Project Archon: Design Document for an Autonomous AI Agent Framework

Overview

This document outlines the architecture for a modular, extensible framework enabling AI agents to autonomously plan and execute multi-step tasks using external tools. The key challenge is designing a robust, loop-safe execution engine that balances agent autonomy with predictable control, effectively translating high-level goals into reliable, observable actions.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

> **Milestone(s):** 1, 2, 3, 4, 5

1. Context and Problem Statement

Building an AI agent that can autonomously complete multi-step tasks in the real world is not just a technical challenge—it's an architectural one. This section establishes the core problem space, introduces an intuitive mental model for thinking about the solution, and analyzes the landscape of existing frameworks to identify the specific gap this project aims to fill.

Mental Model: The Delegating Executive

Imagine you are a senior executive with a broad goal: "Improve our company's market position." You cannot accomplish this alone. Instead, you:

1. **Decompose the goal:** You break it into concrete tasks: "Research competitors," "Analyze Q3 financials," "Draft a strategy memo."
2. **Delegate to specialists:** You assign the research to an analyst, the financials to an accountant, and the memo to a writer. You don't need to know *how* they perform their work, only *what* they need (the task) and *what* they produce (the result).
3. **Manage a plan:** You track task dependencies (the memo can't start until research and analysis are complete), sequence them, and adjust when something goes wrong.
4. **Learn from a corporate memory:** You refer to past reports and meeting notes to inform decisions, and you archive the outcomes of this project for future reference.

Project Archon is the system architecture for an **"AI Executive."** Its core components are direct analogs to this real-world scenario:

- **The Planner** is the executive's project management skill, responsible for breaking down nebulous goals into actionable subtasks.
- **The Tool System** represents the team of specialized employees (the analyst, the accountant, the writer). Each tool is a well-defined capability with a clear interface (a job description and expected inputs/outputs).
- **The ReAct Loop** is the executive's daily operating rhythm: `Think` (consider the next step), `Act` (delegate a task), `Observe` (review the result), and repeat.
- **The Memory System** is the executive's notebook (short-term), whiteboard (working memory for the current project), and corporate archive (long-term, searchable memory).
- **The Orchestrator** (in multi-agent setups) is the department head, coordinating multiple executives (agents) who each have their own specialized teams (tools).

This mental model transforms an abstract engineering problem into a relatable system design challenge. The goal is not to build a monolithic "AI brain," but to design the **organizational and operational framework** that enables a Large Language Model (LLM) to function as a competent, autonomous executive.

The Problem of Autonomous Action

A raw Large Language Model (LLM) is a brilliant strategist trapped in a conference room. It can reason, analyze, and propose plans with remarkable sophistication, but it cannot *act* on the world. It lacks the ability to execute code, query a database, call an API, or manipulate a file. This fundamental limitation stems from several interconnected challenges:

1. **Tool Use & Integration:** An LLM's knowledge is static, frozen at its training cut-off. It cannot access real-time information (stock prices, weather) or perform concrete computations. To be useful, it must be able to **safely and reliably invoke external tools**. This requires a consistent interface for describing tools, validating their inputs, executing them, and interpreting their outputs—all while the LLM operates in unstructured natural language.

2. **Planning & State Management:** Real-world tasks are rarely single-step. "Book a business trip" involves checking calendars, finding flights, booking hotels, and submitting an expense report—a graph of dependent steps. An agent must be able to **decompose a high-level goal, formulate a plan, track its progress through multiple states, and adapt when steps fail**. This requires maintaining a persistent, structured representation of the task's state beyond the LLM's limited context window.

3. **Loop Control & Halting:** If you give an LLM the ability to act, you must also give it the ability to **know when to stop**. An agent without a termination condition can loop infinitely, retrying failed actions or pursuing irrelevant tangents. The system must enforce guardrails like iteration limits, provide clear signals for task completion, and have fallback mechanisms to gracefully abort unproductive loops.

4. **Context & Memory Limitation:** An LLM's context window is a small, volatile scratchpad. To work on a long task, an agent must **selectively remember and recall relevant information**. This includes the immediate conversation history, key facts learned during execution, and lessons from past similar tasks. Without effective memory management, the agent suffers from "amnesia," repeatedly re-discovering the same information or losing track of its overall objective.
5. **Error Handling & Robustness:** The external world is messy. Tools time out, APIs return errors, and user instructions are ambiguous. An autonomous agent cannot simply crash. It must **detect failures, classify them (transient vs. permanent), attempt recovery (e.g., retries), and escalate unsolvable problems** to a fallback strategy or the user. This requires a systematic approach to error propagation and observability throughout the execution pipeline.

> The core architectural challenge is **bridging the semantic gap** between the LLM's unstructured reasoning and the structured, procedural world of software tools, while maintaining safety, observability, and control.

Existing Approaches and Trade-offs

Several frameworks have attempted to solve parts of this problem. Understanding their strengths and weaknesses clarifies the design space for Project Archon.

Framework	Primary Strength	Key Weakness / Trade-off	Relevance to Our Goals
LangChain / LangGraph	**Ecosystem & Integration.** Offers a vast, pre-built collection of tools, agents, and memory implementations. Excellent for rapidly prototyping common patterns (ReAct, OpenAI Functions).	**Abstraction Leak & Complexity.** The high-level `AgentExecutor` can feel like a black box. Debugging requires understanding intricate chains of callbacks and intermediate states. Its "kitchen sink" approach can lead to bloated, hard-to-reason-about applications for custom needs.	**Reference Implementation.** We will study its patterns (especially the ReAct and tool-calling designs) but aim for a more transparent, modular, and explicit architecture where data flow and state are first-class concepts.
AutoGPT / BabyAGI	**Ambition & Goal-Oriented Design.** Pioneered the concept of a fully autonomous agent with long-running tasks, planning, and internet-enabled tools. Strong focus on recursive decomposition and persistent memory.	**Instability & Brittleness.** Early versions were notorious for getting stuck in loops or making poor planning decisions. Highlights the critical need for robust loop control, planning validation, and failure recovery—problems we must solve directly.	**Cautionary Tale & Inspiration.** We embrace its vision of full autonomy but must engineer far greater reliability through structured planning (DAGs vs. simple lists), better termination logic, and systematic error handling.
OpenAI Assistants API / GPTs	**Simplicity & Managed Infrastructure.** Provides a hosted, easy-to-use platform for creating agents with file search, code execution, and function calling. Handles much of the orchestration and context management automatically.	**Vendor Lock-in & Limited Control.** The agent loop and memory are opaque and controlled by the provider. It's difficult to implement custom planning logic, complex tool interactions, or fine-grained monitoring. Not a framework for building *systems*.	**Non-Goal.** Our framework is inherently multi-provider and requires deep, customizable control over the agent's internals for research and complex applications. We are building the *platform*, not a client for one.
CrewAI	**Multi-Agent Specialization.** Explicitly models roles, goals, and backstories for collaborative agents. Good abstraction for defining workflows of specialized agents.	**Higher-Level Abstraction.** While good for orchestrating agent *crews*, it often builds upon other frameworks (like LangChain) for the core agent logic, potentially inheriting their complexities. Focuses more on choreography than the engine of a single agent.	**Inspiration for Milestone 5.** We will adopt its mental model of role-based specialization and task delegation but implement it within our own, more transparent agent core and communication protocol.
Raw LLM + Function Calling	**Maximum Control & Simplicity.** Directly using an LLM provider's function-calling API with a handwritten loop offers complete transparency and minimal dependencies.	**Reinventing the Wheel.** Requires manually implementing planning, memory, error handling, and tool orchestration for any non-trivial task. Scales poorly and is error-prone.	**Foundation.** This is essentially the starting point for Milestone 2 (ReAct Loop). Our framework systematizes and extends this pattern with essential production-grade features.

Architecture Decision: Build a Transparent, Modular Framework Over a Monolithic Library

- **Context:** Learners and developers need to understand the internal mechanics of an AI agent, not just wire together opaque components. Existing libraries often hide critical logic, making debugging and customization difficult.
- **Options Considered:**
  1. **Wrap an existing library (e.g., LangChain):** Provide a simplified interface over a complex, battle-tested codebase.
  2. **Build a new, integrated monolithic framework:** Create a tightly-coupled system optimized for our specific use cases.
  3. **Build a modular, explicit framework:** Design independent, composable components with clear interfaces and data flow.
- **Decision: Option 3: Build a modular, explicit framework.**
- **Rationale:** The primary goal is *educational clarity* and *architectural control*. A modular design forces clean separation of concerns (Tools, Planning, Memory, Loop), making each component's responsibility and data model explicit. This allows learners to understand, implement, and test each piece in isolation. It also accommodates future evolution, as components can be swapped (e.g., different memory backends, planning algorithms) without breaking the entire system.
- **Consequences:** We must design and document clear interfaces between components. There will be some initial overhead compared to using a pre-built library, but the resulting system will be more debuggable, customizable, and pedagogically valuable.

The following table summarizes the key design trade-offs we are consciously making:

Design Choice	Our Approach	Rationale & Consequence
Modularity	High - Strict separation between Tool, Planning, Memory, and Loop components.	**Pro:** Enables testing, understanding, and replacement in isolation. **Con:** Requires careful interface design and more initial "glue" code.
Transparency	Data flow and state are explicit, logged, and inspectable.	**Pro:** Essential for debugging and learning. Agents are not black boxes. **Con:** Slightly more complex API surface than a single `run()` call.
Control	The developer defines the loop, plans, and memory strategies.	**Pro:** Enables fine-tuning for specific reliability and safety needs. **Con:** More responsibility on the framework user; less "magic."
Provider Agnosticism	Core framework is LLM-agnostic; providers are injected via a standard interface.	**Pro:** Avoids vendor lock-in and allows benchmarking. **Con:** Must abstract over differences in function-calling formats and tokenization.

Implementation Guidance

While this section is primarily conceptual, establishing a solid project foundation is critical. Here is the recommended starting structure and core type definitions.

A. Technology Recommendations Table

Component	Simple Option (For Learning/MVP)	Advanced / Production Option
Core Language & Runtime	Python 3.11+ with `asyncio`	Python 3.11+ with `asyncio`; consider `anyio` for advanced task pools.
LLM Provider Interface	Direct HTTP calls via `httpx` to OpenAI/Anthropic APIs.	Abstract `LLMProvider` class supporting multiple backends (OpenAI, Anthropic, Local via `llama.cpp`).
Serialization & Validation	Pydantic V2 for data models and basic validation.	Pydantic V2 + `typeguard` for runtime type checks in performance-critical paths.
Logging & Observability	Python `logging` module with JSON formatting.	`structlog` for structured logging, integrated with OpenTelemetry for traces.

B. Recommended File/Module Structure

Create this directory structure from the start to enforce separation of concerns.

```
project_archon/
├── pyproject.toml              # Project metadata and dependencies
├── README.md
│
├── archon/                     # Main package
│   ├── __init__.py
│   ├── core/                   # Agent Core & ReAct Loop (Milestone 2)
│   │   ├── __init__.py
│   │   ├── agent.py            # Main Agent class, ReAct loop controller
│   │   ├── state.py            # AgentState, Task, SubTask definitions
│   │   └── prompts/            # Prompt templates
│   │       └── react.py
│   │
│   ├── tools/                  # Tool System (Milestone 1)
│   │   ├── __init__.py
│   │   ├── base.py             # ToolBase, ToolRegistry
│   │   ├── engine.py           # ToolExecutionEngine
│   │   └── builtin/            # Built-in tools
│   │       ├── __init__.py
│   │       ├── calculator.py
│   │       └── web_search.py
│   │
│   ├── planning/               # Planning & Decomposition (Milestone 3)
│   │   ├── __init__.py
│   │   ├── planner.py         # LLMPlanner, PlanExecutor
│   │   └── dag.py             # Graph structures for task dependencies
│   │
│   ├── memory/                 # Memory System (Milestone 4)
│   │   ├── __init__.py
│   │   ├── base.py            # Memory interfaces
│   │   ├── conversational.py  # ConversationMemory (short-term)
│   │   ├── working.py         # WorkingMemory
│   │   └── long_term.py       # VectorMemory backend
│   │
│   ├── multi_agent/           # Multi-Agent Collaboration (Milestone 5)
│   │   ├── __init__.py
│   │   ├── message.py         # Message protocol
│   │   ├── role.py            # Role assignment
│   │   └── orchestrator.py    # RoleBasedOrchestrator
│   │
│   └── utils/                  # Shared utilities
│       ├── __init__.py
│       ├── logging.py
│       └── validation.py      # JSON Schema helpers
│
└── tests/                      # Test suite mirroring the structure above
    ├── __init__.py
    ├── test_tools/
    └── test_core/
```

C. Infrastructure Starter Code: Core Data Types

These Pydantic models form the backbone of the framework's state and communication. Place them in `archon/core/state.py`.

