; the LLM should explicitly push one if needed.

            self.logger.info(f"Switched to {log_prefix}workflow: {_fmt_id(new_wf_id)}. Current chain:
            {_fmt_id(primary_chain_id)}. Current goal: {_fmt_id(self.state.current_goal_id)}",
            emoji_key="label")

            # Reset plan, errors, and replan flag for the new workflow context
            self.state.current_plan = [PlanStep(description=f"Start {log_prefix}workflow: '{wf_title}'.
            Goal: {wf_goal_desc}.")]
            self.state.consecutive_error_count = 0
            self.state.needs_replan = False
            self.state.last_error_details = None

    # --- Side effects for Pushing a Sub-Goal ---
    elif tool_name == TOOL_PUSH_SUB_GOAL and result_content.get("success"):
        new_goal = result_content.get("goal") # Assuming tool returns the created goal object
        if isinstance(new_goal, dict) and new_goal.get("goal_id"):
             # Add the new goal to the agent's state stack
             self.state.goal_stack.append(new_goal)
             # Set the new goal as the current goal
             self.state.current_goal_id = new_goal["goal_id"]
             self.logger.info(f"Pushed new sub-goal {_fmt_id(self.state.current_goal_id)} onto stack:
             '{new_goal.get('description', '')[:50]}...'. Stack depth: {len(self.state.goal_stack)}",
             emoji_key="arrow_down")
             # Force replan to address the new sub-goal
             self.state.needs_replan = True
             self.state.current_plan = [PlanStep(description=f"Start new sub-goal:
             '{new_goal.get('description', '')[:50]}...'")]
             self.state.last_error_details = None # Clear errors when pushing new goal
        else:
             self.logger.warning(f"Tool {TOOL_PUSH_SUB_GOAL} succeeded but did not return valid goal
             data: {result_content}")

    # --- Side effects for Marking Goal Status ---
    elif tool_name == TOOL_MARK_GOAL_STATUS and result_content.get("success"):
        goal_id_marked = arguments.get("goal_id")
        new_status = arguments.get("status")

        # Update the goal status within the agent's state stack
        goal_found_in_stack = False
        for goal in self.state.goal_stack:
            if goal.get("goal_id") == goal_id_marked:
                goal["status"] = new_status
                goal_found_in_stack = True
                self.logger.info(f"Updated goal {_fmt_id(goal_id_marked)} status in state stack to:
                {new_status}.")
                break

        if not goal_found_in_stack:
             self.logger.warning(f"Goal {_fmt_id(goal_id_marked)} (marked {new_status}) not found in
             current agent goal stack: {self.state.goal_stack}")

        # If the goal marked was the *current* goal and it's now completed/failed
        if goal_id_marked == self.state.current_goal_id and new_status in ["completed", "failed"]:
            # Pop the completed/failed goal from the stack
            if self.state.goal_stack and self.state.goal_stack[-1].get("goal_id") == goal_id_marked:
                finished_goal = self.state.goal_stack.pop()
                self.logger.info(f"Popped goal {_fmt_id(finished_goal.get('goal_id'))} (status:
                {new_status}) from stack.")
            else:
                 self.logger.warning(f"Attempted to pop goal {_fmt_id(goal_id_marked)}, but it wasn't at
                 the top of the stack.")

            # Set the new current goal to the one now at the top
            self.state.current_goal_id = self.state.goal_stack[-1].get("goal_id") if
            self.state.goal_stack else None
            self.logger.info(f"Returning focus to goal: {_fmt_id(self.state.current_goal_id) if
            self.state.current_goal_id else 'Overall Goal'}. Stack depth: {len(self.state.goal_stack)}",
            emoji_key="arrow_up")

            # Check if the stack is now empty (meaning the root goal was completed/failed)
            if not self.state.goal_stack:
                 self.logger.info("Goal stack empty. Overall goal presumed finished.")
                 # Set overall goal achieved flag *only if the last goal was completed*
                 self.state.goal_achieved_flag = (new_status == "completed")
                 # Optionally update workflow status if stack is empty
                 if self.state.workflow_id and self._find_tool_server(TOOL_UPDATE_WORKFLOW_STATUS):
                     wf_status = WorkflowStatus.COMPLETED.value if self.state.goal_achieved_flag else
                     WorkflowStatus.FAILED.value
                     await self._execute_tool_call_internal(
                         TOOL_UPDATE_WORKFLOW_STATUS,
                         {
                             "workflow_id": self.state.workflow_id,
                             "status": wf_status,
                             "completion_message": f"Overall goal marked {wf_status} after stack
                             completion."
                         },
                         record_action=False
```

```
                    )
                # Clear plan when overall goal is done
                self.state.current_plan = []
            else:
                # Force replan to address the parent goal after returning from sub-goal
                self.state.needs_replan = True
                self.state.current_plan = [PlanStep(description=f"Returned from sub-goal
                {_fmt_id(goal_id_marked)} (status: {new_status}). Re-assess current goal:
                {_fmt_id(self.state.current_goal_id)}.")]
            self.state.last_error_details = None # Clear error details when goal status changes

        # --- Side effects for Workflow Status Update (Completion/Failure/Abandonment) ---
        elif tool_name == TOOL_UPDATE_WORKFLOW_STATUS and result_content.get("success"):
            status = arguments.get("status") # Status requested in the tool call
            wf_id_updated = arguments.get("workflow_id") # Workflow that was updated

            # Check if the *currently active* workflow (top of stack) was the one updated
            if wf_id_updated and self.state.workflow_stack and wf_id_updated ==
            self.state.workflow_stack[-1]:
                # Check if the new status is a terminal one
                is_terminal = status in [
                    WorkflowStatus.COMPLETED.value,
                    WorkflowStatus.FAILED.value,
                    WorkflowStatus.ABANDONED.value
                ]

                if is_terminal:
                    # Remove the finished workflow from the stack
                    finished_wf = self.state.workflow_stack.pop()
                    parent_wf_id = self.state.workflow_stack[-1] if self.state.workflow_stack else None

                    # --- Goal Stack Update for Completed Sub-Workflow ---
                    # Find the goal on the *parent's* stack that likely corresponds to this finished
                    sub-workflow.
                    # This is heuristic - assumes the goal *currently active* when the sub-workflow finished
                    is the relevant one.
                    # A more robust system might store the initiating goal ID when the sub-workflow is
                    created.
                    corresponding_goal_id = self.state.current_goal_id # Goal active *before* popping WF
                    stack
                    if corresponding_goal_id and parent_wf_id: # Ensure we have a goal and a parent WF to
                    mark it in
                        # Infer goal status from workflow status
                        goal_status_to_set = "completed" if status == WorkflowStatus.COMPLETED.value else
                        "failed"
                        self.logger.info(f"Attempting to mark goal {_fmt_id(corresponding_goal_id)} as
                        {goal_status_to_set} due to sub-workflow {_fmt_id(finished_wf)} completion.")
                        if self._find_tool_server(TOOL_MARK_GOAL_STATUS):
                             mark_res = await self._execute_tool_call_internal(
                                 TOOL_MARK_GOAL_STATUS,
                                 {
                                     "goal_id": corresponding_goal_id,
                                     "status": goal_status_to_set,
                                     "reason": f"Sub-workflow {_fmt_id(finished_wf)} finished with status:
                                     {status}"
                                 },
                                 record_action=False
                             )
                             # This call will trigger the goal stack popping logic above if successful
                             if not mark_res.get("success"):
                                 self.logger.warning(f"Failed to automatically mark goal
                                 {_fmt_id(corresponding_goal_id)} after sub-workflow completion:
                                 {mark_res.get('error')}")
                        else:
                             self.logger.warning(f"Cannot mark goal status after sub-workflow completion:
                             Tool {TOOL_MARK_GOAL_STATUS} unavailable.")
                    elif corresponding_goal_id and not parent_wf_id:
                         self.logger.info(f"Root workflow {_fmt_id(finished_wf)} finished. Corresponding goal
                         {_fmt_id(corresponding_goal_id)} likely represents overall completion.")
                         # The MARK_GOAL_STATUS logic above should handle the root goal completion and set
                         goal_achieved_flag
                    else:
                         self.logger.warning(f"Sub-workflow {_fmt_id(finished_wf)} finished, but couldn't
                         identify corresponding goal to mark.")


                    # --- Update Agent State Based on Workflow Stack ---
                    if parent_wf_id:
```

```
                    # If there's a parent workflow remaining on the stack, return to it
                    self.state.workflow_id = parent_wf_id
                    self.state.context_id = self.state.workflow_id # Realign context ID
                    # Set thought chain (should align with parent)
                    await self._set_default_thought_chain_id()
                    # Note: Current goal ID should have been updated by the MARK_GOAL_STATUS call above
                    self.logger.info(f"Sub-workflow {_fmt_id(finished_wf)} finished ({status}). Returning
                    to parent {_fmt_id(self.state.workflow_id)}. Current chain:
                    {_fmt_id(self.state.current_thought_chain_id)}. Current Goal:
                    {_fmt_id(self.state.current_goal_id)}", emoji_key="arrow_left")
                    # Force replan in the parent context
                    self.state.needs_replan = True
                    self.state.current_plan = [PlanStep(description=f"Returned from sub-workflow
                    {_fmt_id(finished_wf)} (status: {status}). Re-assess current goal:
                    {_fmt_id(self.state.current_goal_id)}.")]
                    self.state.last_error_details = None # Clear error details from the sub-task
                else:
                    # If the stack is empty, the root workflow finished
                    self.logger.info(f"Root workflow {_fmt_id(finished_wf)} finished with status:
                    {status}.")
                    # Clear active workflow state
                    self.state.workflow_id = None
                    self.state.context_id = None
                    self.state.current_thought_chain_id = None
                    # The goal_achieved_flag should have been set by the MARK_GOAL_STATUS logic when the
                    stack became empty
                    # Clear the plan as the workflow is over
                    self.state.current_plan = []
        # <<< End Integration Block: Goal Stack Side Effects >>>


    async def _apply_heuristic_plan_update(self, last_decision: Dict[str, Any], last_tool_result_content:
    Optional[Dict[str, Any]] = None):
        """
        Applies heuristic updates to the plan based on the last action's outcome
        when the LLM doesn't explicitly call `agent:update_plan`.