PYTHON

```python
from enum import Enum

from typing import Any, Dict, List, Optional, Union

from pydantic import BaseModel, Field


class ToolResultStatus(str, Enum):

    """Status of a tool execution."""

    SUCCESS = "success"

    ERROR = "error"  # Permanent failure

    TIMEOUT = "timeout"


class ToolResult(BaseModel):

    """Structured result from a tool execution."""

    content: str = Field(description="Textual result for the LLM.")

    status: ToolResultStatus = Field(default=ToolResultStatus.SUCCESS)

    error_message: Optional[str] = Field(default=None, description="Populated if status is ERROR or TIMEOUT.")

    metadata: Dict[str, Any] = Field(default_factory=dict, description="Additional structured data (e.g., raw API response).")


class SubTaskStatus(str, Enum):

    """Status of a subtask within a plan."""

    PENDING = "pending"

    EXECUTING = "executing"

    COMPLETED = "completed"

    FAILED = "failed"

    BLOCKED = "blocked"  # Waiting on a dependency


class SubTask(BaseModel):

    """A single, actionable step within a larger Task or Plan."""

    id: str = Field(description="Unique identifier for the subtask.")

    description: str = Field(description="Natural language description of what to do.")

    expected_output: Optional[str] = Field(default=None, description="What a successful completion looks like.")

    tool_name: Optional[str] = Field(default=None, description="Name of the tool to execute, if applicable.")

    tool_args: Optional[Dict[str, Any]] = Field(default=None, description="Arguments for the tool.")

    dependencies: List[str] = Field(default_factory=list, description="List of `SubTask.id` that must complete before this one.")

    status: SubTaskStatus = Field(default=SubTaskStatus.PENDING)

    result: Optional[ToolResult] = Field(default=None, description="Result after execution.")


class Task(BaseModel):

    """A user's high-level request, which may be decomposed into SubTasks."""

    id: str = Field(default_factory=lambda: f"task_{uuid.uuid4().hex[:8]}")

    original_query: str = Field(description="The user's initial request.")

    goal: str = Field(description="Refined, actionable objective.")

    subtasks: List[SubTask] = Field(default_factory=list, description="The decomposition of this task.")

    context: Dict[str, Any] = Field(default_factory=dict, description="Shared state/data across subtasks.")
```

```
class AgentState(str, Enum):
    """High-level state of the Agent Core."""
    IDLE = "idle"
    PLANNING = "planning"
    EXECUTING = "executing"  # In the ReAct loop for a subtask
    REPLANNING = "replanning"
    FINISHED = "finished"
    ERROR = "error"
```

D. Core Logic Skeleton Code: The Agent's Main Loop

This is a high-level skeleton for the Agent Core's main entry point (`archon/core/agent.py`). The detailed ReAct loop will be built in Milestone 2.

```python
from typing import Optional

from archon.core.state import Task, AgentState

from archon.planning.planner import Planner

from archon.tools.base import ToolRegistry


class Agent:

    """The central controller for an autonomous AI agent."""


    def __init__(self, planner: Planner, tool_registry: ToolRegistry):

        self.planner = planner

        self.tools = tool_registry

        self.state: AgentState = AgentState.IDLE

        self.current_task: Optional[Task] = None


    async def run_task(self, user_query: str) -> str:

        """

        Primary public method: takes a user query and returns a final answer.

        This orchestrates the high-level flow: Plan -> Execute (ReAct) -> Return.

        """

        # TODO 1: Transition state from IDLE to PLANNING.

        # TODO 2: Use the planner to decompose the user_query into a Task object with subtasks.

        # TODO 3: Transition state to EXECUTING.

        # TODO 4: For each subtask in the plan (respecting dependencies):

        #     a. Execute the subtask using the _execute_subtask method (which will run the ReAct loop).

        #     b. If a subtask fails, trigger replanning (transition to REPLANNING, then back to EXECUTING).

        # TODO 5: When all subtasks are complete, compile a final answer from the results.

        # TODO 6: Transition state to FINISHED and return the final answer.

        # TODO 7: Throughout, handle exceptions and transition to ERROR state if unrecoverable.

        pass


    async def _execute_subtask(self, subtask: SubTask) -> ToolResult:

        """

        Executes a single subtask using the ReAct loop.

        This is the core of Milestone 2.

        """

        # TODO: This will be implemented in detail later.

        # It will involve:

        # 1. Setting up the LLM prompt with context, tools, and the subtask description.

        # 2. Running the Think -> Act -> Observe loop.

        # 3. Returning a structured ToolResult.

        pass
```

E. Language-Specific Hints (Python)

- Use `asyncio` from the start for all I/O (LLM calls, tool execution). This is non-negotiable for performance in multi-tool or multi-agent scenarios.
- **Type Hint Everything:** Use Python's type hints (`str` , `List[SubTask]` , `Optional[ToolResult]`) combined with Pydantic for runtime validation and excellent IDE support.
- Use `Enum` for states (like `AgentState` , `SubTaskStatus`) to avoid string typos and make code more readable.
- For the project structure, use a `pyproject.toml` with `[project]` and `[build-system]` sections (modern Python standard). Specify dependencies like `pydantic>=2.5` , `httpx` , `openai` .

> **Milestone(s):** 1, 2, 3, 4, 5

2. Goals and Non-Goals

This section defines the explicit requirements the framework must satisfy and clearly outlines what is out of scope to bound the project. Establishing these boundaries is critical for a project of this complexity—without them, scope creep and unclear expectations would inevitably derail development. We approach this by first defining what the framework **must** achieve (functional and non-functional goals), then explicitly stating what it **will not** do (non-goals). This clarity ensures that all engineering effort is focused on solving the core problem: building a robust, extensible system where an AI agent can reliably translate high-level goals into a sequence of tool-using actions.

2.1 Functional Goals

Functional goals define the core capabilities the framework must deliver. Each goal maps directly to one of the five project milestones and addresses a specific aspect of the **Semantic Gap**—the disconnect between an LLM's natural language reasoning and the structured world of executable software tools. The framework succeeds when an **AI Executive** can autonomously navigate this gap to accomplish complex, multi-step tasks.

> **Design Insight:** Think of functional goals as the "job description" for the framework. Just as you would hire an executive by evaluating their ability to plan, delegate, and adapt, we evaluate the framework by its ability to perform these cognitive functions through code.

The table below enumerates the primary functional requirements, organized by the milestone they originate from and the core capability they enable.

Milestone	Core Capability	Functional Goal	Success Criteria (How We Measure It)
1: Tool System	**Action Execution**	Provide a standardized, safe interface for the agent to interact with the external world.	The agent can correctly invoke any registered tool with validated parameters, receive structured results, and handle execution failures gracefully.
	Extensibility	Allow new capabilities to be added at runtime without modifying the core agent logic.	A developer can write a new Python class implementing the `Tool` interface, register it, and the agent immediately recognizes and can use it.
2: ReAct Loop	**Reasoned Decision-Making**	Implement the core cognitive cycle where the agent reasons about a situation, selects an action, observes the outcome, and repeats.	For a given query (e.g., "What's the weather in Tokyo and translate the summary to French?"), the agent produces a coherent chain of thought, tool calls, and observations leading to a correct final answer.
	Structured Output Handling	Reliably parse the LLM's free-form text output into structured tool-calling instructions.	The action parser successfully extracts `tool_name` and `tool_args` from LLM responses 95%+ of the time, with clear error fallbacks for malformed outputs.
3: Planning & Decomposition	**Complex Task Breakdown**	Automatically decompose a high-level, ambiguous user goal into a concrete sequence (or graph) of executable subtasks.	Given "Plan a 3-day trip to Paris," the agent produces a logical DAG of subtasks like "Research flights," "Find hotels near attractions," and "Create daily itinerary."
	Adaptive Execution	Execute subtasks in correct dependency order, detect failures, and re-plan when the initial approach fails.	If a "Book hotel" subtask fails due to no availability, the agent replans to "Search for alternative hotels" or "Adjust travel dates" without human intervention.
4: Memory & Context	**Context Retention**	Maintain, retrieve, and manage conversation history and task state across multiple interaction turns.	In a multi-turn dialogue, the agent remembers user preferences stated earlier (e.g., "I'm vegetarian") and uses them to filter restaurant recommendations in later steps.
	Relevant Recall	Retrieve semantically relevant past experiences from long-term storage to inform current reasoning.	When asked "How do I debug a Python segmentation fault?", the agent retrieves and references a past conversation where the user successfully used `gdb` for C++ debugging.
5: Multi-Agent Collaboration	**Specialized Coordination**	Enable multiple agents with different roles to work together on a single task through structured communication.	A "Research & Write" task is automatically decomposed: a `ResearcherAgent` finds sources, a `WriterAgent` drafts content, and an `EditorAgent` reviews for clarity, all coordinated by an `Orchestrator`.
	Conflict Resolution	Merge or adjudicate contradictory outputs from different agents into a coherent final result.	When two `CoderAgent` instances propose different implementations, the `Orchestrator` applies a defined strategy (e.g., choose the simpler one, merge features, or ask for clarification).

These functional goals combine to create a complete **AI Executive** system. The agent can perceive a goal (via user query), formulate a plan (decomposition), execute the plan through a cycle of reasoning and tool use (ReAct), while drawing on both immediate context and past experience (memory), and if needed, delegate parts of the work to specialized peers (multi-agent collaboration). The ultimate functional goal is encapsulated in the primary public API method: `Agent.run_task(user_query: str) -> str`, which must return a useful, actionable result for any well-formed query within the system's capability boundaries.

2.2 Non-Functional Goals

Non-functional goals define the *quality attributes* of the system—how it should behave, rather than what it does. These are critical for the framework's usability, maintainability, and safety in production-like scenarios. A system that meets all functional goals but is brittle, opaque, or dangerous would fail as a foundation for building reliable AI agents.

> **Mental Model: The Framework as Public Infrastructure.** Think of the framework as a city's electrical grid. Functional goals ensure it delivers power (the lights turn on). Non-functional goals ensure it's reliable during storms, extensible for new neighborhoods, safe to avoid electrocution, and maintainable so engineers can fix outages quickly. All are equally vital for public trust.

Quality Attribute	Goal	Rationale & Concrete Measurement
Extensibility	The framework must allow new components (tools, memory backends, planning strategies) to be added with minimal changes to existing code.	**Rationale:** AI capabilities evolve rapidly; the framework must not become a bottleneck. **Measurement:** A developer can add a new tool by creating a single Python class and registering it, without modifying any core framework files.
Safety & Guardrails	The agent must operate within defined safety boundaries, with mechanisms to prevent harmful, unauthorized, or infinitely looping actions.	**Rationale:** Autonomous tool use carries inherent risk (e.g., deleting files, making unauthorized API calls). **Measurement:** All tool executions are wrapped with timeouts, permission checks (if configured), and a maximum iteration limit for the ReAct loop. Critical tools (e.g., file write, shell exec) require explicit opt-in.
Debuggability & Observability	The internal state and decision-making process of the agent must be inspectable at runtime, with structured logs tracing the complete thought-action chain.	**Rationale:** AI agents are non-deterministic; when they fail, developers need clear visibility into *why*. **Measurement:** Every agent run produces a chronological log of `Thought`, `Action`, `Observation` tuples, and the state of the task DAG is queryable at any point.
Performance & Latency	The framework should minimize overhead, allowing the agent's speed to be bounded primarily by LLM API latency and tool execution time.	**Rationale:** Complex tasks may require dozens of LLM calls and tool invocations; inefficient framework code would make tasks impractically slow. **Measurement:** Framework overhead (parsing, state management, memory retrieval) adds <100ms per agent iteration. Independent subtasks can execute in parallel when dependencies allow.
LLM Agnosticism	The core agent logic should be decoupled from any specific LLM provider or API, allowing easy swapping of models (GPT-4, Claude, open-source, etc.).	**Rationale:** LLM landscape is competitive and evolving; lock-in is undesirable. **Measurement:** The LLM interface is abstracted behind a generic `LLMClient` class; switching providers requires changing only configuration, not core agent code.
Modularity & Clean Boundaries	Components (Tool System, Memory, Planner) must have well-defined interfaces and minimal cross-dependencies, allowing them to be tested, developed, and replaced in isolation.	**Rationale:** Enables incremental development (per our milestones) and allows users to customize or replace components (e.g., use a different vector database). **Measurement:** Each component's public interface is documented and has <3 direct dependencies on other framework components.
Error Resilience	The system must degrade gracefully when components fail (LLM unavailable, tool times out, network error) and provide clear, actionable error messages to the calling application.	**Rationale:** In a distributed, API-dependent system, failures are inevitable. **Measurement:** Tool failures are caught and presented as structured `ToolResult` with `ERROR` status; the agent can replan. LLM API errors trigger retries with exponential backoff. No unhandled exception should crash the entire agent process.

These non-functional goals are not afterthoughts—they are first-class design constraints that will shape the architecture. For instance, **safety** drives the design of the tool execution engine with sandboxing and timeouts. **Debuggability** leads to the structured logging of the entire ReAct loop. **Modularity** justifies the clear separation between the Planner, Memory, and Tool System components.

2.3 Explicit Non-Goals

Explicitly stating what the framework will **not** do is equally important. It sets realistic expectations, prevents scope creep, and allows the team to focus on the core value proposition. A common pitfall in ambitious projects is trying to solve every related problem, resulting in a bloated, half-finished system.

> **Design Principle: Sharp Boundaries Foster Focus.** A sculptor removes marble to reveal the statue. Similarly, we define non-goals to chisel away peripheral concerns, leaving a sharp, focused framework that excels at its primary mission: autonomous task execution via LLM-driven planning and tool use.

Non-Goal	Explanation & Rationale	What We Might Do Instead
Train or fine-tune LLM models	The framework is a *consumer* of LLMs, not a model trainer. We assume a capable pre-trained LLM is available via API or local inference.	We will provide interfaces for popular LLM APIs (OpenAI, Anthropic) and local inference servers (vLLM, Ollama), but the training/fine-tuning process is entirely out of scope.
Provide a graphical user interface (GUI) or chatbot frontend	The framework is a backend library/engine. It provides a programmatic API (`Agent.run_task()`). Building web UIs, Discord bots, or mobile apps is left to the application layer.	We will ensure the API is clean and well-documented so that others can easily build UIs on top. Example CLI drivers may be provided for testing, but they are not the primary deliverable.
Guarantee 100% safety or prevent all misuse	While we will implement significant guardrails (timeouts, permission checks), we cannot eliminate all risk from autonomous AI systems. Malicious users or poorly designed tools can still cause harm.	We will design for "safety by default" (tools are opt-in, sandboxed where possible) and provide clear warnings and configuration knobs for dangerous operations. Ultimate responsibility lies with the developer integrating the framework.
Solve general artificial intelligence (AGI)	The framework orchestrates LLM reasoning and tool use; it does not create new cognitive capabilities. The agent's intelligence is bounded by the underlying LLM's capabilities and the tools provided.	We focus on the *orchestration* problem: effectively using the capabilities that LLMs and tools already provide. Breakthroughs in LLM reasoning must come from model research, not our framework.
Replace human judgment in critical domains	The framework is an augmentation tool, not an autonomous decision-maker for high-stakes domains like medical diagnosis, legal advice, or financial trading without human oversight.	We will include features like human-in-the-loop confirmation for specific tool calls (a future extension), but the base framework assumes the agent operates autonomously within its defined sandbox.
Handle multi-modal inputs/outputs natively (images, audio, video)	The initial framework is designed for text-centric tasks. While tools could be built to process images/audio, the core agent reasoning, memory, and communication are text-based.	The architecture can be extended later to support multi-modal LLMs and tools, but for Milestones 1-5, we assume textual interaction.
Provide built-in tools for every possible domain	We will include a few illustrative built-in tools (calculator, web search, safe code execution). However, we will not build hundreds of domain-specific tools (e.g., for CRM, accounting, CAD).	The framework's value is in enabling *others* to build tools easily. We will focus on making the tool interface and registry exceptionally simple and powerful.
Manage infrastructure scaling (load balancing, Kubernetes)	The framework is a library that runs within a single process. Scaling horizontally to handle thousands of concurrent agent sessions is an application/deployment concern.	We will ensure the framework is stateless where possible (persisting state via provided memory interfaces) to facilitate scaling by the embedding application.

By declaring these non-goals, we make intentional trade-offs. We choose depth over breadth, focusing on creating the best possible *engine* for AI autonomy, not the entire *vehicle*. This allows us to invest engineering effort where it matters most: robust loops, clean abstractions, and reliable execution—the foundations upon which countless AI agent applications can be built.

---

3. High-Level Architecture

> **Milestone(s):** 1, 2, 3, 4, 5

This section presents the architectural blueprint for Project Archon—the high-level component structure, their responsibilities, and the flow of control and data between them. Before diving into the intricate details of each subsystem, we establish the mental model of how the entire framework fits together as a cohesive unit capable of transforming vague user requests into concrete, executed actions.

At its core, the framework must bridge the **semantic gap** between the natural language reasoning of an LLM and the structured, deterministic world of software tools. It does this by constructing an **AI Executive**—an entity that can receive a high-level goal, formulate a plan, delegate work to specialized capabilities (tools), track progress, adapt to failures, and ultimately deliver a result. This executive is not monolithic but composed of five interacting pillars, each with a clear separation of concerns.

Component Overview

Think of the framework as a modern corporate office building. Each floor houses a specialized department with its own expertise and workflows, yet they all work together under shared leadership to accomplish complex projects. The blueprints for this building are captured in the system component diagram below.

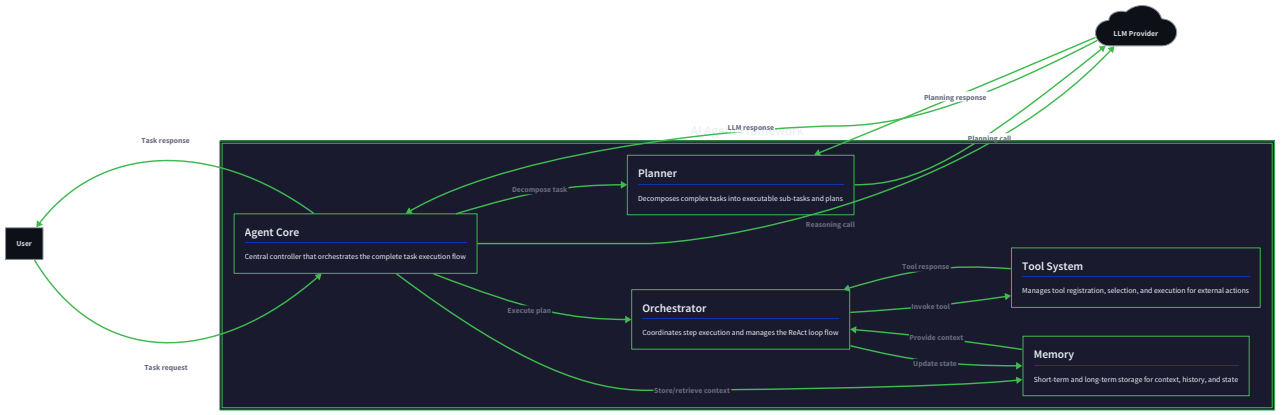


Table 3.1: The Five Pillars of the AI Executive

Component	Mental Model	Primary Responsibility	Key Data It Owns
Agent Core	The CEO's Office	The central coordination point and public API. Manages the agent's overall state, orchestrates the flow between Planner, ReAct Loop, and Memory, and serves as the entry point for task execution.	`AgentState` enum, current `Task` object, configuration settings (LLM client, timeouts, limits).
Tool System	The Workshop & Tool Shed	Provides the agent's "hands" — a dynamic registry of executable capabilities. Handles tool discovery, parameter validation, safe execution, and result formatting.	Registry of all registered `Tool` instances, execution history, tool configuration (timeouts, permissions).
Planner	The Project Management Office (PMO)	Breaks down ambiguous, high-level goals into a concrete sequence (or graph) of actionable steps. Creates, manages, and optionally revises the execution plan.	The `Plan` (a DAG of `SubTask` objects), dependency graph, plan execution status.
Memory System	The Corporate Library & Whiteboards	Provides the agent with context. Manages short-term conversation history, the working state of the current task, and long-term knowledge from past interactions.	`ConversationMemory` buffer, `WorkingMemory` key-value store, `VectorStore` for semantic retrieval.
Orchestrator (Multi-Agent)	The Corporate Headquarters	In multi-agent scenarios, this component manages a team of specialized agents. It handles role assignment, task delegation, inter-agent communication, and result aggregation.	Registry of `Agent` instances with their roles, `Message` queue/bus, delegation rules.

The Agent Core: The Conductor of the Orchestra The Agent Core is the central nervous system. It exposes the primary `Agent.run_task(user_query: str) -> str` method that users call. Internally, it does not perform the reasoning or tool execution itself. Instead, it acts as a conductor:

1. **Receives the user's goal** and initializes a new `Task`.
2. **Invokes the Planner** to decompose the goal into a `Plan` (a list or DAG of `SubTask` objects).
3. **For each `SubTask`**, it triggers the **ReAct Loop Engine** to execute it (think -> act -> observe).
4. **Throughout this process**, it consults the **Memory System** to retrieve relevant context and store new observations.
5. **Manages the agent's lifecycle state** (`AgentState` : IDLE, PLANNING, EXECUTING, etc.), handling errors and transitioning between phases.

The Tool System: The Agent's Swiss Army Knife Tools are how the agent interacts with the external world. Each tool is a well-defined interface with a name, description, parameter schema (using JSON Schema), and an `execute` function. The Tool System's registry pattern allows capabilities to be added at runtime—imagine plugging a new attachment into a multi-tool. The system ensures safety via parameter validation, execution timeouts, and sandboxing where necessary (e.g., for code execution). Built-in tools provide foundational capabilities like web search, calculation, and file operations.

The Planner: The Strategic Mapmaker When a user says "Research the market and write a summary report," the Planner's job is to translate this into a sequence like: 1) Web search for "current market trends Q3 2024," 2) Extract key points from search results, 3) Draft report outline, 4) Write introduction, etc. It models these steps as a **Directed Acyclic Graph (DAG)**, where some steps depend on the outputs of others. This allows for parallel execution of independent tasks and structured handling of failures.

The Memory System: Context as a Layered Cake An agent without memory is amnesic—it forgets the conversation history and past learnings after each interaction. Our Memory System has three distinct layers:

- **Episodic/Conversation Memory:** A simple, in-order buffer of the dialogue (user queries, agent thoughts, tool outputs). This is the agent's "short-term recall" of the current session.
- **Working Memory:** A scratchpad for the *current* task. It holds intermediate results, partial conclusions, and the state of the execution plan. Think of it as the whiteboard in a meeting room.

- **Long-Term Memory:** A vector database that stores embeddings of past interactions, facts, and outcomes. It allows the agent to perform semantic search ("What did we learn about API X last week?") and inject relevant past knowledge into new tasks, enabling learning across sessions.

The Orchestrator: The Manager of Specialists For complex tasks that benefit from specialization, the framework supports multiple agents. The Orchestrator assigns roles (e.g., Researcher, Writer, Coder) and coordinates them. It receives a high-level task, decomposes it, and routes subtasks to the agent whose advertised capabilities best match the requirement. It then manages the conversation between agents, collects their outputs, and synthesizes a final result. This enables a single "virtual" executive to leverage a team of specialized sub-agents.

> **Key Architectural Insight:** The separation between the **Planner** (which decides *what* to do) and the **ReAct Loop Engine** (which figures out *how* to do each subtask) is critical. This mirrors human problem-solving: we first make a high-level plan ("Go to the store, buy ingredients, then cook"), and then for each step, we engage in moment-to-moment reasoning and action ("To go to the store, I need my keys, then open the door, then start the car...").

Interaction Flow

To understand how these components interact dynamically, let's trace the lifecycle of a single user query through a **single-agent** scenario: `"What was the weather in Tokyo last Tuesday, and what's the current price of Bitcoin in JPY?"`

Table 3.2: End-to-End Task Execution Flow

Step	Component Involved	Action & Data Transformation
1. Initiation	User → Agent Core	The user calls `agent.run_task("Weather in Tokyo last Tues & BTC price in JPY")`. The Agent Core creates a new `Task` object, assigns it a unique ID, sets the initial `AgentState` to `PLANNING`, and stores the original query.
2. Context Enrichment	Agent Core → Memory System	Before planning, the Agent Core asks the Memory System: "Given this query, what relevant past context should we include?" The Memory System queries its vector store (long-term memory) for semantically similar past tasks (e.g., previous weather or crypto queries) and returns a few relevant memory snippets.
3. Task Decomposition	Agent Core → Planner	The Agent Core passes the enriched query and context to the Planner. The Planner uses an LLM call with a specialized prompt to break the compound question into atomic subtasks. It outputs a `Plan` containing two independent `SubTask` objects: 1. `SubTask A`: Description="Get historical weather for Tokyo on [last Tuesday's date]", tool="web_search". 2. `SubTask B`: Description="Get current Bitcoin to Japanese Yen exchange rate", tool="web_search". The Planner notes no dependencies between them, allowing parallel execution.
4. Plan Execution (Loop)	Agent Core → ReAct Loop Engine	The Agent Core's state changes to `EXECUTING`. It identifies that Subtask A and B are independent and can be executed concurrently. It calls `_execute_subtask(subtask)` for each.
4a. ReAct for Subtask A	ReAct Loop Engine → Tool System → LLM	For Subtask A, the ReAct Engine begins its cycle: 1. **THINK:** LLM is prompted with the subtask description, available tools, and context. It reasons: "I need to find last Tuesday's date, then search for weather." 2. **ACT:** LLM outputs an action: `{"tool": "calendar", "args": {"operation": "compute_date", "offset": -7}}`. The parser validates and extracts this. 3. **OBSERVE:** Tool System executes the `calendar` tool, returns: "Last Tuesday was 2024-04-09." This observation is fed back. 4. **THINK (again):** LLM reasons: "Now I have the date, I can search." 5. **ACT:** Outputs: `{"tool": "web_search", "args": {"query": "Tokyo weather April 9 2024"}}`. 6. **OBSERVE:** Tool executes, returns a summary of weather data. The ReAct Loop detects this is a final answer for the subtask, packages it into a `ToolResult`, and returns.
4b. Parallel Execution	Agent Core	While Subtask A is in its ReAct cycle, Subtask B begins its own, independent ReAct process, potentially using a different tool (like a financial API).
5. Result Synthesis	Agent Core → Memory System	Both subtasks complete. The Agent Core aggregates their `ToolResult` contents. It then invokes the Memory System to store the entire interaction: the original query, the generated plan, tool calls, and final results are appended to the conversation memory and embedded into the long-term vector store for future retrieval.
6. Final Response	Agent Core → User	The Agent Core formats the aggregated results into a coherent natural language answer: "Last Tuesday (April 9), Tokyo had sunny skies with a high of 18°C. Currently, 1 Bitcoin is worth approximately 8,450,000 Japanese Yen." It updates its `AgentState` to `FINISHED` and returns the final string to the user.

> **Design Principle: Structured Concurrency.** The Agent Core manages subtasks as concurrent units of work. Independent subtasks run in parallel (using Python's `asyncio`) to minimize latency, while dependent subtasks are sequenced according to their DAG. This makes efficient use of resources during I/O-bound operations like LLM calls and web requests.

The flow for a **multi-agent** scenario adds an extra layer of delegation. The Orchestrator acts as a meta-agent: it receives the task, performs its own decomposition, and then routes each subtask to the most suitable specialized agent (e.g., sending a research subtask to a `ResearcherAgent` and a writing subtask to a `WriterAgent`). Each of those agents then runs its own internal Agent Core flow (Plan → ReAct). The Orchestrator collects their results and synthesizes the final output.