        This acts as a default progression mechanism. It marks steps completed
        on success, handles failures by marking the step failed and inserting
        an analysis step, and updates meta-cognitive counters.
        """
        # <<< Start Integration Block: Heuristic Plan Update Method (Phase 1, Step 3) >>>
        self.logger.info("Applying heuristic plan update (fallback)...", emoji_key="clipboard")

        # Handle case where plan might be empty (shouldn't usually happen)
        if not self.state.current_plan:
            self.logger.warning("Plan is empty during heuristic update, adding default re-evaluation step.")
            self.state.current_plan = [PlanStep(description="Fallback: Re-evaluate situation.")]
            self.state.needs_replan = True # Force replan if plan was empty
            return

        # Get the step the agent was working on (assumed to be the first)
        current_step = self.state.current_plan[0]
        decision_type = last_decision.get("decision") # What the LLM decided to do

        action_successful = False # Flag to track if the action succeeded for counter updates
        tool_name_executed = last_decision.get("tool_name") # Tool name if a tool was called

        # --- Update plan based on decision type and success ---
        # Case 1: LLM called a tool (and it wasn't AGENT_TOOL_UPDATE_PLAN)
        if decision_type == "call_tool" and tool_name_executed != AGENT_TOOL_UPDATE_PLAN:
            # Check the success status from the tool execution result
            tool_success = isinstance(last_tool_result_content, dict) and
            last_tool_result_content.get("success", False)
            action_successful = tool_success

            if tool_success:
                # On success, mark step completed and remove from plan
                current_step.status = ActionStatus.COMPLETED.value # Use enum value
                # Generate a concise summary for the plan step
                summary = "Success."
                if isinstance(last_tool_result_content, dict):
                     # Prioritize specific meaningful keys from the result
                     summary_keys = ["summary", "message", "memory_id", "action_id", "artifact_id",
                     "link_id", "chain_id", "state_id", "report", "visualization", "goal_id"] # Added goal_id
                     data_payload = last_tool_result_content.get("data", last_tool_result_content) # Look in
                     'data' or root
                     if isinstance(data_payload, dict):
```

```
                for k in summary_keys:
                    if k in data_payload and data_payload[k]:
                        # Format IDs or use string representation
                        summary = f"{k}: {_fmt_id(data_payload[k]) if 'id' in k else
                        str(data_payload[k])}"
                        break
                else: # Fallback preview for dicts
                    data_str = str(data_payload)[:70]
                    summary = f"Success. Data: {data_str}..." if len(str(data_payload)) > 70 else
                    f"Success. Data: {data_str}"
            else: # Handle non-dict data payload
                data_str = str(data_payload)[:70]
                summary = f"Success. Data: {data_str}..." if len(str(data_payload)) > 70 else
                f"Success. Data: {data_str}"

            current_step.result_summary = summary[:150] # Add summary to step, truncated
            self.state.current_plan.pop(0) # Remove completed step from the front
            # If the plan is now empty, add a final analysis step
            if not self.state.current_plan:
                self.logger.info("Plan completed. Adding final analysis step.")
                self.state.current_plan.append(PlanStep(description="Plan finished. Analyze overall
                result and decide if goal is met."))
            self.state.needs_replan = False # Success usually doesn't require immediate replan
        else: # Tool failed
            current_step.status = ActionStatus.FAILED.value # Mark step as failed
            # Extract error message for summary
            error_msg = "Unknown failure"
            if isinstance(last_tool_result_content, dict):
                # Use the enhanced error details if available
                error_details = self.state.last_error_details
                if error_details:
                    error_msg = f"Type: {error_details.get('type', 'Unknown')}, Msg:
                    {error_details.get('error', 'Unknown')}"
                else: # Fallback to basic error message
                    error_msg = str(last_tool_result_content.get('error', 'Unknown failure'))

            current_step.result_summary = f"Failure: {error_msg[:150]}" # Add error summary
            # Keep the failed step in the plan for context.
            # Insert an analysis step *after* the failed step, if one isn't already there.
            if len(self.state.current_plan) < 2 or not
            self.state.current_plan[1].description.startswith("Analyze failure of step"):
                self.state.current_plan.insert(1, PlanStep(description=f"Analyze failure of step
                '{current_step.description[:30]}...' and replan."))
            self.state.needs_replan = True # Failure always requires replanning

    # Case 2: LLM decided to record a thought
    elif decision_type == "thought_process":
        action_successful = True # Recording a thought is considered a successful step completion
        heuristically
        current_step.status = ActionStatus.COMPLETED.value
        current_step.result_summary = f"Thought Recorded: {last_decision.get('content','')[:50]}..." #
        Summary based on thought
        self.state.current_plan.pop(0) # Remove completed step
        # If plan is empty after thought, add next step prompt
        if not self.state.current_plan:
            self.logger.info("Plan completed after thought. Adding next action step.")
            self.state.current_plan.append(PlanStep(description="Decide next action based on recorded
            thought and overall goal."))
        self.state.needs_replan = False # Recording a thought doesn't force replan

    # Case 3: LLM signaled completion
    elif decision_type == "complete":
        action_successful = True # Achieving goal is success
        # Overwrite plan with a final step (status handled in main loop)
        self.state.current_plan = [PlanStep(description="Goal Achieved. Finalizing.",
        status="completed")]
        self.state.needs_replan = False

    # Case 4: Handle errors or unexpected decisions (including AGENT_TOOL_UPDATE_PLAN failure)
    else:
        action_successful = False # Mark as failure for counter updates
        # Only mark the *current plan step* as failed if it wasn't the plan update tool itself that caused
        the error state
        if tool_name_executed != AGENT_TOOL_UPDATE_PLAN:
            current_step.status = ActionStatus.FAILED.value
            # Use the last action summary (which should contain the error) for the result summary
            err_summary = self.state.last_action_summary or "Unknown agent error"
            current_step.result_summary = f"Agent/Tool Error: {err_summary[:100]}..."
            # Insert re-evaluation step if not already present
            if len(self.state.current_plan) < 2 or not
            self.state.current_plan[1].description.startswith("Re-evaluate due to agent error"):
                 self.state.current_plan.insert(1, PlanStep(description="Re-evaluate due to agent error
                 or unclear decision."))
        # Always set needs_replan if an error occurred or the decision was unexpected
        self.state.needs_replan = True

    # --- Update Meta-Cognitive Counters ---
    if action_successful:
        # Reset error counter on any successful progression
        self.state.consecutive_error_count = 0

        # Increment success counters *only* if the action wasn't internal/meta
        # Check if a tool was executed and if it's not in the excluded set
        if tool_name_executed and tool_name_executed not in self._INTERNAL_OR_META_TOOLS:
             # Use float increments for flexibility
             self.state.successful_actions_since_reflection += 1.0
             self.state.successful_actions_since_consolidation += 1.0
             self.logger.debug(f"Incremented success counters
             R:{self.state.successful_actions_since_reflection:.1f},
             C:{self.state.successful_actions_since_consolidation:.1f} after successful action:
             {tool_name_executed}")
        elif decision_type == "thought_process":
             # Option: Count thoughts as partial progress (e.g., 0.5)
             self.state.successful_actions_since_reflection += 0.5 # Example: count thought as half an
             action
             self.state.successful_actions_since_consolidation += 0.5
             self.logger.debug(f"Incremented success counters
             R:{self.state.successful_actions_since_reflection:.1f},
             C:{self.state.successful_actions_since_consolidation:.1f} after thought recorded.")

    else: # Action failed or was an error condition handled above
        # Increment consecutive error count
        self.state.consecutive_error_count += 1
        self.logger.warning(f"Consecutive error count increased to:
        {self.state.consecutive_error_count}")
        # Reset reflection counter immediately on error to encourage faster reflection
        if self.state.successful_actions_since_reflection > 0:
             self.logger.info(f"Resetting reflection counter due to error (was
             {self.state.successful_actions_since_reflection:.1f}).")
             self.state.successful_actions_since_reflection = 0
             # Policy Decision: Keep consolidation counter running unless error rate is very high (handled
             in _adapt_thresholds)

    # --- Log Final Plan State ---
    log_plan_msg = f"Plan updated heuristically. Steps remaining: {len(self.state.current_plan)}. "
    if self.state.current_plan:
        next_step = self.state.current_plan[0]
        depends_str = f"Depends: {[_fmt_id(d) for d in next_step.depends_on]}" if next_step.depends_on
        else "Depends: None"
        log_plan_msg += f"Next: '{next_step.description[:60]}...' (Status: {next_step.status},
        {depends_str})"
    else:
        log_plan_msg += "Plan is now empty."
    self.logger.info(log_plan_msg, emoji_key="clipboard")
    # <<< End Integration Block: Heuristic Plan Update Method >>>


# ------------------------------------------------ adaptive thresholds --
def _adapt_thresholds(self, stats: Dict[str, Any]) -> None:
    """
    Adjusts reflection and consolidation thresholds based on memory statistics,
    tool usage patterns, and **progress stability** to dynamically control
    meta-cognition frequency. Includes "Mental Momentum" bias.
    """
    # <<< Start Integration Block: Enhance _adapt_thresholds (Goal Stack + Momentum) >>>
    # Validate stats input
    if not stats or not stats.get("success"):
         self.logger.warning("Cannot adapt thresholds: Invalid or failed stats received.")
         return

    self.logger.debug(f"Adapting thresholds based on stats: {stats}")
    # Use dampening factor from constant
    adjustment_dampening = THRESHOLD_ADAPTATION_DAMPENING
    changed = False # Flag to log if any threshold changed

    # --- Consolidation Threshold Adaptation ---
```

```
        episodic_count = stats.get("by_level", {}).get(MemoryLevel.EPISODIC.value, 0)
        total_memories = stats.get("total_memories", 1) # Avoid division by zero
        episodic_ratio = episodic_count / total_memories if total_memories > 0 else 0
        # Target range for episodic ratio (e.g., ideally keep it below 30%)
        target_episodic_ratio_upper = 0.30
        target_episodic_ratio_lower = 0.10
        # Calculate deviation from the middle of the target range
        mid_target_ratio = (target_episodic_ratio_upper + target_episodic_ratio_lower) / 2
        ratio_deviation = episodic_ratio - mid_target_ratio
        # Calculate adjustment: more negative deviation (low ratio) -> increase threshold
        # More positive deviation (high ratio) -> decrease threshold
        # Scale adjustment based on current threshold (larger thresholds allow bigger steps)
        consolidation_adjustment = -math.ceil(ratio_deviation * self.state.current_consolidation_threshold *
        2.0) # Factor of 2 controls sensitivity

        # Apply dampening to the adjustment
        dampened_adjustment = int(consolidation_adjustment * adjustment_dampening)
        if dampened_adjustment != 0:
            old_threshold = self.state.current_consolidation_threshold
            # Calculate potential new threshold, enforcing MIN/MAX bounds
            potential_new = max(MIN_CONSOLIDATION_THRESHOLD, min(MAX_CONSOLIDATION_THRESHOLD, old_threshold +
            dampened_adjustment))
            # Apply change only if it's different from the current threshold
            if potential_new != old_threshold:
                change_direction = "Lowering" if dampened_adjustment < 0 else "Raising"
                self.logger.info(
                    f"{change_direction} consolidation threshold: {old_threshold} -> {potential_new} "
                    f"(Episodic Ratio: {episodic_ratio:.1%}, Deviation: {ratio_deviation:+.1%}, Adjustment:
                    {dampened_adjustment})"
                )
                self.state.current_consolidation_threshold = potential_new
                changed = True