Recommended File/Module Structure

A well-organized codebase is paramount for maintainability and extensibility. The following Python package structure mirrors the component architecture, separating concerns and defining clear import boundaries. The `src/` layout is recommended for modern Python projects to avoid import confusion.

```
project_archon/                    # Project root
├── pyproject.toml                 # Build, dependencies, tooling config (Poetry/Pipenv)
├── README.md
├── requirements.txt               # Optional: for pip
│
├── src/                           # Main source package
│   └── archon/                    # Framework's top-level package
│       ├── __init__.py            # Public API exports
│       ├── agent/                 # Agent Core pillar
│       │   ├── __init__.py
│       │   ├── core.py            # Main Agent class with run_task()
│       │   ├── state.py           # AgentState enum, state machine logic
│       │   └── orchestrator.py    # Multi-agent Orchestrator (Milestone 5)
│       │
│       ├── tools/                 # Tool System pillar (Milestone 1)
│       │   ├── __init__.py
│       │   ├── base.py            # ToolBase class, ToolResult
│       │   ├── registry.py        # ToolRegistry class
│       │   ├── engine.py          # Tool execution engine with validation & safety
│       │   └── builtins/          # Built-in tools
│       │       ├── __init__.py
│       │       ├── web_search.py
│       │       ├── calculator.py
│       │       └── code_executor.py
│       │
│       ├── planning/              # Planner pillar (Milestone 3)
│       │   ├── __init__.py
│       │   ├── planner.py         # Base Planner, LLMPlanner
│       │   ├── plan.py            # Task, SubTask, SubTaskStatus, Plan (DAG) classes
│       │   └── executor.py        # Plan execution engine (topological sort, parallel)
│       │
│       ├── memory/                # Memory System pillar (Milestone 4)
│       │   ├── __init__.py
│       │   ├── base.py            # Base memory interfaces
│       │   ├── conversation.py    # Episodic/conversation memory buffer
│       │   ├── working.py         # Working memory (key-value for task state)
│       │   ├── long_term/         # Long-term memory with vector store
│       │   │   ├── __init__.py
│       │   │   ├── vector_store.py  # Abstract interface & implementations (Chroma, FAISS)
│       │   │   └── retriever.py   # Semantic retriever logic
│       │   └── summarizer.py      # Context window summarization/compression
│       │
│       ├── engine/                # ReAct Loop Engine pillar (Milestone 2)
│       │   ├── __init__.py
│       │   ├── react.py           # ReActAgent class, loop controller
│       │   ├── parser.py          # Action parsing strategies (regex, JSON)
│       │   └── prompts.py         # ReAct and planning prompt templates
│       │
│       ├── llm/                   # LLM abstraction layer
│       │   ├── __init__.py
│       │   ├── client.py          # Base LLMClient, adapters for OpenAI, Anthropic, etc.
│       │   └── schemas.py         # Common message/response structures
│       │
│       ├── messaging/             # Multi-agent communication (Milestone 5)
│       │   ├── __init__.py
│       │   ├── message.py         # Message class, message bus interface
│       │   └── bus.py             # In-memory or Redis-based message bus
│       │
│       └── utils/                 # Shared utilities
│           ├── __init__.py
│           ├── logging.py         # Structured JSON logging
│           └── validation.py      # Shared validation helpers
│
├── tests/                         # Comprehensive test suite
│   ├── __init__.py
│   ├── conftest.py
│   ├── unit/
│   │   ├── test_tools.py
│   │   ├── test_react_engine.py
│   │   └── ...
│   ├── integration/
│   │   ├── test_agent_workflow.py
│   │   └── ...
│   └── e2e/
│       └── test_full_agent.py
│
└── examples/                      Example scripts and notebooks
    ├── single_agent_demo.py
```

```
    ├── multi_agent_chat.py
    └── custom_tool_integration.ipynb
```

Key Structural Decisions:

1. **Pillar-Centric Modules (`agent/`, `tools/`, `planning/`, etc.)**: Each major component lives in its own directory, encapsulating its related classes and functions. This makes the codebase navigable and aligns with the architectural diagram.
2. **Clear Public API via `archon/__init__.py`**: This top-level file should expose the key classes users need to get started, e.g., `from archon import Agent, Tool, ToolRegistry`. Implementation details are kept internal.
3. **Separate `llm/` Layer**: Abstracting the LLM provider behind a common `LLMClient` interface ensures the framework is not locked into a single vendor and makes testing with mock LLMs straightforward.
4. **Dedicated `tests/` Hierarchy**: Mirroring the `src/` structure with unit, integration, and end-to-end tests supports the testing pyramid strategy outlined later in the document.

This structure provides a solid foundation that can scale as new tools, memory backends, or planning strategies are developed. Contributors know exactly where to add new functionality, and the separation of concerns minimizes unintended coupling between components.

---

4. Data Model

> **Milestone(s):** 1, 2, 3, 4, 5

The data model forms the backbone of our framework—it's the DNA that defines how information flows, how state is tracked, and how components communicate. Without a well-designed data model, even the most sophisticated algorithms become brittle spaghetti code. This section defines the core types, their fields, and their relationships, providing the structural foundation for all five milestones.

Think of the data model as **the standardized paperwork of our AI Executive's office**. Just as a well-run company uses consistent forms for purchase orders (with fields for item, quantity, vendor) and employee records (with name, role, contact info), our framework needs standardized data structures for tools, tasks, memories, and agent states. This consistency ensures that when the Planner hands off a task to the ReAct Engine, both understand exactly what "task status" means; when the Memory System retrieves a past conversation, it returns it in a format the Agent Core can immediately use.

Core Types and Enums

Below are the foundational data types that appear throughout the framework. Each is presented as a table showing all fields, their types, and detailed descriptions. These types are **immutable by design**—once created, their fields shouldn't be modified directly. Instead, we create new instances with updated values (functional updates) to maintain predictable state transitions and avoid subtle bugs from shared mutable state.

Agent State Enum

Name	Type	Description
`AgentState`	`Enum`	The high-level operational state of an agent instance. This is crucial for debugging, monitoring, and controlling agent execution.
`IDLE`	Enum value	Agent is initialized but not currently processing any task. This is the starting state after agent creation.
`PLANNING`	Enum value	Agent is decomposing a high-level goal into a plan (Task with SubTasks). The LLM is generating the initial breakdown.
`EXECUTING`	`Enum` value	Agent is actively running the ReAct loop on one or more subtasks. This is the main "working" state.
`REPLANNING`	`Enum` value	A subtask has failed or new information requires the agent to adjust its plan. The LLM is generating a revised plan.
`FINISHED`	`Enum` value	All subtasks are completed (or the agent determined the goal is satisfied). The final answer is ready.
`ERROR`	`Enum` value	An unrecoverable system error occurred (e.g., LLM API failure, critical tool crash). Execution is halted.

> **Design Insight**: We intentionally separate `EXECUTING` from `PLANNING` and `REPLANNING` because these are fundamentally different cognitive modes for the LLM. During planning, the LLM thinks about structure and dependencies; during execution, it focuses on concrete tool usage. Tracking this distinction helps with debugging—if an agent is stuck in `EXECUTING` for too long, we know the issue is with tool execution, not plan generation.

Tool Result Types

Name	Type	Description
`ToolResultStatus`	`Enum`	The outcome category of a tool execution. This drives error recovery logic— `ERROR` might trigger retry, while `TIMEOUT` might skip the tool entirely.
`SUCCESS`	Enum value	Tool executed without errors and produced valid output.
`ERROR`	Enum value	Tool execution failed due to a business logic error (e.g., "divide by zero" for calculator, "website not found" for web search).
`TIMEOUT`	Enum value	Tool execution exceeded its allotted time limit and was forcibly terminated.
`ToolResult`	`DataClass`	The complete output from executing a tool, including status, content, and metadata. This is what gets passed back to the LLM as an observation.
`content`	`str`	The primary output text from the tool, formatted for LLM consumption. For a web search tool, this would be the summarized results; for a calculator, the numeric answer. Always present, even for errors (contains error description).
`status`	`ToolResultStatus`	Categorical outcome (SUCCESS, ERROR, TIMEOUT).
`error_message`	`Optional[str]`	Detailed error description if status is ERROR or TIMEOUT. For SUCCESS, this is `None`.
`metadata`	`Dict[str, Any]`	Additional structured data from the tool execution that isn't part of the LLM-visible content. Examples: raw API response, execution duration, token usage, confidence scores. Used for logging, analytics, and advanced recovery logic.

> **Design Insight**: The `ToolResult` deliberately separates `content` (for the LLM) from `metadata` (for the system). This prevents the LLM from being overwhelmed with technical details while preserving debug information. The metadata field uses `Dict[str, Any]` for extensibility—different tools can include tool-specific metadata without changing the core type.

Task and SubTask Types

Name	Type	Description
`SubTaskStatus`	`Enum`	The lifecycle state of an individual subtask within a larger plan. This enables dependency tracking—a subtask can only execute when all its dependencies are `COMPLETED`.
`PENDING`	Enum value	Subtask has been created but hasn't started execution. All dependencies are satisfied, but it's waiting its turn in the execution queue.
`EXECUTING`	Enum value	Subtask is currently being processed by the ReAct loop. The agent is thinking, acting, and observing for this specific subtask.
`COMPLETED`	Enum value	Subtask finished successfully. Its `result` field contains the `ToolResult`.
`FAILED`	Enum value	Subtask execution failed after all retries. The `result` contains the error details.
`BLOCKED`	Enum value	Subtask cannot start because one or more dependencies are not yet `COMPLETED`. The system will periodically re-evaluate blocked subtasks.
`SubTask`	`DataClass`	A single, atomic unit of work within a larger Task. This is the fundamental building block of agent plans.
`id`	`str`	Unique identifier (UUID recommended). Used for referencing in dependencies and tracking execution.
`description`	`str`	Human-readable description of what this subtask should accomplish. This is what the LLM sees when deciding which tool to use. Example: "Search the web for current temperature in Tokyo."
`expected_output`	`Optional[str]`	The anticipated format or content of the result. Used for validation and for guiding the LLM. Example: "A single temperature value in Celsius."
`tool_name`	`Optional[str]`	The name of the tool to execute (must match a registered tool). For planning-only subtasks (like "decide which tools to use"), this may be `None`.
`tool_args`	`Optional[Dict[str, Any]]`	The arguments to pass to the tool, validated against the tool's JSON Schema. Must be `None` if `tool_name` is `None`.
`dependencies`	`List[str]`	List of `SubTask.id` values that must complete before this subtask can start. Empty list means no dependencies.
`status`	`SubTaskStatus`	Current lifecycle state. Initially `PENDING` or `BLOCKED` (if dependencies exist).
`result`	`Optional[ToolResult]`	The output from executing this subtask. `None` until the subtask completes (successfully or otherwise).
`Task`	`DataClass`	A complete unit of work representing a user's original query. Contains the goal, all subtasks, and execution context.
`id`	`str`	Unique identifier for the task (UUID recommended). Used for logging and correlating related activities.
`original_query`	`str`	The exact text the user submitted. Preserved for reference and for inclusion in memory.
`goal`	`str`	The refined, actionable objective derived from the original query. May be rephrased by the planner for clarity. Example: Original query "What's the weather?" becomes goal "Determine current temperature and conditions in the user's location."
`subtasks`	`List[SubTask]`	The decomposed steps to achieve the goal, in dependency order. This is the plan created by the Planner component.
`context`	`Dict[str, Any]`	Arbitrary key-value data shared across all subtasks. Used to pass intermediate results between subtasks. Example: After subtask "Get user location" completes, it stores `{"location": "Tokyo"}` in context, which subtask "Get Tokyo weather" can read.

> **Design Insight**: The separation between `Task` (the overall objective) and `SubTask` (individual steps) mirrors the distinction between a project charter and individual tickets in a project management system. This separation is crucial for **replanning**—when a subtask fails, we can regenerate just the remaining subtasks while preserving the original `Task.id` and `original_query` for continuity.

Memory Types

Name	Type	Description
`MemoryEntry`	`DataClass`	A single unit of stored memory, whether in short-term conversation history or long-term vector store.
`id`	`str`	Unique identifier (UUID recommended).
`content`	`str`	The textual content to remember. For conversation history, this is the exact message; for long-term facts, it's a summarized statement.
`timestamp`	`datetime`	When this memory was created or last accessed. Used for recency-based retrieval.
`metadata`	`Dict[str, Any]`	Additional context: `{"type": "user_message"}`, `{"tool_used": "web_search"}`, `{"importance_score": 0.8}`, etc.
`embedding`	`Optional[List[float]]`	Vector representation of the content for semantic search. `None` for entries in non-vector stores (like simple conversation buffer).
`Message`	`DataClass`	A structured communication between agents in a multi-agent system. This is the envelope that carries requests, responses, and notifications.
`sender_id`	`str`	Identifier of the sending agent.
`receiver_id`	`str`	Identifier of the intended recipient agent. Use "broadcast" for messages to all agents.
`message_type`	`str`	Category: "task_request", "task_response", "notification", "error".
`content`	`Dict[str, Any]`	The payload. Structure varies by type: for "task_request", contains `{"task": Task, "deadline": ...}`; for "task_response", contains `{"result": ToolResult}`.
`timestamp`	`datetime`	When the message was sent.
`correlation_id`	`Optional[str]`	For linking requests to responses (like a session ID).

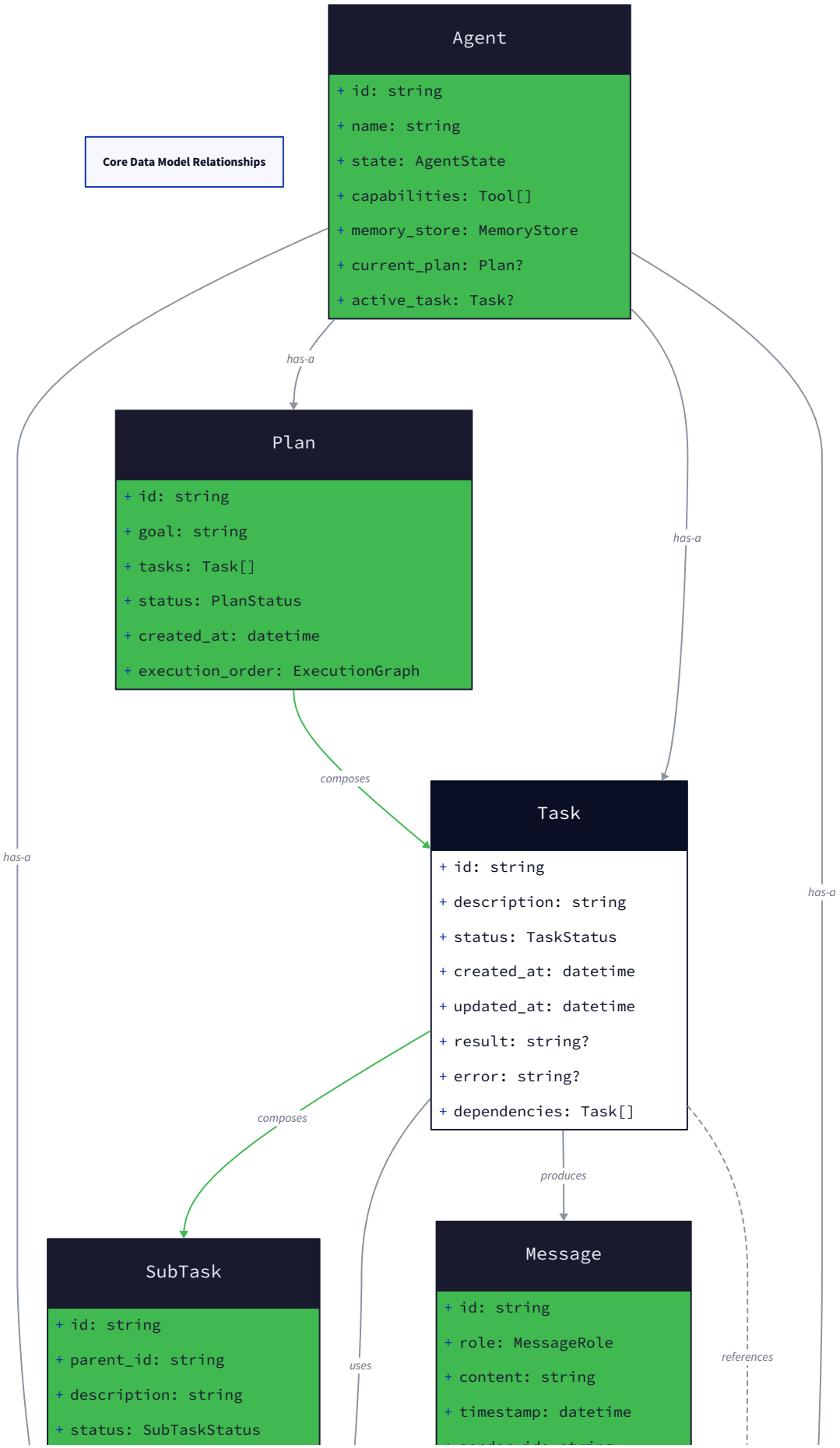
> **Design Insight**: The `MemoryEntry` uses an optional `embedding` field rather than requiring it for all entries because different storage backends have different needs. The conversation buffer (short-term memory) doesn't need embeddings since it just stores sequential messages; the vector store (long-term memory) requires them for similarity search. Making it optional avoids unnecessary computation for non-vector memories.