        # --- Reflection Threshold Adaptation ---
        # Use tool usage stats accumulated in agent state
        total_calls = sum(v.get("success", 0) + v.get("failure", 0) for v in
        self.state.tool_usage_stats.values())
        total_failures = sum(v.get("failure", 0) for v in self.state.tool_usage_stats.values())
        # Calculate failure rate, avoid division by zero, require minimum calls for statistical significance
        min_calls_for_rate = 5 # Need at least 5 calls to calculate a meaningful rate
        failure_rate = (total_failures / total_calls) if total_calls >= min_calls_for_rate else 0.0
        # Target failure rate (e.g., aim for below 10%)
        target_failure_rate = 0.10
        failure_deviation = failure_rate - target_failure_rate
        # Calculate adjustment: more positive deviation (high failure) -> decrease threshold
        # More negative deviation (low failure) -> increase threshold
        reflection_adjustment = -math.ceil(failure_deviation * self.state.current_reflection_threshold * 3.0)
        # Factor of 3 controls sensitivity

        # --- Mental Momentum Bias ---
        is_stable_progress = (failure_rate < target_failure_rate * 0.5) and
        (self.state.consecutive_error_count == 0)
        if is_stable_progress and reflection_adjustment >= 0: # Only apply bias if adjustment is non-negative
        (i.e., increasing or zero)
             momentum_bias = math.ceil(reflection_adjustment * (MOMENTUM_THRESHOLD_BIAS_FACTOR - 1.0)) #
             Calculate the *additional* bias
             reflection_adjustment += momentum_bias # Add the positive bias
             self.logger.debug(f"Applying Mental Momentum: Adding +{momentum_bias} bias to reflection
             threshold adjustment (Stable progress).")

        # Apply dampening to the (potentially biased) adjustment
        dampened_adjustment = int(reflection_adjustment * adjustment_dampening)
        if dampened_adjustment != 0 and total_calls >= min_calls_for_rate: # Only adjust if enough calls
            old_threshold = self.state.current_reflection_threshold
            # Calculate potential new threshold, enforcing MIN/MAX bounds
            potential_new = max(MIN_REFLECTION_THRESHOLD, min(MAX_REFLECTION_THRESHOLD, old_threshold +
            dampened_adjustment))
            # Apply change only if different
            if potential_new != old_threshold:
                change_direction = "Lowering" if dampened_adjustment < 0 else "Raising"
                momentum_tag = " (+Momentum)" if is_stable_progress and momentum_bias > 0 else ""
                self.logger.info(
                    f"{change_direction} reflection threshold: {old_threshold} -> {potential_new} "
                    f"(Failure Rate: {failure_rate:.1%}, Deviation: {failure_deviation:+.1%}, Adjustment:
                    {dampened_adjustment}{momentum_tag})"
                )
                self.state.current_reflection_threshold = potential_new
                changed = True
        # Log if no changes were made
        if not changed:
             self.logger.debug("No threshold adjustments triggered based on current stats/heuristics.")
        # <<< End Integration Block: Enhance _adapt_thresholds >>>


    # ----------------------------------------------- periodic task runner --
    async def _run_periodic_tasks(self):
        """
        Runs scheduled cognitive maintenance and enhancement tasks periodically.

        Checks intervals and thresholds to trigger tasks like reflection,
        consolidation, working memory optimization, focus updates, promotion checks,
        statistics computation, threshold adaptation, and memory maintenance.
        Executes triggered tasks sequentially within the loop cycle.
        """
        # Prevent running if no workflow or shutting down
        if not self.state.workflow_id or not self.state.context_id or self._shutdown_event.is_set():
            return

        # List to hold tasks scheduled for this cycle: (tool_name, args_dict)
        tasks_to_run: List[Tuple[str, Dict]] = []
        # List to track reasons for triggering tasks (for logging)
        trigger_reasons: List[str] = []

        # Check tool availability once per cycle for efficiency
        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 1: Highest Priority - Stats Check & Adaptation ---
        # Increment counter for stats adaptation interval
        self.state.loops_since_stats_adaptation += 1
        # Check if interval reached
        if self.state.loops_since_stats_adaptation >= STATS_ADAPTATION_INTERVAL:
            if stats_tool_available:
                trigger_reasons.append("StatsInterval")
                try:
                    # Fetch current statistics for the workflow
                    stats = await self._execute_tool_call_internal(
                        TOOL_COMPUTE_STATS, {"workflow_id": self.state.workflow_id}, record_action=False
                    )
                    if stats.get("success"):
                        # Adapt thresholds based on the fetched stats
                        self._adapt_thresholds(stats)
                        # Example: Potentially trigger consolidation *now* if stats show high episodic count
                        episodic_count = stats.get("by_level", {}).get(MemoryLevel.EPISODIC.value, 0)
                        # Trigger if count significantly exceeds the *current dynamic* threshold
                        if episodic_count > (self.state.current_consolidation_threshold * 2.0) and
                        consolidation_tool_available:
                            # Check if consolidation isn't already scheduled for other reasons this cycle
                            if not any(task[0] == TOOL_CONSOLIDATION for task in tasks_to_run):
                                self.logger.info(f"High episodic count ({episodic_count}) detected via stats,
                                scheduling consolidation.")
                                # Schedule consolidation task
                                tasks_to_run.append((TOOL_CONSOLIDATION, {
                                    "workflow_id": self.state.workflow_id,
                                    "consolidation_type": "summary", # Default to summary
                                    # Filter consolidation sources to episodic memories
                                    "query_filter": {"memory_level": MemoryLevel.EPISODIC.value},
                                    "max_source_memories": self.consolidation_max_sources
                                }))
                                trigger_reasons.append(f"HighEpisodic({episodic_count})")
                                # Reset consolidation counter as we're triggering it now based on stats
                                self.state.successful_actions_since_consolidation = 0
                    else:
                        # Log if stats computation failed
                        self.logger.warning(f"Failed to compute stats for adaptation: {stats.get('error')}")
                except Exception as e:
                    # Log errors during the stats/adaptation process
                    self.logger.error(f"Error during stats computation/adaptation: {e}", exc_info=False)
                finally:
                     # Reset the interval counter regardless of success/failure
                     self.state.loops_since_stats_adaptation = 0
            else:
                # Log if stats tool is unavailable
                self.logger.warning(f"Skipping stats/adaptation: Tool {TOOL_COMPUTE_STATS} not available")

        # --- Tier 2: Reflection & Consolidation (Based on dynamic thresholds) ---
        # Reflection Trigger (Check replan flag OR success counter vs. *dynamic* threshold)
        needs_reflection = self.state.needs_replan or self.state.successful_actions_since_reflection >=
        self.state.current_reflection_threshold
        if needs_reflection:
            if reflection_tool_available:
                # Check if reflection isn't already scheduled this cycle
                if not any(task[0] == TOOL_REFLECTION for task in tasks_to_run):
                    # Cycle through different reflection types for variety
                    reflection_type = self.reflection_type_sequence[self.state.reflection_cycle_index %
                    len(self.reflection_type_sequence)]
                    # Schedule reflection task
                    tasks_to_run.append((TOOL_REFLECTION, {"workflow_id": self.state.workflow_id,
                    "reflection_type": reflection_type}))
                    # Log the specific reason for triggering
                    reason_str = f"ReplanNeeded({self.state.needs_replan})" if self.state.needs_replan else
                    f"SuccessCount({self.state.successful_actions_since_reflection:.1f}>={self.state.current
                    _reflection_threshold})"
                    trigger_reasons.append(f"Reflect({reason_str})")
                    # Reset the success counter and advance the cycle index
                    self.state.successful_actions_since_reflection = 0
                    self.state.reflection_cycle_index += 1
            else:
                # Log if tool unavailable, still reset counter to prevent immediate re-trigger
                self.logger.warning(f"Skipping reflection: Tool {TOOL_REFLECTION} not available")
                self.state.successful_actions_since_reflection = 0

        # Consolidation Trigger (Check success counter vs. *dynamic* threshold)
        needs_consolidation = self.state.successful_actions_since_consolidation >=
        self.state.current_consolidation_threshold
        if needs_consolidation:
            if consolidation_tool_available:
                # Check if consolidation isn't already scheduled this cycle
                if not any(task[0] == TOOL_CONSOLIDATION for task in tasks_to_run):
                    # Schedule consolidation task (e.g., summarize episodic memories)
                    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_reasons.append(f"ConsolidateThreshold({self.state.successful_actions_since_conso
                    lidation:.1f}>={self.state.current_consolidation_threshold})")
                    # Reset the success counter
                    self.state.successful_actions_since_consolidation = 0
            else:
                # Log if tool unavailable, still reset counter
                self.logger.warning(f"Skipping consolidation: Tool {TOOL_CONSOLIDATION} not available")
                self.state.successful_actions_since_consolidation = 0

        # --- Tier 3: Optimization & Focus (Based on loop interval) ---
        # Increment optimization interval counter
        self.state.loops_since_optimization += 1
        # Check if interval reached
        if self.state.loops_since_optimization >= OPTIMIZATION_LOOP_INTERVAL:
            # Schedule working memory optimization if tool available
            if optimize_wm_tool_available:
                tasks_to_run.append((TOOL_OPTIMIZE_WM, {"context_id": self.state.context_id}))
                trigger_reasons.append("OptimizeInterval")
            else:
                self.logger.warning(f"Skipping optimization: Tool {TOOL_OPTIMIZE_WM} not available")

            # Schedule automatic focus update if tool available
            if auto_focus_tool_available:
                tasks_to_run.append((TOOL_AUTO_FOCUS, {"context_id": self.state.context_id}))
                trigger_reasons.append("FocusUpdate")
            else:
                self.logger.warning(f"Skipping auto-focus: Tool {TOOL_AUTO_FOCUS} not available")

            # Reset the interval counter
            self.state.loops_since_optimization = 0
```

```
        # --- Tier 4: Promotion Check (Based on loop interval) ---
        # Increment promotion check interval counter
        self.state.loops_since_promotion_check += 1
        # Check if interval reached
        if self.state.loops_since_promotion_check >= MEMORY_PROMOTION_LOOP_INTERVAL:
            if promote_mem_tool_available:
                # Schedule the internal check function, not the tool directly
                tasks_to_run.append(("CHECK_PROMOTIONS", {})) # Special marker task name
                trigger_reasons.append("PromotionInterval")
            else:
                self.logger.warning(f"Skipping promotion check: Tool {TOOL_PROMOTE_MEM} unavailable.")
            # Reset the interval counter
            self.state.loops_since_promotion_check = 0

        # --- Tier 5: Lowest Priority - Maintenance ---
        # Increment maintenance interval counter
        self.state.loops_since_maintenance += 1
        # Check if interval reached
        if self.state.loops_since_maintenance >= MAINTENANCE_INTERVAL:
            if maintenance_tool_available:
                # Schedule memory expiration task
                tasks_to_run.append((TOOL_DELETE_EXPIRED_MEMORIES, {})) # No args needed usually
                trigger_reasons.append("MaintenanceInterval")
                self.state.loops_since_maintenance = 0 # Reset interval counter
            else:
                # Log if tool unavailable
                self.logger.warning(f"Skipping maintenance: Tool {TOOL_DELETE_EXPIRED_MEMORIES} not
                available")


        # --- Execute Scheduled Tasks ---
        if tasks_to_run:
            unique_reasons_str = ', '.join(sorted(set(trigger_reasons))) # Log unique reasons
            self.logger.info(f"Running {len(tasks_to_run)} periodic tasks (Triggers:
            {unique_reasons_str})...", emoji_key="brain")

            # Optional: Prioritize tasks (e.g., run maintenance first)
            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)

            # Execute tasks sequentially in this loop cycle
            for tool_name, args in tasks_to_run:
                # Check shutdown flag before each task execution
                if self._shutdown_event.is_set():
                    self.logger.info("Shutdown detected during periodic tasks, aborting remaining.")
                    break
                try:
                    # Handle the special internal promotion check task
                    if tool_name == "CHECK_PROMOTIONS":
                        await self._trigger_promotion_checks() # Call the helper method
                        continue # Move to next scheduled task

                    # Execute standard UMS tool calls
                    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 # Don't record periodic tasks as agent actions
                    )

                    # --- Meta-Cognition Feedback Loop ---
                    # Check if the task was reflection or consolidation and if it succeeded
                    if tool_name in [TOOL_REFLECTION, TOOL_CONSOLIDATION] and result_content.get('success'):
                        feedback = ""
                        # Extract the relevant content from the result dictionary
                        if tool_name == TOOL_REFLECTION:
                            feedback = result_content.get("content", "")
                        elif tool_name == TOOL_CONSOLIDATION:
                             feedback = result_content.get("consolidated_content", "")

                        # Handle cases where result might be nested under 'data'
                        if not feedback and isinstance(result_content.get("data"), dict):
                            if tool_name == TOOL_REFLECTION:
                                feedback = result_content["data"].get("content", "")
                            elif tool_name == TOOL_CONSOLIDATION:
                                feedback = result_content["data"].get("consolidated_content", "")

                        # If feedback content exists, store a summary for the next main loop iteration
                        if feedback:
                            # Create a concise summary (e.g., first line)
                            feedback_summary = str(feedback).split('\n', 1)[0][:150]
                            self.state.last_meta_feedback = f"Feedback from {tool_name.split(':')[-1]}:
                            {feedback_summary}..."
                            self.logger.info(f"Received meta-feedback: {self.state.last_meta_feedback}")
                            # Force replan after receiving significant feedback
                            self.state.needs_replan = True
                        else:
                            # Log if the task succeeded but returned no content for feedback
                            self.logger.debug(f"Periodic task {tool_name} succeeded but provided no feedback
                            content.")

                except Exception as e:
                    # Log errors from periodic tasks but allow the agent loop to continue
                    self.logger.warning(f"Periodic task {tool_name} failed: {e}", exc_info=False)

                # Optional small delay between periodic tasks within a cycle
                await asyncio.sleep(0.1)


    async def _trigger_promotion_checks(self):
        """
        Queries for recently accessed memories eligible for promotion
        (Episodic -> Semantic, Semantic -> Procedural) and schedules
        background checks for each candidate using `_check_and_trigger_promotion`.
        """
        # Ensure a workflow is active
        if not self.state.workflow_id:
             self.logger.debug("Skipping promotion check: No active workflow.")
             return

        self.logger.debug("Running periodic promotion check for recent memories...")
        query_tool_name = TOOL_QUERY_MEMORIES # Tool for searching memories
        # Check tool availability
        if not self._find_tool_server(query_tool_name):
            self.logger.warning(f"Skipping promotion check: Tool {query_tool_name} unavailable.")
            return

        candidate_memory_ids = set() # Use set to store unique IDs
        try:
            # 1. Find recent Episodic memories (potential for Semantic promotion)
            episodic_args = {
                "workflow_id": self.state.workflow_id,
                "memory_level": MemoryLevel.EPISODIC.value,
                "sort_by": "last_accessed", # Prioritize recently used
                "sort_order": "DESC",
                "limit": 5, # Check top N recent/relevant
                "include_content": False # Don't need content for this check
            }
            episodic_results = await self._execute_tool_call_internal(query_tool_name, episodic_args,
            record_action=False)
            if episodic_results.get("success"):
                mems = episodic_results.get("memories", [])
                if isinstance(mems, list):
                    # Add valid memory IDs to the candidate set
                    candidate_memory_ids.update(m.get('memory_id') for m in mems if isinstance(m, dict) and
                    m.get('memory_id'))

            # 2. Find recent Semantic memories of PROCEDURE or SKILL type (potential for Procedural promotion)
            semantic_args = {
                "workflow_id": self.state.workflow_id,
                "memory_level": MemoryLevel.SEMANTIC.value,
                # No type filter here yet, filter after retrieval
                "sort_by": "last_accessed",
                "sort_order": "DESC",
                "limit": 5,
                "include_content": False
            }
            semantic_results = await self._execute_tool_call_internal(query_tool_name, semantic_args,
            record_action=False)
            if semantic_results.get("success"):
                mems = semantic_results.get("memories", [])
                if isinstance(mems, list):
                     # Filter for specific types eligible for promotion to Procedural
                     candidate_memory_ids.update(
                          m.get('memory_id') for m in mems
                          if isinstance(m, dict) and m.get('memory_id') and
                          m.get('memory_type') in [MemoryType.PROCEDURE.value, MemoryType.SKILL.value] #
                          Check type
                    )

            # 3. Schedule background checks for each candidate
            if candidate_memory_ids:
                self.logger.debug(f"Checking {len(candidate_memory_ids)} memories for potential promotion:
                {[_fmt_id(item_id) for item_id in candidate_memory_ids]}")
                # Create background tasks to check each memory individually
                promo_tasks = []
                for mem_id in candidate_memory_ids:
                     # Use _start_background_task to correctly snapshot state and manage task
                     task = self._start_background_task(AgentMasterLoop._check_and_trigger_promotion,
                     memory_id=mem_id)
                     promo_tasks.append(task)
                # Option: Wait for these checks if promotion status is needed immediately,
                # otherwise let them run truly in the background. For now, fire-and-forget.
                # await asyncio.gather(*promo_tasks, return_exceptions=True) # Uncomment to wait
            else:
                # Log if no eligible candidates were found
                self.logger.debug("No recently accessed, eligible memories found for promotion check.")
        except Exception as e:
            # Log errors occurring during the query phase
            self.logger.error(f"Error during periodic promotion check query: {e}", exc_info=False)

    # ================================================================ context gather --
    async def _gather_context(self) -> Dict[str, Any]:
        """
        Gathers comprehensive context for the agent LLM.

        Includes:
        - Core context (recent actions, important memories, key thoughts) via TOOL_GET_CONTEXT.
        - **Current goal stack context (NEW)**.
        - Current working memory via TOOL_GET_WORKING_MEMORY **(Prioritized)**.
        - Proactively searched memories relevant to the current plan step **(Limited Fetch)**.
        - Relevant procedural memories **(Limited Fetch)**.
        - Summary of links around a focal or important memory **(Limited Fetch)**.
        - **Freshness indicators** for components.
        - Handles potential errors during retrieval.
        - Initiates context compression if token estimates exceed thresholds.
        """
        # <<< Start Integration Block: Enhance _gather_context (Goal Stack Context) >>>
        self.logger.info("Gathering comprehensive context...", emoji_key="satellite")
        start_time = time.time()
        retrieval_timestamp = datetime.now(timezone.utc).isoformat() # Timestamp for freshness

        # Initialize context dictionary with placeholders and essential state
        base_context = {
            # Core agent state info
            "current_loop": self.state.current_loop,
            "workflow_id": self.state.workflow_id, # Include current WF ID
            "context_id": self.state.context_id, # Include current Context ID
            "current_plan": [p.model_dump(exclude_none=True) for p in self.state.current_plan], # Current
            plan state
            "last_action_summary": self.state.last_action_summary,
            "consecutive_error_count": self.state.consecutive_error_count,
            "last_error_details": copy.deepcopy(self.state.last_error_details), # Deep copy error details
            "needs_replan": self.state.needs_replan, # Include replan flag in context
            "workflow_stack": self.state.workflow_stack,
            "meta_feedback": self.state.last_meta_feedback, # Include feedback from last meta task
            "current_thought_chain_id": self.state.current_thought_chain_id,
            # Placeholders for dynamically fetched context components
            "current_goal_context": None, # (NEW) Dict: {"retrieved_at": ..., "current_goal": {...},
            "goal_stack_summary": [...]}
            "core_context": None, # From TOOL_GET_CONTEXT
            "current_working_memory": None, # From TOOL_GET_WORKING_MEMORY (will be dict)
            "proactive_memories": None, # Dict: {"retrieved_at": ..., "memories": [...]}
            "relevant_procedures": None, # Dict: {"retrieved_at": ..., "memories": [...]}
            "contextual_links": None, # Dict: {"retrieved_at": ..., "summary": {...}}
            "compression_summary": None, # If compression is applied
            "status": "Gathering...", # Initial status
            "errors": [] # List to collect errors during gathering
        }
        # Clear feedback after adding it to context
        self.state.last_meta_feedback = None

        # Determine the current workflow and context IDs from state
        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 no workflow is active, return immediately
        if not current_workflow_id:
            base_context["status"] = "No Active Workflow"
            base_context["message"] = "Agent must create or load a workflow."
            self.logger.warning(base_context["message"])
            return base_context