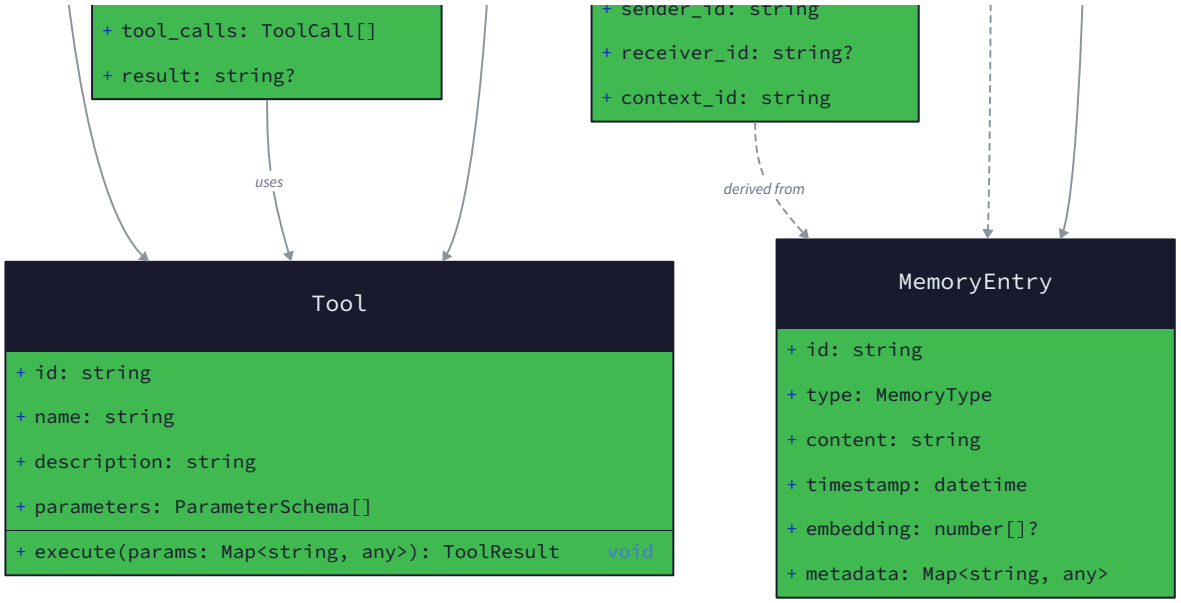
Additional Supporting Types

Name	Type	Description
`Tool`	`Interface`	The abstract definition of a capability available to the agent. Not a data class but a behavioral contract.
`name`	`str` (property)	Unique, descriptive name (e.g., "web_search", "calculator"). Used by the LLM to select the tool.
`description`	`str` (property)	Natural language explanation of what the tool does and when to use it. Critical for LLM tool selection.
`parameters_schema`	`Dict[str, Any]` (property)	JSON Schema defining the expected arguments. Example: `{"type": "object", "properties": {"query": {"type": "string"}}}`.
`execute`	`Callable[[Dict[str, Any]], ToolResult]` (method)	The function that runs when the tool is invoked. Takes validated arguments, returns `ToolResult`.
`Plan`	`DataClass`	Alternative representation of a Task focused on execution. Essentially a Task viewed through the lens of its dependency graph.
`task_id`	`str`	References the parent `Task.id`.
`subtask_dag`	`Dict[str, List[str]]`	Adjacency list representation: `{subtask_id: [dependency_id1, dependency_id2, ...]}`.
`execution_order`	`List[str]`	Topologically sorted list of `SubTask.id`s for sequential execution. Computed from the DAG.

Entity Relationships

The core entities don't exist in isolation—they form a web of relationships that define how the framework operates. Understanding these relationships is key to implementing correct data flow between components.


Core Data Model Relationships
Agent
+ id: string
+ name: string
+ state: AgentState
+ capabilities: Tool[]
+ memory_store: MemoryStore
+ current_plan: Plan?
+ active_task: Task?
has-a
has-a
has-a
has-a
Plan
+ id: string
+ goal: string
+ tasks: Task[]
+ status: PlanStatus
+ created_at: datetime
+ execution_order: ExecutionGraph
composes
Task
+ id: string
+ description: string
+ status: TaskStatus
+ created_at: datetime
+ updated_at: datetime
+ result: string?
+ error: string?
+ dependencies: Task[]
composes
produces
uses
references
SubTask
+ id: string
+ parent_id: string
+ description: string
+ status: SubTaskStatus
Message
+ id: string
+ role: MessageRole
+ content: string
+ timestamp: datetime



Primary Relationships

1. **Agent ⇄ State ⇄ Task**
   - **Relationship**: An **Agent** has a current `AgentState` and executes exactly one **Task** at a time (though it may queue multiple tasks).
   - **Data flow**: When a user submits a query via `Agent.run_task()`, the agent transitions from `IDLE` → `PLANNING` and creates a new `Task` instance. During execution, the agent's state cycles between `EXECUTING` and `REPLANNING` as it works through the task's subtasks. When all subtasks are `COMPLETED`, the agent transitions to `FINISHED`.
2. **Task ⇄ SubTask (Plan)**
   - **Relationship**: A **Task** is composed of multiple **SubTask** instances arranged in a dependency graph (Directed Acyclic Graph).
   - **Structural pattern**: This is a **whole-part relationship** (composition). The Task owns its subtasks—if a Task is deleted, its subtasks are also deleted. The dependency graph is represented implicitly via each subtask's `dependencies` field (which lists other subtask IDs within the same Task).
   - **Execution implication**: The Planner component creates this structure by decomposing the Task's `goal` into subtasks. The Plan Executor then processes subtasks in topological order based on their dependencies.
3. **SubTask ⇄ Tool ⇄ ToolResult**
   - **Relationship**: A **SubTask** invokes a **Tool** (via its `tool_name`), which produces a **ToolResult** stored in the subtask's `result` field.
   - **Linking mechanism**: The `SubTask.tool_name` must match the `Tool.name` of a registered tool. The `SubTask.tool_args` must conform to the tool's `parameters_schema`. After execution, the `ToolResult` is attached to the subtask, completing its lifecycle.
   - **Error handling**: If the tool returns `ToolResultStatus.ERROR`, the subtask's status becomes `FAILED`, which may trigger replanning at the Task level.
4. **Agent ⇄ Memory System ⇄ MemoryEntry**
   - **Relationship**: An **Agent** has access to a **Memory System** which stores and retrieves **MemoryEntry** instances.
   - **Storage layers**: The memory system internally manages different storage backends:
     - **Conversation Memory**: A sequential buffer of `MemoryEntry` instances representing the current session's dialogue (no embeddings needed).
     - **Working Memory**: A key-value store (implemented as part of `Task.context`) for the current task's intermediate state.
     - **Long-term Memory**: A vector store of `MemoryEntry` instances with computed `embedding` fields for semantic retrieval.
   - **Retrieval flow**: During each ReAct loop iteration, the agent queries the memory system for relevant past experiences, which are injected into the LLM prompt as context.
5. **Agent ⇄ Message ⇄ Agent (Multi-Agent)**
   - **Relationship**: In a multi-agent system, **Agent** instances communicate by sending and receiving **Message** objects through a shared communication channel.
   - **Orchestration patterns**:
     - **Hierarchical**: An orchestrator agent sends `Message` objects with `message_type="task_request"` to worker agents, who respond with `message_type="task_response"`.
     - **Broadcast**: An agent sends a message with `receiver_id="broadcast"` that all agents receive.
   - **Correlation**: The `correlation_id` field allows linking responses to specific requests, enabling complex conversational patterns between agents.

State Lifecycle Examples

SubTask Status Transitions:

```
PENDING → EXECUTING (when all dependencies are COMPLETED and executor picks it up)
EXECUTING → COMPLETED (tool execution succeeds with ToolResultStatus.SUCCESS)
EXECUTING → FAILED (tool execution fails after retries)
PENDING → BLOCKED (system detects unresolved dependencies)
BLOCKED → PENDING (when all dependencies become COMPLETED)
```

Task Execution Flow:

1. User calls `Agent.run_task("Research AI ethics and write a summary")`
2. Agent creates `Task` with `id="task_123"`, `original_query="Research AI ethics..."`, `goal="Comprehensive AI ethics summary"`
3. Planner decomposes into subtasks:
   - `SubTask(id="st1", description="Search for recent AI ethics papers", tool_name="web_search", dependencies=[])`
   - `SubTask(id="st2", description="Extract key arguments from search results", tool_name="text_analyzer", dependencies=["st1"])`
   - `SubTask(id="st3", description="Write 500-word summary", tool_name="text_writer", dependencies=["st2"])`
4. Executor processes subtasks in order: st1 → st2 → st3
5. After st3 completes, Task is marked as finished, final answer compiled from all subtask results

Data Model Design Principles

> **Principle 1: Immutability Where Possible** All core data classes (`ToolResult` , `SubTask` , `Task` , `MemoryEntry` , `Message`) should be treated as immutable records. When state needs to change (e.g., subtask status updates), create a new instance with the updated field. This prevents race conditions in concurrent execution and makes debugging easier—you can log the entire history of state changes.

> **Principle 2: Optional Fields for Progressive Enhancement** Many fields are `Optional` (like `SubTask.expected_output` , `MemoryEntry.embedding`) because they're only relevant in certain contexts or advanced configurations. This keeps the core data model simple while allowing extensibility.

> **Principle 3: String IDs for Loose Coupling** All relationships use string IDs (`SubTask.dependencies` lists IDs, not object references). This allows:
>
> - Serialization to JSON for storage/transmission
> - Loose coupling between components (Planner doesn't need actual SubTask objects, just their IDs)
> - Easy debugging (IDs appear in logs) The trade-off is an extra lookup step to resolve IDs to objects, but this is minimal compared to the benefits.

> **Principle 4: Metadata Bags for Extensibility** The `metadata` fields in `ToolResult` and `MemoryEntry` follow the "bag of properties" pattern. Instead of adding specific fields for every possible piece of auxiliary data (which would cause constant type changes), we use a dictionary. Different components can agree on conventions for metadata keys (e.g., `"token_usage"` , `"confidence_score"` , `"source_url"`).

Common Pitfalls in Data Modeling

⚠️ Pitfall: Mutating Shared Data Structures

- **Description**: Directly modifying a `SubTask` object's `status` field while multiple components hold references to it.
- **Why it's wrong**: In concurrent execution (parallel subtasks), this causes race conditions. Even in sequential execution, it makes debugging hard—you can't trace when the status changed.
- **Fix**: Use functional updates: `new_subtask = subtask.with_status(SubTaskStatus.EXECUTING)` . Implement `.copy()` or `.replace()` methods that return new instances.

⚠️ Pitfall: Circular Dependencies in SubTask Graph

- **Description**: Creating a `SubTask` where `dependencies` includes an ID that eventually depends back on this subtask (A depends on B depends on A).
- **Why it's wrong**: The executor gets stuck in infinite loop trying to find a subtask with no unsatisfied dependencies.
- **Fix**: Validate the dependency graph is acyclic when building the plan. Use topological sort detection during planning phase.

⚠️ Pitfall: Inconsistent ID Referencing

- **Description**: A `SubTask` lists a dependency ID that doesn't exist in the parent `Task.subtasks` .
- **Why it's wrong**: The executor will mark the subtask as `BLOCKED` forever, waiting for a non-existent dependency to complete.
- **Fix**: Validate all referenced IDs exist within the same Task when constructing the plan. Maintain referential integrity checks.

⚠️ Pitfall: Missing Metadata in ToolResult

- **Description**: Tools returning `ToolResult` with empty `metadata` dict, losing valuable debugging information.
- **Why it's wrong**: When a tool fails mysteriously, there's no forensic data to diagnose the issue (execution time, intermediate values, error codes).
- **Fix**: Establish conventions for minimum metadata: `{"execution_ms": 150, "tool_version": "1.0"}`. Better: include stack traces for errors.

⚠️ **Pitfall: Overloading the Context Dictionary**

- **Description**: Storing too much unstructured data in `Task.context`, making it a "global variable bag" that's hard to reason about.
- **Why it's wrong**: Subtasks become tightly coupled through implicit data dependencies (subtask B expects subtask A to have set `context["formatted_data"]`).
- **Fix**: Document context key conventions. Consider typed context objects or a schema for the context dictionary.