        # --- Fetch Goal Stack Context (NEW) ---
        goal_context_data = {"retrieved_at": retrieval_timestamp, "current_goal": None, "goal_stack_summary":
        []}
        if self.state.goal_stack:
             # Provide summary of the stack (limited depth)
             goal_context_data["goal_stack_summary"] = self.state.goal_stack[-CONTEXT_GOAL_STACK_SHOW_LIMIT:]
             # Fetch details for the current goal (top of stack) if tool exists
             current_goal_id = self.state.current_goal_id
             if current_goal_id and self._find_tool_server(TOOL_GET_GOAL_DETAILS):
                 try:
                     goal_details_res = await self._execute_tool_call_internal(
                         TOOL_GET_GOAL_DETAILS, {"goal_id": current_goal_id}, record_action=False
                     )
                     if goal_details_res.get("success") and isinstance(goal_details_res.get("goal"), dict):
                         goal_context_data["current_goal"] = goal_details_res["goal"] # Store full details of
                         current goal
                     else:
                          err_msg = f"Failed to get details for current goal {_fmt_id(current_goal_id)}:
                          {goal_details_res.get('error')}"
                          goal_context_data["current_goal"] = {"goal_id": current_goal_id, "error": err_msg}
                          # Store error marker
                          base_context["errors"].append(err_msg)
                          self.logger.warning(err_msg)
                 except Exception as e:
                      err_msg = f"Exception getting goal details for {_fmt_id(current_goal_id)}: {e}"
                      goal_context_data["current_goal"] = {"goal_id": current_goal_id, "error": err_msg}
                      base_context["errors"].append(err_msg)
                      self.logger.error(err_msg, exc_info=False)
             elif current_goal_id: # If tool missing, use basic info from state
                goal_context_data["current_goal"] = next # If tool missing, use basic info from state stack
                if available
                current_goal_from_stack = next((g for g in self.state.goal_stack if g.get('goal_id') ==
                current_goal_id), None)
                if current_goal_from_stack:
                    goal_context_data["current_goal"] = current_goal_from_stack # Use basic info from state
                    self.logger.debug(f"Using basic goal info from state stack for current goal
                    {_fmt_id(current_goal_id)} (Tool {TOOL_GET_GOAL_DETAILS} unavailable).")
                else: # This case should be rare if state is consistent
                    err_msg = f"Current goal ID {_fmt_id(current_goal_id)} exists, but details unavailable
                    (Tool missing or goal not in stack)."
                    goal_context_data["current_goal"] = {"goal_id": current_goal_id, "error": err_msg}
                    base_context["errors"].append(err_msg)
                    self.logger.warning(err_msg)
        else: # No goals on the stack
             goal_context_data["current_goal"] = None
             goal_context_data["goal_stack_summary"] = []
        # Assign the gathered goal context to the main context dictionary
        base_context["current_goal_context"] = goal_context_data
        self.logger.debug(f"Gathered goal context. Current Goal ID: {_fmt_id(self.state.current_goal_id)}")

        # --- Fetch Core Context (e.g., Recent Actions, Important Memories, Key Thoughts) ---
        if self._find_tool_server(TOOL_GET_CONTEXT):
            try:
                # Fetch core context with predefined limits (FETCH_LIMIT constants)
                core_ctx_result = await self._execute_tool_call_internal(
                    TOOL_GET_CONTEXT,
                    {
                        "workflow_id": current_workflow_id,
                        # Use FETCH limits here, truncation to SHOW limits happens later if needed
                        "recent_actions_limit": CONTEXT_RECENT_ACTIONS_FETCH_LIMIT,
                        "important_memories_limit": CONTEXT_IMPORTANT_MEMORIES_FETCH_LIMIT,
                        "key_thoughts_limit": CONTEXT_KEY_THOUGHTS_FETCH_LIMIT,
                    },
                    record_action=False # Internal context fetch
                )
                if core_ctx_result.get("success"):
                    # Store the successful result and add freshness timestamp
                    base_context["core_context"] = core_ctx_result
                    base_context["core_context"]["retrieved_at"] = retrieval_timestamp # Add freshness
                    # Clean up redundant success/timing info from the nested result
                    base_context["core_context"].pop("success", None)
                    base_context["core_context"].pop("processing_time", None)
                    self.logger.debug(f"Successfully retrieved core context via {TOOL_GET_CONTEXT}.")
                else:
                    err_msg = f"Core context retrieval ({TOOL_GET_CONTEXT}) failed:
                    {core_ctx_result.get('error')}"
                    base_context["errors"].append(err_msg)
                    self.logger.warning(err_msg)
            except Exception as e:
                err_msg = f"Core context retrieval exception: {e}"
                self.logger.error(err_msg, exc_info=False)
                base_context["errors"].append(err_msg)
        else:
            self.logger.warning(f"Skipping core context: Tool {TOOL_GET_CONTEXT} unavailable.")

        # --- Get Current Working Memory (Provides active memories and focal point) ---
        focal_mem_id_from_wm: Optional[str] = None # Store focal ID if found
        working_mem_list_from_wm: List[Dict] = [] # Store memory list if found
        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, # Keep context lighter
                        "include_links": False # Links fetched separately if needed
                    },
                    record_action=False
                )
                if wm_result.get("success"):
                    # Store the entire result dict and add freshness
                    base_context["current_working_memory"] = wm_result
                    base_context["current_working_memory"]["retrieved_at"] = retrieval_timestamp # Add
                    freshness
                    # Clean up redundant fields
                    base_context["current_working_memory"].pop("success", None)
                    base_context["current_working_memory"].pop("processing_time", None)
                    # Extract focal ID and memory list for later use
                    focal_mem_id_from_wm = wm_result.get("focal_memory_id")
                    working_mem_list_from_wm = wm_result.get("working_memories", [])
                    # Log count of retrieved working memories
                    wm_count = len(working_mem_list_from_wm)
                    self.logger.info(f"Retrieved {wm_count} items from working memory (Context:
                    {_fmt_id(current_context_id)}). Focal: {_fmt_id(focal_mem_id_from_wm)}")
                else:
                    err_msg = f"Working memory retrieval failed: {wm_result.get('error')}"
                    base_context["errors"].append(err_msg)
                    self.logger.warning(err_msg)
            except Exception as e:
                err_msg = f"Working memory retrieval exception: {e}"
                self.logger.error(err_msg, exc_info=False)
                base_context["errors"].append(err_msg)
        else:
            self.logger.warning(f"Skipping working memory retrieval: Context ID missing or tool
            {TOOL_GET_WORKING_MEMORY} unavailable.")


        # --- Goal-Directed Proactive Memory Retrieval (Using Hybrid Search) ---
        # Find memories relevant to the current plan step OR current goal description
        # Determine the primary query source: plan step or goal description
        query_source_desc = "Achieve main goal" # Default if no plan/goal
        if self.state.current_plan:
            query_source_desc = self.state.current_plan[0].description
        elif goal_context_data.get("current_goal"):
            query_source_desc = goal_context_data["current_goal"].get("description", query_source_desc)

        # Formulate a query based on the current step/goal
        proactive_query = f"Information relevant to planning or executing: {query_source_desc}"
        # Prefer hybrid search, fallback to semantic
        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_FETCH_LIMIT, # Use FETCH limit
                "include_content": False # Keep context light
            }
            # Adjust weights for hybrid search
            if search_tool_proactive == TOOL_HYBRID_SEARCH:
                search_args.update({"semantic_weight": 0.7, "keyword_weight": 0.3}) # Prioritize semantics a
                bit
            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", [])
                    # Determine score key based on tool used
                    score_key = "hybrid_score" if search_tool_proactive == TOOL_HYBRID_SEARCH else
                    "similarity"
                    # Format results for context, include freshness
                    base_context["proactive_memories"] = {
                        "retrieved_at": retrieval_timestamp,
                        "memories": [
                            {
                                "memory_id": m.get("memory_id"),
                                "description": m.get("description"),
                                "score": round(m.get(score_key, 0), 3), # Include score
                                "type": m.get("memory_type") # Include type
                             }
                            for m in proactive_mems # Iterate through results
                        ]
                    }
                    if base_context["proactive_memories"]["memories"]:
                        self.logger.info(f"Retrieved {len(base_context['proactive_memories']['memories'])}
                        proactive memories using {search_tool_proactive.split(':')[-1]}.")
                else:
                    err_msg = f"Proactive memory search ({search_tool_proactive}) failed:
                    {result_content.get('error')}"
                    base_context["errors"].append(err_msg)
                    self.logger.warning(err_msg)
            except Exception as e:
                err_msg = f"Proactive memory search exception: {e}"
                self.logger.warning(err_msg, exc_info=False)
                base_context["errors"].append(err_msg)
        else:
            self.logger.warning("Skipping proactive memory search: No suitable search tool available.")

        # --- Fetch Relevant Procedural Memories (Using Hybrid Search) ---
        # Find procedural memories related to the current step/goal
        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):
            # Formulate a query focused on finding procedures/how-to steps based on the same source
            description
            proc_query = f"How to accomplish step-by-step: {query_source_desc}"
            search_args = {
                "workflow_id": current_workflow_id,
                "query": proc_query,
                "limit": CONTEXT_PROCEDURAL_MEMORIES_FETCH_LIMIT, # Use FETCH limit
                "memory_level": MemoryLevel.PROCEDURAL.value, # Explicitly filter for procedural level
                "include_content": False # Keep context light
            }
            if search_tool_proc == TOOL_HYBRID_SEARCH:
                search_args.update({"semantic_weight": 0.6, "keyword_weight": 0.4}) # Balanced weights maybe?
            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"
                    # Format results for context, include freshness
                    base_context["relevant_procedures"] = {
                         "retrieved_at": retrieval_timestamp,
                         "procedures": [
                            {
                                "memory_id": m.get("memory_id"),
                                "description": m.get("description"),
                                "score": round(m.get(score_key, 0), 3)
                            }
                            for m in proc_mems
                         ]
                    }
                    if base_context["relevant_procedures"]["procedures"]:
                        self.logger.info(f"Retrieved
                        {len(base_context['relevant_procedures']['procedures'])} relevant procedures using
                        {search_tool_proc.split(':')[-1]}.")
                else:
                    err_msg = f"Procedure search failed: {proc_result.get('error')}"
                    base_context["errors"].append(err_msg)
                    self.logger.warning(err_msg)
            except Exception as e:
                err_msg = f"Procedure search exception: {e}"
                self.logger.warning(err_msg, exc_info=False)
                base_context["errors"].append(err_msg)
        else:
            self.logger.warning("Skipping procedure search: No suitable search tool available.")