Implementation Guidance

Technology Recommendations Table

Component	Simple Option	Advanced Option
Data Classes	Python `dataclasses` with `@dataclass`	Pydantic `BaseModel` with validation
Enum Implementation	Python `Enum` class	`StrEnum` (Python 3.11+) for string values
ID Generation	`uuid.uuid4().hex`	Time-ordered UUIDs (ULID) for sortable IDs
JSON Schema Validation	`jsonschema` library	Pydantic models with automatic JSON Schema generation
Vector Storage	In-memory FAISS index	ChromaDB or Pinecone for persistence

Recommended File/Module Structure

```
project_archon/
├── core/
│   ├── __init__.py
│   ├── types.py           # ← All data classes and enums go here
│   ├── agent.py           # Agent class definition
│   └── interfaces.py      # Abstract base classes
├── tools/
│   ├── base.py            # Tool interface
│   ├── registry.py        # ToolRegistry class
│   └── builtin/           # Built-in tool implementations
├── planning/
│   ├── planner.py         # Planner interface
│   ├── task.py            # Task/SubTask execution logic
│   └── dag.py             # DAG utilities
├── memory/
│   ├── base.py            # Memory interfaces
│   ├── conversation.py    # Conversation memory
│   └── vector.py          # Vector store implementation
└── multi_agent/
    ├── message.py         # Message class
    ├── orchestrator.py    # Orchestrator implementation
    └── roles.py           # Role definitions
```

Infrastructure Starter Code

Complete `types.py` with core data classes:

PYTHON

```python
"""Core data types for the AI Agent Framework."""

from dataclasses import dataclass, field, replace

from datetime import datetime

from enum import Enum

from typing import Any, Dict, List, Optional

import uuid


class ToolResultStatus(Enum):

    """Status of a tool execution."""

    SUCCESS = "success"

    ERROR = "error"

    TIMEOUT = "timeout"


@dataclass(frozen=True)

class ToolResult:

    """Result from executing a tool."""

    content: str

    status: ToolResultStatus

    error_message: Optional[str] = None

    metadata: Dict[str, Any] = field(default_factory=dict)


    @classmethod

    def success(cls, content: str, metadata: Optional[Dict[str, Any]] = None) -> "ToolResult":

        """Create a successful tool result."""

        return cls(

            content=content,

            status=ToolResultStatus.SUCCESS,

            metadata=metadata or {}

        )


    @classmethod

    def error(cls, error_msg: str, metadata: Optional[Dict[str, Any]] = None) -> "ToolResult":

        """Create an error tool result."""

        return cls(

            content=f"Error: {error_msg}",

            status=ToolResultStatus.ERROR,

            error_message=error_msg,

            metadata=metadata or {}

        )
```

```python
    @classmethod
    def timeout(cls, timeout_seconds: float, metadata: Optional[Dict[str, Any]] = None) -> "ToolResult":
        """Create a timeout tool result."""
        return cls(
            content=f"Tool execution timed out after {timeout_seconds} seconds",
            status=ToolResultStatus.TIMEOUT,
            error_message="Execution timeout",
            metadata=metadata or {}
        )


class SubTaskStatus(Enum):
    """Lifecycle status of a subtask."""
    PENDING = "pending"
    EXECUTING = "executing"
    COMPLETED = "completed"
    FAILED = "failed"
    BLOCKED = "blocked"


@dataclass
class SubTask:
    """A single atomic unit of work within a task."""
    id: str = field(default_factory=lambda: str(uuid.uuid4()))
    description: str = ""
    expected_output: Optional[str] = None
    tool_name: Optional[str] = None
    tool_args: Optional[Dict[str, Any]] = None
    dependencies: List[str] = field(default_factory=list)
    status: SubTaskStatus = SubTaskStatus.PENDING
    result: Optional[ToolResult] = None

    def with_status(self, new_status: SubTaskStatus) -> "SubTask":
        """Return a new SubTask with updated status (functional update)."""
        return replace(self, status=new_status)

    def with_result(self, result: ToolResult) -> "SubTask":
        """Return a new SubTask with updated result (functional update)."""
        return replace(self, result=result, status=SubTaskStatus.COMPLETED)

    def is_ready(self) -> bool:
        """Check if this subtask can execute (dependencies satisfied)."""
```

```python
        return self.status == SubTaskStatus.PENDING and not self.dependencies


@dataclass
class Task:
    """A complete unit of work representing a user query."""
    id: str = field(default_factory=lambda: str(uuid.uuid4()))
    original_query: str = ""
    goal: str = ""
    subtasks: List[SubTask] = field(default_factory=list)
    context: Dict[str, Any] = field(default_factory=dict)

    def get_subtask(self, subtask_id: str) -> Optional[SubTask]:
        """Retrieve a subtask by ID."""
        for subtask in self.subtasks:
            if subtask.id == subtask_id:
                return subtask
        return None

    def update_subtask(self, updated_subtask: SubTask) -> "Task":
        """Return a new Task with the given subtask updated."""
        new_subtasks = []
        for subtask in self.subtasks:
            if subtask.id == updated_subtask.id:
                new_subtasks.append(updated_subtask)
            else:
                new_subtasks.append(subtask)
        return replace(self, subtasks=new_subtasks)


class AgentState(Enum):
    """High-level operational state of an agent."""
    IDLE = "idle"
    PLANNING = "planning"
    EXECUTING = "executing"
    REPLANNING = "replanning"
    FINISHED = "finished"
    ERROR = "error"


@dataclass
class MemoryEntry:
    """A single unit of stored memory."""
```

```
    id: str = field(default_factory=lambda: str(uuid.uuid4()))
    content: str = ""
    timestamp: datetime = field(default_factory=datetime.now)
    metadata: Dict[str, Any] = field(default_factory=dict)
    embedding: Optional[List[float]] = None


@dataclass
class Message:
    """Structured communication between agents."""
    sender_id: str = ""
    receiver_id: str = ""
    message_type: str = ""
    content: Dict[str, Any] = field(default_factory=dict)
    timestamp: datetime = field(default_factory=datetime.now)
    correlation_id: Optional[str] = None
```

Core Logic Skeleton Code

Agent class with data model integration:

PYTHON

```python
"""Agent core implementation."""

from typing import Optional

from .types import AgentState, Task, SubTask, ToolResult


class Agent:

    """Main agent class coordinating all components."""


    def __init__(self, agent_id: str):

        self.agent_id = agent_id

        self.state = AgentState.IDLE

        self.current_task: Optional[Task] = None

        self.memory = []  # Will be replaced with proper memory system


    def run_task(self, user_query: str) -> str:

        """Primary public method to execute a user's task.


        Args:

            user_query: The natural language request from the user.


        Returns:

            Final answer as a string.

        """

        # TODO 1: Transition state from IDLE to PLANNING

        # TODO 2: Create a new Task instance with the user_query as original_query

        # TODO 3: Call the Planner to decompose the query into subtasks

        # TODO 4: Transition state to EXECUTING

        # TODO 5: Execute each subtask in dependency order using _execute_subtask

        # TODO 6: Collect results from all completed subtasks

        # TODO 7: Compile final answer

        # TODO 8: Transition state to FINISHED and return answer

        # TODO 9: Handle errors by transitioning to ERROR state with appropriate messaging

        pass


    def _execute_subtask(self, subtask: SubTask) -> ToolResult:

        """Internal method to run the ReAct loop for a single subtask.


        Args:

            subtask: The subtask to execute.


        Returns:
```

```
            ToolResult from the execution.
        """
        # TODO 1: Validate subtask has a tool_name and tool_args
        # TODO 2: Look up the tool from the ToolRegistry by tool_name
        # TODO 3: Validate tool_args against the tool's parameter schema
        # TODO 4: Execute the tool with timeout protection
        # TODO 5: Capture the ToolResult
        # TODO 6: Update the subtask with the result (functional update)
        # TODO 7: Store relevant information in memory
        # TODO 8: Return the ToolResult
        pass
```

Language-Specific Hints

- **Python dataclasses**: Use `@dataclass(frozen=True)` for truly immutable types, but note that frozen dataclasses can't have default mutable fields like `List`. Use `field(default_factory=list)` instead of `default=[]`.
- **Functional updates**: The `replace()` function from `dataclasses` creates a new instance with updated fields. This is cleaner than manual `copy()` and update.
- **Enum serialization**: Python Enums serialize to their value (string) when using `json.dumps()`. For `ToolResultStatus.SUCCESS`, JSON becomes `"success"`.
- **Type hints**: Use `from typing import Optional` for nullable fields. Add `-> None` to `__post_init__` methods for proper type checking.
- **ID generation**: `uuid.uuid4().hex` gives a 32-character hex string without hyphens, ideal for clean IDs in logs and UI.

Milestone Checkpoint

After implementing the data model types, verify with:

```bash
# Run the type validation tests
python -m pytest tests/test_types.py -v

# Expected output should show:
# test_tool_result_creation (PASS)
# test_subtask_dependencies (PASS)
# test_task_immutability (PASS)
# test_enum_serialization (PASS)
```

Create a simple test to verify the data model works correctly:

```python
# Quick verification script
from core.types import Task, SubTask, SubTaskStatus

# Create a simple task with subtasks
task = Task(
    original_query="What's 2+2?",
    goal="Calculate 2+2",
    subtasks=[
        SubTask(
            description="Calculate 2+2",
            tool_name="calculator",
            tool_args={"expression": "2+2"}
        )
    ]
)

print(f"Task created: {task.id}")
print(f"First subtask status: {task.subtasks[0].status}")
print(f"Subtask ready?: {task.subtasks[0].is_ready()}")

# Functional update
updated_subtask = task.subtasks[0].with_status(SubTaskStatus.EXECUTING)
print(f"Updated status: {updated_subtask.status}")
```

Expected output should show proper ID generation, correct default status, and functional updates working without modifying the original object.

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
"SubTask has no attribute 'id'"	Forgot to use `field(default_factory=...)` for ID generation	Check if `SubTask()` creation without ID argument fails	Use `field(default_factory=lambda: str(uuid.uuid4()))`
"JSON serialization error on Enum"	Trying to serialize Enum object directly	Print `type(task.subtasks[0].status)`	Ensure Enums have string values and use `.value` when custom serialization needed
"Updated subtask but original also changed"	Mutating dataclass in place instead of functional update	Check for `subtask.status = new_status` assignments	Use `replace()` or `.with_status()` method
"Dependency resolution stuck"	Circular dependency in subtask graph	Print adjacency list and run cycle detection	Validate graph is acyclic during planning

5.1 Component: Tool System (Milestone 1)

> **Milestone(s):** 1

The Tool System is the foundation upon which the AI Executive extends its capabilities into the external world. Just as a CEO cannot personally type every email or analyze every spreadsheet, the AI Executive cannot directly interact with databases, APIs, or calculation engines. Instead, it must delegate these actions to specialized tools. This component defines the contract, registration, and safe execution of these tools, bridging the **semantic gap** between the LLM's natural language reasoning and the structured world of software interfaces.

Mental Model: The Agent's Toolbox

Imagine giving a skilled but disembodied assistant a Swiss Army knife. The assistant understands language and can reason about problems, but it needs physical attachments (blades, scissors, screwdrivers) to manipulate the world. Each attachment has a specific purpose, a safe way to hold it, and clear instructions for use. The **Tool System** is this Swiss Army knife—a curated, well-documented, and safely accessible collection of capabilities.

- **Well-Defined Interfaces:** Each tool has a precise name, a clear description of what it does, and an exact specification for its inputs (like which screwdriver head fits which screw). This eliminates ambiguity for the LLM when choosing which tool to use.
- **Safe Handling:** Tools are designed with guards—input validation, timeouts, and error handling—to prevent the assistant from accidentally cutting itself (crashing the system) or applying the wrong force (corrupting data).
- **Dynamic Attachment:** New tools can be added to the knife's handle at any time, even while the assistant is working, without needing to rebuild the entire knife. The assistant can immediately discover and use these new capabilities.

This mental model emphasizes that tools are not arbitrary code snippets but are **controlled, observable, and predictable extensions** of the agent's will.

Interface and Contract

Every tool in the framework must adhere to a strict interface. This consistency enables the LLM to reason about tool usage, the registry to manage them uniformly, and the execution engine to invoke them safely.

The `Tool` interface defines the following mandatory properties and method:

Property/Method	Type	Description
`name`	`str`	A unique, descriptive identifier for the tool (e.g., `"web_search"`, `"execute_python_code"`). Must be lowercase with underscores. This is the primary key the LLM uses to invoke the tool.
`description`	`str`	A natural language explanation of the tool's purpose, its inputs, and the nature of its output. This is injected into the LLM's prompt and is critical for accurate tool selection. A good description answers: "What does this do? When should I use it? What do I give it, and what do I get back?"
`parameters_schema`	`Dict[str, Any]`	A **JSON Schema** object that rigorously defines the expected parameters. This schema is used for runtime validation before execution. It specifies required fields, data types, allowed values, and constraints (e.g., `"maxLength"` for strings).
`execute(self, **kwargs) -> ToolResult`	Method	The core implementation. Accepts keyword arguments validated against the `parameters_schema`. It must return a `ToolResult` object, which standardizes success/error outcomes and includes metadata for debugging. Execution must be idempotent where possible and must not have hidden side effects beyond those described.

The `ToolResult` type, as defined in the naming conventions, is the universal wrapper for all tool outputs:

Field	Type	Description
`content`	`str`	The primary output of the tool, formatted as text consumable by the LLM (e.g., a search result summary, a calculation answer). For errors, this may contain a user-friendly error message.
`status`	`ToolResultStatus`	An enum (`SUCCESS`, `ERROR`, `TIMEOUT`) indicating the categorical outcome. This allows the agent's loop logic to handle failures differently from successes.
`error_message`	`Optional[str]`	For `ERROR` or `TIMEOUT` statuses, a detailed, technical error message for logging and debugging. This is separate from the LLM-consumable `content`.
`metadata`	`Dict[str, Any]`	A **bag of properties** for extensible, tool-specific information (e.g., `{"latency_ms": 120, "result_count": 5}`). This enables observability and advanced post-processing without polluting the primary `content`.

Factory methods (`ToolResult.success`, `.error`, `.timeout`) should be used to ensure consistent construction of these objects.

Tool Registry Pattern

The Tool Registry acts as the central directory and dispatcher for all available tools. It follows the **Service Locator** pattern, providing a single point of lookup. Its primary responsibilities are dynamic registration, discovery by name, and providing a complete list of tools for prompt construction.

Key Design Points:

1. **Singleton/Global Access:** The registry should be accessible throughout the agent's runtime. This is typically implemented as a class with class-level methods or a module-level global instance, though dependency injection is preferred for testability.
2. **Thread-Safe Registration:** Since tools can be added at runtime (e.g., by plugins), registration operations must be thread-safe to prevent race conditions.
3. **Immutable Tool Definitions:** Once registered, a tool's interface (`name`, `description`, `schema`) should be treated as immutable. The `execute` method's behavior is the tool author's responsibility.