        # --- Contextual Link Traversal (Prioritizing Focal Memory) ---
        # Find memories linked to the current focus or most relevant items
        get_linked_memories_tool = TOOL_GET_LINKED_MEMORIES
        if self._find_tool_server(get_linked_memories_tool):
            mem_id_to_traverse = None
            # 1. PRIORITIZE focal memory from working memory result
            if focal_mem_id_from_wm:
                mem_id_to_traverse = focal_mem_id_from_wm
                self.logger.debug(f"Link traversal starting from focal memory:
                {_fmt_id(mem_id_to_traverse)}")

            # 2. If no focal, try the first memory *in* the working memory list
            if not mem_id_to_traverse and working_mem_list_from_wm:
                first_wm_item = working_mem_list_from_wm[0]
                if isinstance(first_wm_item, dict):
                    mem_id_to_traverse = first_wm_item.get("memory_id")
                    if mem_id_to_traverse:
                        self.logger.debug(f"Link traversal starting from first working memory item:
                        {_fmt_id(mem_id_to_traverse)}")

            # 3. If still no ID, try the first important memory from the core context
            if not mem_id_to_traverse:
                core_ctx_data = base_context.get("core_context", {})
                if isinstance(core_ctx_data, dict):
                    important_mem_list = core_ctx_data.get("important_memories", [])
                    if isinstance(important_mem_list, list) and important_mem_list:
                        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"Link traversal starting from first important memory:
                                {_fmt_id(mem_id_to_traverse)}")

            # If we found a relevant memory ID to start traversal from
            if mem_id_to_traverse:
                self.logger.debug(f"Attempting link traversal from relevant memory:
                {_fmt_id(mem_id_to_traverse)}...")
                try:
                    links_result_content = await self._execute_tool_call_internal(
                        get_linked_memories_tool,
                        {
                            "memory_id": mem_id_to_traverse,
                            "direction": "both", # Get incoming and outgoing links
                            "limit": CONTEXT_LINK_TRAVERSAL_FETCH_LIMIT, # Use FETCH limit
                            "include_memory_details": False # Just need link info for context
                         },
                        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", [])
                        # Create a concise summary of the links found
                        link_summary = {
                            "source_memory_id": mem_id_to_traverse,
                            "outgoing_count": len(outgoing_links),
                            "incoming_count": len(incoming_links),
                            "top_links_summary": [] # List of concise link descriptions
                        }
                        # Add summaries for top outgoing links (up to SHOW limit)
                        for link in outgoing_links[:CONTEXT_LINK_TRAVERSAL_SHOW_LIMIT]:
                            link_summary["top_links_summary"].append(
                                f"OUT: {link.get('link_type', 'related')} ->
                                {_fmt_id(link.get('target_memory_id'))}"
                            )
                        # Add summaries for top incoming links (up to SHOW limit)
                        for link in incoming_links[:CONTEXT_LINK_TRAVERSAL_SHOW_LIMIT]:
                            link_summary["top_links_summary"].append(
                                f"IN: {_fmt_id(link.get('source_memory_id'))} -> {link.get('link_type',
                                'related')}"
                            )
                        # Add freshness timestamp to the link summary
                        base_context["contextual_links"] = {"retrieved_at": retrieval_timestamp, "summary":
                        link_summary}
                        self.logger.info(f"Retrieved link summary for memory {_fmt_id(mem_id_to_traverse)}
                        ({len(outgoing_links)} out, {len(incoming_links)} in).")
                    else:
                        err_msg = f"Link retrieval ({get_linked_memories_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:
                # Log if no suitable starting point for link traversal was found
                self.logger.debug("No relevant memory found (focal/working/important) to perform link
                traversal from.")
        else:
            # Log if the link retrieval tool is unavailable
            self.logger.debug(f"Skipping link traversal: Tool '{get_linked_memories_tool}' unavailable.")

        # --- Context Compression Check ---
        # This should happen *after* all components are gathered
        try:
            # Estimate tokens of the fully assembled context
            estimated_tokens = await self._estimate_tokens_anthropic(base_context)
            # If estimate exceeds threshold, attempt compression
            if estimated_tokens > CONTEXT_MAX_TOKENS_COMPRESS_THRESHOLD:
                self.logger.warning(f"Context ({estimated_tokens} tokens) exceeds threshold
                {CONTEXT_MAX_TOKENS_COMPRESS_THRESHOLD}. Attempting summary compression.")
                # Check if summarization tool is available
                if self._find_tool_server(TOOL_SUMMARIZE_TEXT):
                    # Strategy: Summarize the potentially longest/most verbose part first
                    # Example: Summarize 'core_context' -> 'recent_actions' if it exists and is large
                    core_ctx = base_context.get("core_context")
                    actions_to_summarize = None
                    # Check structure before accessing potentially missing keys
                    if isinstance(core_ctx, dict):
                        actions_to_summarize = core_ctx.get("recent_actions")

                    # Only summarize if the actions list exists and is substantial (e.g., > 1000 chars JSON)
                    if actions_to_summarize and isinstance(actions_to_summarize, list) and
                    len(json.dumps(actions_to_summarize, default=str)) > 1000:
                         actions_text = json.dumps(actions_to_summarize, default=str) # Serialize for
                         summarizer
                         summary_result = await self._execute_tool_call_internal(
                              TOOL_SUMMARIZE_TEXT,
                              {
                                  "text_to_summarize": actions_text,
                                  "target_tokens": CONTEXT_COMPRESSION_TARGET_TOKENS, # Use target constant
                                  # Use specialized summarizer prompt for actions
                                  "prompt_template": "summarize_context_block", # Assuming this maps to the
                                  right prompt in UMS
                                  "context_type": "actions", # Tell summarizer what it's summarizing
                                  "workflow_id": current_workflow_id, # Provide workflow context
                                  "record_summary": False # Don't store this ephemeral summary
                              },
                              record_action=False # Internal action
                         )
                         if summary_result.get("success"):
                              # Add a note about the compression to the context
                              base_context["compression_summary"] = f"Summary of recent actions:
                              {summary_result.get('summary', 'Summary failed.')[:150]}..."
                              # Replace the original actions list in core_context with a reference to the
                              summary
                              if isinstance(core_ctx, dict):
                                   core_ctx["recent_actions"] = f"[Summarized: See compression_summary
                                   field]"
                              self.logger.info(f"Compressed recent actions. New context size estimate:
                              {await self._estimate_tokens_anthropic(base_context)} tokens")
                    else:
                        err_msg = f"Context compression tool ({TOOL_SUMMARIZE_TEXT}) failed:
                        {summary_result.get('error')}"
                        base_context["errors"].append(err_msg); self.logger.warning(err_msg)
                else:
                    # Log if no suitable large component found for compression
                    self.logger.info("No large action list found to compress, context remains large.")
            else:
                # Log if summarization tool is unavailable
                self.logger.warning(f"Cannot compress context: Tool '{TOOL_SUMMARIZE_TEXT}'
                unavailable.")
    except Exception as e:
        # Catch errors during the estimation/compression process
        err_msg = f"Error during context compression check: {e}"
        self.logger.error(err_msg, exc_info=False)
        base_context["errors"].append(err_msg)
    # <<< End Integration Block: Context Gathering >>>

    # Final status update
    base_context["status"] = "Ready" if not base_context["errors"] else "Ready with Errors"
    self.logger.info(f"Context gathering complete. Errors: {len(base_context['errors'])}. Time:
    {(time.time() - start_time):.3f}s")
    return base_context


# ----------------------------------------------------------- call-llm --
async def _call_agent_llm(
    self, goal: str, context: Dict[str, Any]
) -> Dict[str, Any]:
    """
    Calls the Anthropic LLM with the constructed prompt and context,
    parsing the response to extract the agent's decision (tool call,
    thought, or completion signal). Includes retry logic for API errors.

    Returns:
        A dictionary representing the agent's decision:
        - {"decision": "call_tool", "tool_name": str, "arguments": dict}
        - {"decision": "thought_process", "content": str}
        - {"decision": "complete", "summary": str}
        - {"decision": "error", "message": str}
        - {"decision": "plan_update", "updated_plan_steps": List[PlanStep]} (Parsed from text)
    """
    # --- 1. Prepare Prompt ---
    messages = self._construct_agent_prompt(goal, context)

    # --- 2. Define Tools for Anthropic API ---
    # Filter schemas slightly for Anthropic's specific format if needed,
    # although MCPClient's format_tools_for_anthropic should handle this.
    anthropic_tools = self.tool_schemas

    # --- 3. Execute LLM Call (with Retries) ---
    async def _do_llm_call():
        # Ensure client is available
        if not self.anthropic_client:
            raise RuntimeError("Anthropic client unavailable for LLM call")

        # Make the API call using the configured model
        response = await self.anthropic_client.messages.create(
            model=MASTER_LEVEL_AGENT_LLM_MODEL_STRING,
            max_tokens=2048, # Allow sufficient tokens for response + tool use
            system="", # System prompt is included in user message per original structure
            messages=messages, # Pass the constructed messages list
            tools=anthropic_tools, # Provide the tool schemas
            tool_choice={"type": "auto"}, # Let model decide whether to use a tool
            temperature=0.5, # Moderate temperature for balanced creativity/determinism
        )
        return response

    try:
        # Use retry wrapper for the API call
        response = await self._with_retries(
            _do_llm_call,
            max_retries=3,
            # Retry on specific Anthropic API errors and common network issues
            retry_exceptions=(
                APIConnectionError, RateLimitError, APIStatusError,
                asyncio.TimeoutError, ConnectionError
            ),
        )
        self.logger.debug(f"LLM Response Stop Reason: {response.stop_reason}")
        # --- 4. Parse LLM Response ---
        # Check if the response involves a tool call
        if response.stop_reason == "tool_use" and response.content:
            tool_calls = []
            # Iterate through content blocks to find tool_use blocks
            for block in response.content:
                if block.type == "tool_use":
                    # Found a tool use block, extract name and input arguments
                    tool_name_raw = block.name # This is the sanitized name
                    tool_args = block.input or {}
                    # Map sanitized name back to original MCP name
                    original_name =
                    self.mcp_client.server_manager.sanitized_to_original.get(tool_name_raw,
                    tool_name_raw)
                    tool_calls.append({"tool_name": original_name, "arguments": tool_args})
                    self.logger.info(f"LLM decided to call tool: {original_name}", emoji_key="hammer")

            # Currently designed to handle ONE tool call per response
            if len(tool_calls) == 1:
                return {"decision": "call_tool", **tool_calls[0]}
            elif len(tool_calls) > 1:
                # If multiple tool calls are requested, log a warning and handle only the first one
                self.logger.warning(f"LLM requested multiple tool calls ({len(tool_calls)}). Executing
                only the first: {tool_calls[0]['tool_name']}")
                return {"decision": "call_tool", **tool_calls[0]}
            else: # Should not happen if stop_reason is tool_use, but handle defensively
                 err_msg = "LLM stop reason was 'tool_use' but no tool calls found in content."
                 self.logger.error(err_msg)
                 return {"decision": "error", "message": err_msg}

        # Check if the response is plain text (thought or goal completion)
        elif response.content and response.content[0].type == "text":
            response_text = response.content[0].text.strip()
            # Check for explicit Goal Achieved signal
            if response_text.upper().startswith("GOAL ACHIEVED:"):
                summary = response_text[len("GOAL ACHIEVED:"):].strip()
                self.logger.info(f"LLM signaled goal completion. Summary: {summary}", emoji_key="tada")
                return {"decision": "complete", "summary": summary}

            # --- Check for Plan Update via Text ---
            # Heuristic check: Does the text contain something like "Updated Plan:", "New Plan:", etc.
            # followed by a list structure? This is less robust than the tool but provides a fallback.
            plan_marker = "updated plan:" # Case-insensitive check
            if plan_marker in response_text.lower():
                 try:
                      # Attempt to extract and parse the plan structure after the marker
                      plan_part = response_text[response_text.lower().find(plan_marker) +
                      len(plan_marker):].strip()
                      # Simple JSON/list parsing attempt (might need more robust parsing)
                      if plan_part.startswith("[") and plan_part.endswith("]"):
                           plan_data = json.loads(plan_part)
                           if isinstance(plan_data, list):
                                # Validate structure using Pydantic models
                                validated_plan = [PlanStep(**step_data) for step_data in plan_data]
                                self.logger.info(f"LLM proposed plan update via text
                                ({len(validated_plan)} steps).")
                                # Return the validated plan steps for application
                                return {"decision": "plan_update", "updated_plan_steps": validated_plan}
                 except (json.JSONDecodeError, ValidationError, TypeError) as e:
                      # If parsing/validation fails, treat it as a regular thought
                      self.logger.warning(f"LLM text suggested plan update, but parsing/validation
                      failed: {e}. Treating as thought.")
                      pass # Fall through to treat as thought

            # Otherwise, treat the text response as a thought process
            self.logger.info(f"LLM provided reasoning/thought: '{response_text[:100]}...'",
            emoji_key="speech_balloon")
            return {"decision": "thought_process", "content": response_text}
        else:
            # Handle unexpected empty response or unknown content type
            err_msg = f"LLM returned unexpected response content or stop reason: {response.stop_reason},
            Content: {response.content}"
            self.logger.error(err_msg)
            return {"decision": "error", "message": err_msg}

    except APIStatusError as e: # Handle API status errors specifically
```

```
            err_msg = f"Anthropic API Error {e.status_code}: {e.message}"
            self.logger.error(err_msg, exc_info=False)
            return {"decision": "error", "message": err_msg}
        except Exception as e: # Catch other exceptions during API call or parsing
            err_msg = f"Error calling LLM or parsing response: {e}"
            self.logger.error(err_msg, exc_info=True)
            return {"decision": "error", "message": err_msg}


    # ----------------------------------------------------------- main loop --
    async def run(self, goal: str, max_loops: int = 100) -> None:
        """
        The main execution loop for the agent.

        Args:
            goal: The high-level goal for the agent to achieve.
            max_loops: Maximum number of iterations before stopping automatically.
        """
        self.logger.info(f"Starting agent loop. Goal: '{goal}'. Max loops: {max_loops}")

        # Initialize or ensure workflow exists
        if not self.state.workflow_id:
            self.logger.info("No active workflow found. Creating initial workflow.")
            wf_create_args = {
                "title": f"Agent Task: {goal[:50]}...",
                "goal": goal,
                "description": f"Agent workflow initiated at {datetime.now(timezone.utc).isoformat()} to
                achieve: {goal}",
                "tags": ["agent_run", AGENT_NAME.lower()]
            }
            wf_create_result = await self._execute_tool_call_internal(
                TOOL_CREATE_WORKFLOW, wf_create_args, record_action=False
            )
            # Note: _handle_workflow_and_goal_side_effects should set self.state.workflow_id, context_id,
            stack, chain_id etc.
            if not wf_create_result.get("success") or not self.state.workflow_id:
                self.logger.critical(f"Failed to create initial workflow: {wf_create_result.get('error')}.
                Aborting.")
                return
            # Note: The root goal should now be created within _handle_workflow_and_goal_side_effects if
            successful
            elif not self.state.current_goal_id:
                 self.logger.critical("Workflow created, but root goal ID was not set. Aborting.")
                 return

        elif not self.state.current_thought_chain_id:
            # If workflow loaded but no chain ID, try setting default
            await self._set_default_thought_chain_id()
        # --- ADDED: Ensure current_goal_id is set if stack isn't empty ---
        elif self.state.goal_stack and not self.state.current_goal_id:
             self.state.current_goal_id = self.state.goal_stack[-1].get("goal_id")
             self.logger.info(f"Set current_goal_id from top of loaded stack:
             {_fmt_id(self.state.current_goal_id)}")


        # --- Main Think-Act Loop ---
        while self.state.current_loop < max_loops and not self.state.goal_achieved_flag and not
        self._shutdown_event.is_set():
            self.state.current_loop += 1
            self.logger.info(f"--- Starting Loop {self.state.current_loop}/{max_loops} (WF:
            {_fmt_id(self.state.workflow_id)}, Goal: {_fmt_id(self.state.current_goal_id)}) ---")

            # --- 1. Periodic Cognitive Tasks ---
            try:
                 await self._run_periodic_tasks()
            except Exception as e:
                 self.logger.error(f"Error during periodic tasks: {e}", exc_info=True) # Log but continue
                 loop

            # Check shutdown flag again after periodic tasks
            if self._shutdown_event.is_set(): break

            # --- 2. Gather Context ---
            context = await self._gather_context()

            # Check if no active workflow after context gathering (e.g., root workflow finished)
            if not self.state.workflow_id:
                 self.logger.info("No active workflow. Agent loop concluding.")
```

```
     break # Exit loop if workflow finished

# --- 3. Pre-computation/Pre-analysis (If needed before LLM call) ---
# (Placeholder for future logic)

# --- 4. Call LLM for Decision ---
# Clear error details before LLM call (unless needs_replan is set for error recovery)
if not self.state.needs_replan:
     self.state.last_error_details = None
# Call the LLM
decision = await self._call_agent_llm(goal, context)

# --- 5. Execute Decision ---
tool_result_content: Optional[Dict[str, Any]] = None # Store result for heuristic update
llm_proposed_plan = decision.get("updated_plan_steps") # Check if LLM proposed plan via text

# Handle decision based on type
if decision.get("decision") == "call_tool":
    tool_name = decision.get("tool_name")
    arguments = decision.get("arguments", {})

    # --- Plan Validation (Pre-Execution Check) ---
    # Ensure the intended step (likely first in plan) is valid
    if not self.state.current_plan:
        self.logger.error("Attempting tool call but plan is empty! Forcing replan.")
        self.state.needs_replan = True
        self.state.last_error_details = {"error": "Plan became empty before tool call.", "type":
        "PlanValidationError"}
    elif not self.state.current_plan[0].description:
        self.logger.error(f"Attempting tool call for invalid plan step (missing description)!
        Step ID: {self.state.current_plan[0].id}. Forcing replan.")
        self.state.needs_replan = True
        self.state.last_error_details = {"error": "Current plan step is invalid (missing
        description).", "type": "PlanValidationError", "step_id": self.state.current_plan[0].id}
    # If checks pass, execute the tool call
    elif tool_name:
        # Extract dependencies *from the current plan step* that this tool call corresponds to
        current_step_deps = self.state.current_plan[0].depends_on if self.state.current_plan else
        []
        # Execute the tool call, passing planned dependencies for checking
        tool_result_content = await self._execute_tool_call_internal(
            tool_name, arguments, record_action=True, planned_dependencies=current_step_deps
        )
    else: # Should not happen if parsing worked correctly
        self.logger.error("LLM decided to call tool but no tool name was provided.")
        self.state.last_error_details = {"error": "LLM tool call decision missing tool name.",
        "type": "LLMOutputError"}
        tool_result_content = {"success": False, "error": "Missing tool name from LLM."}

elif decision.get("decision") == "thought_process":
    # Record the thought provided by the LLM
    thought_content = decision.get("content")
    if thought_content:
        # Default to 'inference' type if LLM doesn't specify
        thought_type = ThoughtType.INFERENCE.value
        # Advanced: Could try to infer type from text content if needed
        tool_result_content = await self._execute_tool_call_internal(
            TOOL_RECORD_THOUGHT,
            {"content": thought_content, "thought_type": thought_type},
            # Allow workflow_id and thought_chain_id injection
            record_action=False # Recording thoughts isn't a primary world action
        )
    else: # Handle missing content
        self.logger.warning("LLM provided 'thought_process' decision but no content.")
        tool_result_content = {"success": False, "error": "Missing thought content from LLM."}

elif decision.get("decision") == "complete":
    # Goal achieved signal from LLM (Handles *overall* goal completion)
    self.logger.info(f"LLM signaled OVERALL goal completion: {decision.get('summary')}")
    self.state.goal_achieved_flag = True # Set flag to terminate loop
    # Update workflow status to completed if possible
    if self.state.workflow_id and self._find_tool_server(TOOL_UPDATE_WORKFLOW_STATUS):
        await self._execute_tool_call_internal(
            TOOL_UPDATE_WORKFLOW_STATUS,
            {
                "workflow_id": self.state.workflow_id,
                "status": WorkflowStatus.COMPLETED.value,
                "completion_message": decision.get('summary', 'Overall goal marked achieved by
                agent.')
            },
            record_action=False # Meta-action
        )
        # Note: _handle_workflow_and_goal_side_effects will clear WF state if root WF finished
    # Break the loop after handling completion
    break

elif decision.get("decision") == "plan_update": # Handle plan parsed from text
    self.logger.info("LLM proposed plan update via text response.")
    # The plan is already validated in _call_agent_llm
    llm_proposed_plan = decision.get("updated_plan_steps") # Already validated PlanStep list
    # We'll handle application of this plan in Step 6

elif decision.get("decision") == "error":
     # LLM or internal error during decision making
     self.logger.error(f"LLM decision error: {decision.get('message')}")
     self.state.last_action_summary = f"LLM Decision Error: {decision.get('message',
     'Unknown')[:100]}"
     # Store error details if not already set by tool execution
     if not self.state.last_error_details:
         self.state.last_error_details = {"error": decision.get('message'), "type": "LLMError"}
     self.state.needs_replan = True # Force replan after LLM error
     # action_successful remains False

# --- 6. Apply Plan Updates ---
# Priority: Plan proposed by LLM via text > Heuristic update
if llm_proposed_plan:
     try:
         # --- Plan Cycle Detection ---
         if self._detect_plan_cycle(llm_proposed_plan):
             err_msg = "LLM-proposed plan contains a dependency cycle. Applying heuristic update
             instead."
             self.logger.error(err_msg)
             self.state.last_error_details = {"error": err_msg, "type": "PlanValidationError",
             "proposed_plan": [p.model_dump() for p in llm_proposed_plan]}
             self.state.needs_replan = True # Force replan again
             # Fallback to heuristic update if LLM plan is invalid
             await self._apply_heuristic_plan_update(decision, tool_result_content)
         else:
             # Apply the LLM's validated plan
             self.state.current_plan = llm_proposed_plan
             self.state.needs_replan = False # LLM provided the plan, assume it's intended
             self.logger.info(f"Applied LLM-proposed plan update ({len(llm_proposed_plan)}
             steps).")
             # Clear errors after successful LLM plan update
             self.state.last_error_details = None
             self.state.consecutive_error_count = 0
     except Exception as plan_apply_err:
          # Catch errors applying the plan (should be rare if validation passed)
          self.logger.error(f"Error applying LLM proposed plan: {plan_apply_err}. Falling back to
          heuristic.", exc_info=True)
          self.state.last_error_details = {"error": f"Failed to apply LLM plan:
          {plan_apply_err}", "type": "PlanUpdateError"}
          self.state.needs_replan = True
          await self._apply_heuristic_plan_update(decision, tool_result_content)

elif decision.get("tool_name") != AGENT_TOOL_UPDATE_PLAN: # Heuristic only if LLM didn't
explicitly update plan
    # Apply heuristic updates if LLM didn't use the plan update tool or provide a valid text plan
    await self._apply_heuristic_plan_update(decision, tool_result_content)
# else: If AGENT_TOOL_UPDATE_PLAN was called, success/failure already handled by
_execute_tool_call_internal