4. **Namespacing:** Consider supporting namespaced tool names (e.g., `"math.calculator"`, `"web.search"`) to avoid collisions in large tool ecosystems, though a simple unique `name` suffices for the initial implementation.

The registry's interface is simple:

Method	Parameters	Returns	Description
`register_tool`	`tool: Tool`	`None`	Adds a tool instance to the registry. Raises a `ValueError` if a tool with the same name already exists.
`get_tool`	`name: str`	`Optional[Tool]`	Retrieves a tool by its unique name. Returns `None` if not found.
`list_tools`	`None`	`List[Tool]`	Returns a list of all registered tools, typically for inclusion in the LLM's system prompt.

Execution Engine with Safety

The Execution Engine is the critical safety layer that sits between the LLM's action choice and the actual tool code. It transforms a raw tool call request ("use `calculator` with `expression: '2+2'`") into a safe, validated, observed, and error-handled execution. This process follows a strict pipeline:

1. **Lookup & Existence Check:** The engine receives a requested tool `name` and `arguments` dict. It first queries the registry. If the tool does not exist, it immediately returns a `ToolResult.error` with a message like "Tool 'foo' not found."
2. **Schema Validation:** Using the tool's `parameters_schema`, the engine validates the provided `arguments`. It checks for required fields, correct data types (e.g., ensuring a parameter expecting a number isn't given a string), and value constraints. Any validation failure results in a descriptive `ToolResult.error` (e.g., "Parameter 'expression' must be a string, got type int").
3. **Safe Execution Wrapping:**
   - **Timeout Enforcement:** The tool's `execute` method is invoked inside a timeout context (e.g., using `asyncio.wait_for` or `threading.Timer`). This prevents buggy or long-running tools from hanging the entire agent loop indefinitely. A timeout triggers a `ToolResult.timeout`.
   - **Exception Isolation:** The call is wrapped in a try-except block. Any uncaught exception from the tool is caught, logged, and converted into a `ToolResult.error`. The `error_message` should include the exception type and traceback for debugging, while the `content` may have a simpler message for the LLM (e.g., "The tool encountered an internal error.").
   - **Sandboxing (Advanced):** For tools that execute code (e.g., Python), the engine should spawn execution in an isolated sandbox (like a restricted subprocess or a `seccomp`-confined environment) with limits on CPU, memory, and filesystem access. This is a **non-goal** for Milestone 1 but is a critical consideration for production.
4. **Result Standardization:** The raw output from `tool.execute` must already be a `ToolResult`. The engine ensures this contract is upheld. It may also attach engine-level metadata to the result's `metadata` field (e.g., `validation_time_ms`, `execution_time_ms`).
5. **Observability Logging:** The engine logs the tool call attempt, validation result, execution duration, and final outcome. This is essential for debugging agent behavior and monitoring tool performance.

> The key insight is that **the execution engine treats all tools as potentially hostile or buggy**. It assumes nothing about the tool's internal reliability and enforces safety and observability at the boundary.

ADR: JSON Schema for Parameter Validation

> **Decision: Use JSON Schema for Tool Parameter Definition and Validation**
>
> - **Context**: The LLM outputs unstructured text that must be parsed into structured arguments for tool execution. We need a robust, declarative way to define the expected structure of these arguments and validate them before execution to prevent runtime errors and improve LLM guidance.
> - **Options Considered**:
>   1. **Unstructured Strings (Free-form)**: The LLM provides arguments as a concatenated string (e.g., `"2 + 2"`), and the tool function parses it internally.
>   2. **Pydantic/Type-Hinted Python Models**: Define tool parameters as Pydantic `BaseModel` classes. Validation uses Python's type system and Pydantic's validators.
>   3. **JSON Schema**: Define parameters as a JSON Schema (Draft 7 or 2020-12) dictionary.
> - **Decision**: Use **JSON Schema**.
> - **Rationale**:
>   1. **Language Agnosticism**: JSON Schema is a universal standard, not tied to Python. This aligns with our potential future TypeScript/JavaScript runtime and makes tool definitions portable.
>   2. **Rich Validation Ecosystem**: JSON Schema supports complex validations (regex patterns, number ranges, array item constraints) that are cumbersome to express purely with Python type hints.
>   3. **LLM-Native Format**: Modern LLMs are extensively trained on JSON and JSON Schema. Providing the schema directly in the prompt (as used by OpenAI's function calling) is a highly effective technique for guiding the LLM to produce correctly formatted arguments.
>   4. **Clear Separation of Concerns**: The schema is a pure declaration. The validation logic (using a library like `jsonschema`) is separate from the tool's business logic, promoting cleaner code.
> - **Consequences**:
>   - **Positive**: Enables high-quality validation and excellent LLM guidance. Tool definitions become declarative and portable.
>   - **Negative**: Adds a dependency on a JSON Schema validation library. Schema definitions are slightly more verbose than Python dataclasses. Developers must learn JSON Schema syntax.

Option	Pros	Cons	Chosen?
Unstructured Strings	Simple to implement; mirrors some early agent frameworks.	Extremely brittle; shifts parsing burden to each tool; no validation before execution; poor LLM guidance.	❌
Pydantic Models	Leverages Python's strong typing; excellent IDE support; validation logic can be embedded in model methods.	Ties the framework to Python; schema extraction for LLM prompts requires extra work; less rich constraint language than JSON Schema.	❌
JSON Schema	Universal standard; rich validation; excellent LLM compatibility; declarative and separate from logic.	Slightly more verbose; requires a validation library dependency.	✅

Common Pitfalls and Mitigations

⚠️ Pitfall: Vague or Incomplete Tool Descriptions

- **Description**: Writing a `description` like "Searches the web" provides insufficient context for the LLM. When should it use this versus a "query_database" tool? What format does the query take? What kind of results are returned?
- **Why it's Wrong**: The LLM will guess incorrectly, leading to wasted iterations and failed tasks. The description is the primary documentation for the LLM.
- **How to Fix**: Write descriptions as if instructing a new employee. Example: "`web_search`: Performs a general internet search using the DuckDuckGo API. Use this to find current information, general knowledge, or news. Input: a `query` string (required). Output: A list of up to 5 search results, each with a `title`, `snippet`, and `url`. Does not access specific databases or internal documents."

⚠️ Pitfall: Missing or Weak Parameter Validation

- **Description**: Relying on the tool's internal code to validate inputs, or defining a schema that's too permissive (e.g., `"type": "string"` for an expression that should be a mathematical formula).
- **Why it's Wrong**: Allows malformed data to reach the tool, causing cryptic runtime exceptions (e.g., `KeyError`, `TypeError`). These are harder for the agent to recover from than a clear validation error.
- **How to Fix**: Use JSON Schema's full power. Define `required` fields, specific `types`, `pattern` regex for strings, `minimum`/`maximum` for numbers, and `enum` for allowed values. The validation error message should be specific enough for the LLM to correct its input.

⚠️ Pitfall: Tool Execution Without Timeout

- **Description**: Calling `tool.execute()` directly without any timeout guard.
- **Why it's Wrong**: A network call that hangs, or a tool with an infinite loop, will freeze the entire agent loop indefinitely. The system becomes unresponsive.
- **How to Fix**: **Always** wrap tool execution in a timeout mechanism. Use `asyncio.wait_for` for async tools or `concurrent.futures.ThreadPoolExecutor` with a timeout for synchronous ones. The timeout should be configurable per-tool.

⚠️ Pitfall: Sensitive Tools Without Permission Checks

- **Description**: Providing tools like `execute_shell_command` or `delete_database_record` without any authorization layer, allowing the LLM to invoke them based purely on its reasoning.
- **Why it's Wrong**: Creates massive security vulnerabilities. The LLM's goal is to satisfy the user's query, which could inadvertently or maliciously lead to destructive actions.
- **How to Fix**: Implement a permission system. Tools can have required permission flags (e.g., `"requires_permission": ["fs_write", "shell_exec"]`). The agent's runtime context should include the user's or session's permissions. The execution engine checks permissions after validation but before execution, rejecting unauthorized calls with a clear error.

Implementation Guidance

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
JSON Schema Validation	`jsonschema` library (pure Python)	`fastjsonschema` (faster, compile-time)
Async Timeout	`asyncio.wait_for`	`anyio` or `trio` for structured concurrency
Sandboxed Code Execution	`subprocess` with resource limits (`resource` module)	Dedicated sandboxing (Docker container, `pysandbox`, `gvisor`)
Tool Registry Storage	In-memory `dict`	Persistent registry (SQLite, Redis) for distributed agents

B. Recommended File/Module Structure:

```
project_archon/
├── archon/
│   ├── __init__.py
│   ├── agent/           # Agent core, ReAct loop (Milestone 2)
│   ├── tools/           # Tool System (THIS MILESTONE)
│   │   ├── __init__.py
│   │   ├── base.py      # ToolBase, ToolResult, ToolRegistry
│   │   ├── engine.py    # ToolExecutionEngine
│   │   ├── builtins/    # Built-in tool implementations
│   │   │   ├── __init__.py
│   │   │   ├── web_search.py
│   │   │   ├── calculator.py
│   │   │   └── code_executor.py
│   │   └── utils.py     # Schema helpers, validation utilities
│   ├── planner/         # (Milestone 3)
│   ├── memory/          # (Milestone 4)
│   └── orchestration/   # (Milestone 5)
└── tests/
    └── tools/
        ├── test_base.py
        ├── test_engine.py
        └── test_builtins/
```

C. Infrastructure Starter Code (Tool Base Classes):

PYTHON

```python
# archon/tools/base.py

import json

from abc import ABC, abstractmethod

from dataclasses import dataclass, field, asdict

from enum import Enum

from typing import Any, Dict, List, Optional, Callable

from datetime import datetime

import jsonschema

from concurrent.futures import TimeoutError


class ToolResultStatus(Enum):

    SUCCESS = "success"

    ERROR = "error"

    TIMEOUT = "timeout"


@dataclass(frozen=True)  # Immutable result

class ToolResult:

    content: str

    status: ToolResultStatus

    error_message: Optional[str] = None

    metadata: Dict[str, Any] = field(default_factory=dict)


    @classmethod

    def success(cls, content: str, metadata: Optional[Dict[str, Any]] = None) -> "ToolResult":

        return cls(

            content=content,

            status=ToolResultStatus.SUCCESS,

            metadata=metadata or {}

        )


    @classmethod

    def error(cls, error_msg: str, metadata: Optional[Dict[str, Any]] = None) -> "ToolResult":

        return cls(

            content=f"Tool execution failed: {error_msg}",

            status=ToolResultStatus.ERROR,

            error_message=error_msg,

            metadata=metadata or {}

        )


    @classmethod

    def timeout(cls, timeout_seconds: float, metadata: Optional[Dict[str, Any]] = None) -> "ToolResult":

        msg = f"Tool execution timed out after {timeout_seconds} seconds"

        return cls(
```

```
            content=msg,
            status=ToolResultStatus.TIMEOUT,
            error_message=msg,
            metadata=metadata or {}
        )

class Tool(ABC):
    """Abstract base class for all tools."""

    @property
    @abstractmethod
    def name(self) -> str:
        """Unique tool identifier (snake_case)."""
        pass

    @property
    @abstractmethod
    def description(self) -> str:
        """Natural language description for the LLM."""
        pass

    @property
    @abstractmethod
    def parameters_schema(self) -> Dict[str, Any]:
        """JSON Schema dict defining the expected parameters."""
        pass

    @abstractmethod
    async def execute(self, **kwargs) -> ToolResult:
        """Execute the tool with validated arguments."""
        pass

    def validate_args(self, arguments: Dict[str, Any]) -> Optional[str]:
        """Validate arguments against schema. Returns error message or None."""
        try:
            jsonschema.validate(instance=arguments, schema=self.parameters_schema)
            return None
        except jsonschema.ValidationError as e:
            return f"Parameter validation error: {e.message} at path '{e.json_path}'"
```

D. Core Logic Skeleton Code (Registry and Execution Engine):

PYTHON

```python
# archon/tools/base.py (continued)

class ToolRegistry:

    """Global registry for tool discovery and lookup."""


    _tools: Dict[str, Tool] = {}


    @classmethod

    def register_tool(cls, tool: Tool) -> None:

        """Register a tool instance. Raises ValueError if name conflict."""

        # TODO 1: Check if tool.name already exists in cls._tools

        # TODO 2: If it exists, raise ValueError(f"Tool '{tool.name}' already registered")

        # TODO 3: Store the tool in cls._tools dictionary with name as key

        pass


    @classmethod

    def get_tool(cls, name: str) -> Optional[Tool]:

        """Retrieve a tool by name. Returns None if not found."""

        # TODO 1: Return the tool from cls._tools dictionary using name as key

        # TODO 2: If key doesn't exist, return None

        pass


    @classmethod

    def list_tools(cls) -> List[Tool]:

        """Return list of all registered tools."""

        # TODO 1: Return all values from the cls._tools dictionary as a list

        pass


# archon/tools/engine.py

import asyncio

import logging

from typing import Dict, Any

from .base import Tool, ToolResult, ToolRegistry


logger = logging.getLogger(__name__)


class ToolExecutionEngine:

    """Orchestrates safe tool lookup, validation, execution, and error handling."""


    def __init__(self, default_timeout_seconds: float = 30.0):

        self.default_timeout = default_timeout_seconds


    async def execute_tool_call(
```

```
        self,
        tool_name: str,
        arguments: Dict[str, Any],
        timeout_seconds: Optional[float] = None
    ) -> ToolResult:
        """
        Main entry point: executes a tool call safely.

        Args:
            tool_name: Name of the tool to execute.
            arguments: Dictionary of arguments for the tool.
            timeout_seconds: Optional override for execution timeout.

        Returns:
            ToolResult representing the outcome.
        """
        # TODO 1: Look up the tool using ToolRegistry.get_tool(tool_name)
        # TODO 2: If tool is None, return ToolResult.error with message "Tool '{tool_name}' not found"

        # TODO 3: Validate arguments using tool.validate_args(arguments)
        # TODO 4: If validation returns an error message, return ToolResult.error with that message

        # TODO 5: Determine actual timeout: use timeout_seconds if provided, else self.default_timeout

        # TODO 6: Log the start of tool execution (tool name, arguments)

        # TODO 7: Wrap tool.execute(**arguments) in asyncio.wait_for with the timeout
        # TODO 8: If wait_for times out (asyncio.TimeoutError), return ToolResult.timeout
        # TODO 9: If tool.execute raises any other Exception, catch it, log it, and return ToolResult.error
        # TODO 10: If execution succeeds, return the ToolResult directly

        # HINT: Use try/except blocks around asyncio.wait_for
        # HINT: Capture start_time = time.time() to calculate duration for metadata
        pass
```

E. Language-Specific Hints:

- **Use `asyncio` for Concurrency:** The framework is async-first. All tool `execute` methods should be `async def`. Use `asyncio.wait_for` for timeouts.
- **JSON Schema Validation with `jsonschema`:** Install via `pip install jsonschema`. Use `jsonschema.validate()` in the `validate_args` helper. Pre-compile schemas for frequently used tools with `jsonschema.Draft7Validator`.
- **Structured Logging:** Use the `logging` module with JSON formatters (e.g., `python-json-logger`) for production. Log tool calls, validation results, execution times, and outcomes.
- **Functional Updates with Dataclasses:** Make `ToolResult` a frozen dataclass (`@dataclass(frozen=True)`) to enforce immutability and enable safe caching/hashing if needed.