# --- 7. Check Error Limit & Save State ---
if self.state.consecutive_error_count >= MAX_CONSECUTIVE_ERRORS:
    self.logger.critical(f"Max consecutive errors ({MAX_CONSECUTIVE_ERRORS}) reached. Aborting
    loop.")
    # Update workflow status to failed if possible
    if self.state.workflow_id and self._find_tool_server(TOOL_UPDATE_WORKFLOW_STATUS):
         await self._execute_tool_call_internal(
             TOOL_UPDATE_WORKFLOW_STATUS,
             {
                 "workflow_id": self.state.workflow_id,
                 "status": WorkflowStatus.FAILED.value,
                 "completion_message": f"Aborted after {MAX_CONSECUTIVE_ERRORS} consecutive
                 errors."
```

```python
                        },
                        record_action=False
                    )
                break # Exit loop

            # Save state at the end of each loop iteration
            await self._save_agent_state()

            # Optional: Small delay between loops
            # await asyncio.sleep(0.5)

        # --- Loop End ---
        if self._shutdown_event.is_set():
            self.logger.info("Agent loop terminated due to shutdown signal.")
        elif self.state.current_loop >= max_loops:
            self.logger.warning(f"Agent loop reached max iterations ({max_loops}). Stopping.")
        elif self.state.goal_achieved_flag:
            self.logger.info("Agent loop finished: Goal achieved.", emoji_key="tada")
        else:
             # Log reason if loop finished but goal_achieved_flag is false (e.g., error limit, no active
             workflow)
             if self.state.consecutive_error_count >= MAX_CONSECUTIVE_ERRORS:
                 self.logger.warning("Agent loop finished due to reaching max consecutive errors.")
             elif not self.state.workflow_id:
                 self.logger.info("Agent loop finished because no workflow is active.")
             else: # Generic case if workflow still active but loop ended
                 self.logger.warning("Agent loop finished for unexpected reason (goal not achieved, loop
                 limit not reached).")


        # Final state save and cleanup handled by shutdown() or run_agent_process() finally block


# =============================================================================
# Driver helpers & CLI entry-point
# =============================================================================

async def run_agent_process(
    mcp_server_url: str,
    anthropic_key: str,
    goal: str,
    max_loops: int,
    state_file: str,
    config_file: Optional[str],
) -> None:
    """
    Sets up the MCPClient, Agent Master Loop, signal handling,
    and runs the main agent execution loop.
    """
    # Ensure MCPClient is available
    if not MCP_CLIENT_AVAILABLE:
        print("  ERROR: MCPClient dependency not met.")
        sys.exit(1)

    mcp_client_instance = None
    agent_loop_instance = None
    exit_code = 0
    # Use standard print initially, switch to client's safe_print if available
    printer = print

    try:
        printer("Instantiating MCP Client...")
        # Initialize MCP Client (pass URL and optional config path)
        mcp_client_instance = MCPClient(base_url=mcp_server_url, config_path=config_file)
        # Use safe_print for console output if provided by the client
        if hasattr(mcp_client_instance, 'safe_print') and callable(mcp_client_instance.safe_print):
            printer = mcp_client_instance.safe_print
            log.info("Using MCPClient's safe_print for output.")

        # Configure API key if not already set in config
        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
                # Re-initialize the anthropic client instance within MCPClient if necessary
                mcp_client_instance.anthropic = AsyncAnthropic(api_key=anthropic_key)
            else:
                # Critical error if key is missing
                printer("  CRITICAL ERROR: Anthropic API key missing in config and not provided.")
                raise ValueError("Anthropic API key missing.")

        printer("Setting up MCP Client connections...")
        # Perform necessary setup (like connecting to servers/discovery)
        await mcp_client_instance.setup(interactive_mode=False) # Run in non-interactive mode

        printer("Instantiating Agent Master Loop...")
        # Create the agent instance, passing the initialized MCP client and state file path
        agent_loop_instance = AgentMasterLoop(mcp_client_instance=mcp_client_instance,
        agent_state_file=state_file)

        # --- Signal Handling Setup ---
        # Get the current asyncio event loop
        loop = asyncio.get_running_loop()
        # Create an event to signal shutdown across tasks
        stop_event = asyncio.Event()

        # Define the signal handler function
        def signal_handler_wrapper(signum):
            signal_name = signal.Signals(signum).name
            log.warning(f"Signal {signal_name} received. Initiating graceful shutdown.")
            # Set the event to signal other tasks
            stop_event.set()
            # Trigger the agent's internal shutdown method asynchronously
            if agent_loop_instance:
                 asyncio.create_task(agent_loop_instance.shutdown())
            # Avoid calling loop.stop() directly, let tasks finish gracefully

        # Register the handler for SIGINT (Ctrl+C) and SIGTERM
        for sig in [signal.SIGINT, signal.SIGTERM]:
            try:
                loop.add_signal_handler(sig, signal_handler_wrapper, sig)
                log.debug(f"Registered signal handler for {sig.name}")
            except ValueError: # Might fail if handler already registered
                log.debug(f"Signal handler for {sig.name} may already be registered.")
            except NotImplementedError: # Signal handling might not be supported (e.g., Windows sometimes)
                log.warning(f"Signal handling for {sig.name} not supported on this platform.")


        printer("Initializing agent...")
        if not await agent_loop_instance.initialize():
             printer("  Agent initialization failed. Exiting.")
             exit_code = 1
             return # Exit early if initialization fails

        printer(f"Running Agent Loop for goal: \"{goal}\"")
        # Create tasks for the main agent run and for waiting on the stop signal
        run_task = asyncio.create_task(agent_loop_instance.run(goal=goal, max_loops=max_loops))
        stop_task = asyncio.create_task(stop_event.wait())

        # Wait for either the agent run to complete OR the stop signal to be received
        done, pending = await asyncio.wait(
            {run_task, stop_task},
            return_when=asyncio.FIRST_COMPLETED # Return as soon as one task finishes
        )

        # Handle shutdown signal completion
        if stop_task in done:
             printer("\n[yellow]Shutdown signal processed. Waiting for agent task to finalize...[/yellow]")
             # If the agent task is still running, attempt to cancel it
             if run_task in pending:
                 run_task.cancel()
                 try:
                     # Wait for the cancellation to be processed
                     await run_task
                 except asyncio.CancelledError:
                      log.info("Agent run task cancelled gracefully after signal.")
                 except Exception as e:
                      # Log error if cancellation failed unexpectedly
                      log.error(f"Exception during agent run task finalization after signal: {e}",
                      exc_info=True)
             # Set standard exit code for signal interruption (like Ctrl+C)
             exit_code = 130

        # Handle normal agent completion or error completion
        elif run_task in done:
             try:
```

```python
                run_task.result() # Check for exceptions raised by the run task
                log.info("Agent run task completed normally.")
            except Exception as e:
                # If the run task raised an exception, log it and set error exit code
                printer(f"\n  Agent loop finished with error: {e}")
                log.error("Agent run task finished with an exception:", exc_info=True)
                exit_code = 1


    except KeyboardInterrupt:
        # Fallback for Ctrl+C if signal handler doesn't catch it (e.g., during setup)
        printer("\n[yellow]KeyboardInterrupt caught (fallback).[/yellow]")
        exit_code = 130
    except ValueError as ve: # Catch specific config/value errors during setup
        printer(f"\n  Configuration Error: {ve}")
        exit_code = 2
    except Exception as main_err:
        # Catch any other critical errors during setup or main execution
        printer(f"\n  Critical error during setup or execution: {main_err}")
        log.critical("Top-level execution error", exc_info=True)
        exit_code = 1
    finally:
        # --- Cleanup Sequence ---
        printer("Initiating final shutdown sequence...")
        # Ensure agent shutdown method is called (might be redundant if signaled, but safe)
        if agent_loop_instance and not agent_loop_instance._shutdown_event.is_set():
             printer("Ensuring agent loop shutdown...")
             await agent_loop_instance.shutdown() # Call directly if not already shutting down
        # Ensure MCP client connections are closed
        if mcp_client_instance:
            printer("Closing MCP client connections...")
            try:
                await mcp_client_instance.close()
            except Exception as close_err:
                printer(f"[red]Error closing MCP client:[/red] {close_err}")
        printer("Agent execution finished.")
        # Exit the process only if running as the main script
        if __name__ == "__main__":
            # Short delay to allow logs/output to flush before exiting
            await asyncio.sleep(0.5)
            sys.exit(exit_code)

# Main execution block when script is run directly
if __name__ == "__main__":
    # (Keep existing __main__ block for configuration loading and running)
    # Load configuration from environment variables or defaults
    MCP_SERVER_URL = os.environ.get("MCP_SERVER_URL", "http://localhost:8013")
    ANTHROPIC_API_KEY = os.environ.get("ANTHROPIC_API_KEY")
    AGENT_GOAL = os.environ.get(
        "AGENT_GOAL", # Default goal for testing/example
        "Create workflow 'Tier 3 Test': Research Quantum Computing impact on Cryptography.",
    )
    MAX_ITER = int(os.environ.get("MAX_ITERATIONS", "30")) # Default max loops
    STATE_FILE = os.environ.get("AGENT_STATE_FILE", AGENT_STATE_FILE) # Use constant default
    CONFIG_PATH = os.environ.get("MCP_CLIENT_CONFIG") # Optional MCPClient config file path

    # Validate essential configuration
    if not ANTHROPIC_API_KEY:
        print("  ERROR: ANTHROPIC_API_KEY missing in environment variables.")
        sys.exit(1)
    if not MCP_CLIENT_AVAILABLE:
        # This check happens earlier, but keep for robustness
        print("  ERROR: MCPClient dependency missing.")
        sys.exit(1)

    # Display configuration being used before starting
    print(f"--- {AGENT_NAME} ---")
    print(f"Memory System URL: {MCP_SERVER_URL}")
    print(f"Agent Goal: {AGENT_GOAL}")
    print(f"Max Iterations: {MAX_ITER}")
    print(f"State File: {STATE_FILE}")
    print(f"Client Config: {CONFIG_PATH or 'Default internal config'}")
    print(f"Log Level: {logging.getLevelName(log.level)}")
    print("Anthropic API Key: Found")
    print("-----------------------------------------")


    # Define the main async function to run the agent process
```

```python
async def _main() -> None:
    await run_agent_process(
        MCP_SERVER_URL,
        ANTHROPIC_API_KEY,
        AGENT_GOAL,
        MAX_ITER,
        STATE_FILE,
        CONFIG_PATH,
    )

# Run the main async function using asyncio.run()
try:
    asyncio.run(_main())
except KeyboardInterrupt:
    # Catch Ctrl+C during initial asyncio.run setup if signal handler isn't active yet
    print("\n[yellow]Initial KeyboardInterrupt detected. Exiting.[/yellow]")
    sys.exit(130)
```