F. Milestone Checkpoint:

After implementing the Tool System, you should be able to run the following test:

```bash
# Run the tool system unit tests

pytest tests/tools/ -v
```

Expected output should show passing tests for:

1. Registering a tool and retrieving it by name.
2. Validating correct arguments against a schema (success).
3. Validating incorrect arguments (failure with descriptive error).
4. Executing a mock tool successfully and receiving a `ToolResult`.
5. Handling a tool timeout and returning a `TIMEOUT` result.
6. Handling a tool exception and returning an `ERROR` result.

Manual Verification:

```python
# Create and register a simple tool

registry = ToolRegistry()

registry.register_tool(MyCalculatorTool())


# Execute via the engine

engine = ToolExecutionEngine()

result = await engine.execute_tool_call("calculator", {"expression": "2+2"})

print(result.status)  # Should be ToolResultStatus.SUCCESS

print(result.content) # Should be "4"
```

Signs of Trouble:

- **"Tool not found" errors:** Check that `ToolRegistry.register_tool` was called before execution.
- **Validation errors even with correct input:** Double-check your JSON Schema against the actual arguments dictionary structure. Use a JSON Schema validator online.
- **Timeout not working:** Ensure you're using `asyncio.wait_for` correctly and that your tool's `execute` method is truly `async`.

5.2 Component: ReAct Loop Engine (Milestone 2)

> **Milestone(s):** 2

The ReAct Loop Engine is the beating heart of the AI Executive—the core processor that translates high-level goals into concrete actions through iterative reasoning. While the Tool System (Milestone 1) provides the *capabilities* and the Planner (Milestone 3) provides the *strategy*, the ReAct Loop is the *tactical executor* that navigates the immediate terrain of a single subtask, making micro-decisions to overcome obstacles in real time.

Mental Model: The OODA Loop

Think of the ReAct Loop not as a simple script, but as the agent's **OODA Loop**—the Observe-Orient-Decide-Act cycle used by fighter pilots for rapid, adaptive decision-making under pressure.

- **Observe**: The agent receives the current state of the world—the user's query, the results of previous tool executions, and any relevant context from memory.
- **Orient**: The agent *thinks*—it processes this information using the LLM to understand the situation, assess options, and form a hypothesis about what action to take next.
- **Decide**: The agent selects a specific tool and formulates the exact parameters needed to execute it.
- **Act**: The agent executes the chosen tool and observes the result, which feeds back into the next Observation phase.

This cyclical process allows the agent to adapt to unexpected outcomes. If a tool fails, the next cycle re-orients and decides on a corrective action. If new information emerges, the agent can change course. The loop continues until the subtask is complete or a termination condition is met. This mental model emphasizes **adaptability** and **continuous feedback**—the agent isn't following a rigid script but piloting toward a goal, constantly adjusting based on what it observes.

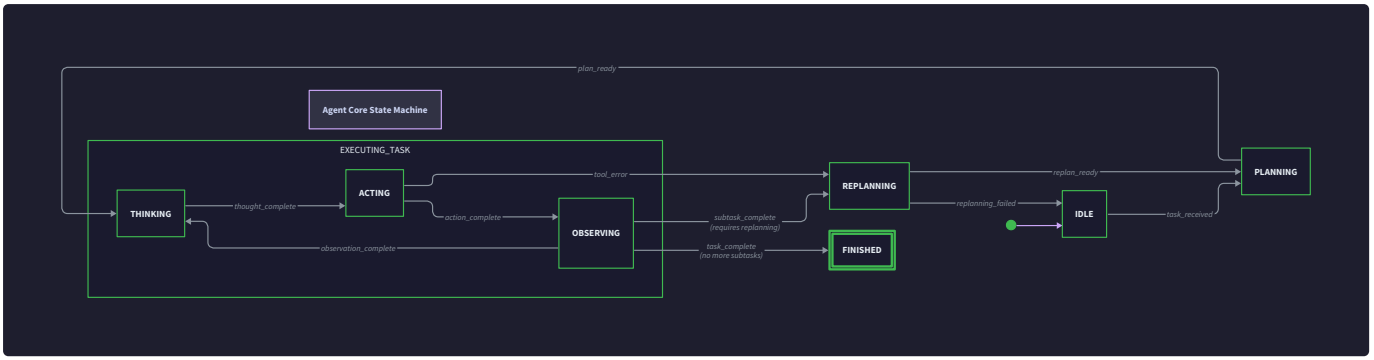
Loop Controller Interface

The ReAct Loop is encapsulated within a `ReActAgent` class that serves as the controller for a single subtask execution. Its primary interface is the `_execute_subtask` method, which implements the full Thought-Action-Observation cycle until completion. This method is called by the higher-level Planner (Milestone 3) for each `SubTask` in the plan.

The controller's interface is defined in the following table:

Method Name	Parameters	Returns	Description
`_execute_subtask`	`subtask: SubTask`	`ToolResult`	**Primary entry point.** Executes a single subtask by running the ReAct loop until the subtask's goal is satisfied, a final answer is produced, or a loop limit is reached. Internally manages the cycle of prompting the LLM, parsing actions, executing tools, and formatting observations.
`_format_react_prompt`	`subtask: SubTask`, `conversation_history: List[str]`, `available_tools: List[Tool]`	`str`	Constructs the prompt sent to the LLM for each reasoning step. Includes the task goal, conversation history, available tools with descriptions, and formatting instructions.
`_parse_llm_output`	`llm_response: str`	`Tuple[Optional[str], Optional[Dict[str, Any]]]`	Extracts the tool name and arguments from the LLM's textual response. Returns `(None, None)` if the response indicates a final answer.
`_run_single_cycle`	`current_state: Dict[str, Any]`	`Tuple[AgentState, Dict[str, Any]]`	Executes one full cycle of the loop (Think → Act → Observe). Returns the updated agent state and the accumulated context. Used internally by `_execute_subtask`.

The agent's internal state during loop execution follows a precise state machine, as shown in the diagram below. This state machine ensures clean transitions between thinking, acting, and observing, and provides clear hooks for monitoring and debugging.



State Transitions During ReAct Loop Execution:

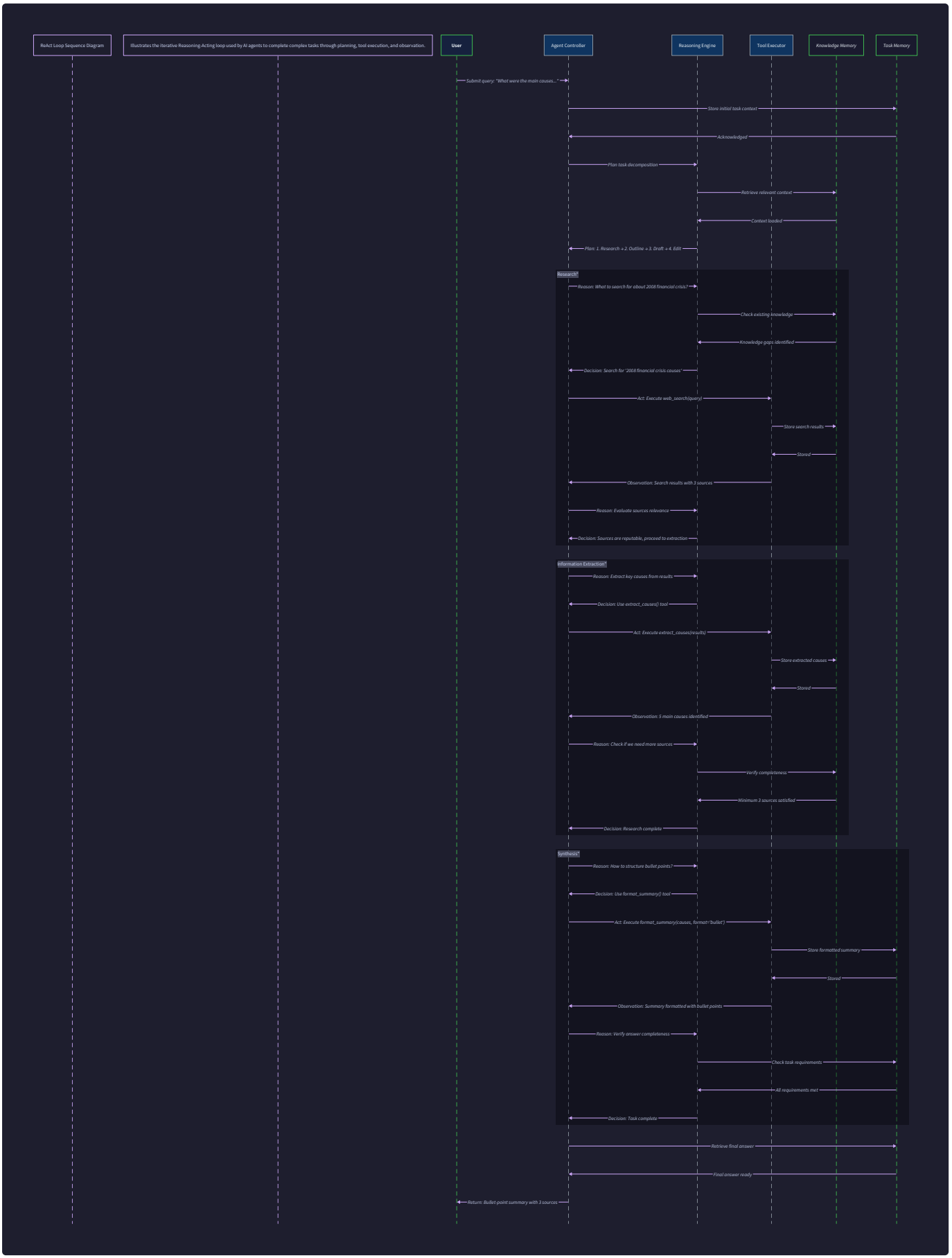
Current State	Event Trigger	Next State	Actions Taken
`IDLE`	`subtask_received`	`THINKING`	Initialize loop context, set iteration counter to 0, load relevant memories.
`THINKING`	`llm_response_received`	`ACTING` (if action) or `FINISHED` (if final answer)	Parse LLM output. If action, validate tool exists; if final answer, prepare result.
`ACTING`	`tool_execution_complete`	`OBSERVING`	Execute the selected tool via `ToolExecutionEngine.execute_tool_call()`.
`OBSERVING`	`observation_formatted`	`THINKING` (if continue) or `FINISHED` (if loop limit)	Format tool result as observation text, append to conversation history. Check iteration limit.
`THINKING`	`parsing_error`	`ERROR`	Log error details, set error state with descriptive message.
`ACTING`	`tool_error`	`THINKING`	Format tool error as observation, feed back into next cycle for recovery.
Any state	`iteration_limit_reached`	`FINISHED`	Stop loop, return timeout result with partial context.

Step-by-Step Loop Algorithm

The core ReAct algorithm is a deterministic cycle that runs within `_execute_subtask`. Below is the detailed, step-by-step procedure explained in prose. This algorithm assumes a subtask has been provided by the Planner, and the agent's goal is to produce a `ToolResult` containing either the final answer or an error.

1. **Initialization**: The agent receives a `SubTask` object. It extracts the `description` and `expected_output` as the goal. It initializes an empty `conversation_history` list to store the sequence of thoughts, actions, and observations. It sets an `iteration_counter` to 0 and defines a `max_iterations` (e.g., 10) to prevent infinite loops.
2. **Context Preparation**: The agent retrieves any relevant context from the Memory System (Milestone 4)—both short-term conversation history from previous subtasks and relevant long-term memories retrieved via semantic search based on the subtask description. This context is formatted and included in the initial prompt.
3. **Cycle Start (Think)**: The agent enters the `THINKING` state. It constructs a prompt using `_format_react_prompt`, which includes:
   - The subtask goal and any `expected_output` criteria.
   - The current conversation history (initially empty).
   - A list of available tools from the `ToolRegistry`, each with its `name`, `description`, and `parameters_schema`.
   - Explicit formatting instructions: the LLM must output its reasoning in a "Thought:" section, then either an "Action:" section (with JSON) or a "Final Answer:" section.
4. **LLM Reasoning**: The prompt is sent to the LLM via the configured provider (e.g., OpenAI API). The agent waits for the response and increments the `iteration_counter`.
5. **Output Parsing**: The agent parses the LLM's response using `_parse_llm_output`. This function looks for three possible patterns:
   - **Action Pattern**: Extracts a tool name and a JSON dictionary of arguments from the "Action:" section.
   - **Final Answer Pattern**: Detects a "Final Answer:" section and extracts the content.
   - **Malformed Output**: If neither pattern is found, it's a parsing error.
6. **Action Execution**: If an action was parsed:
   - The agent transitions to the `ACTING` state.
   - It validates that the tool name exists in the `ToolRegistry` via `ToolRegistry.get_tool()`.
   - It calls `ToolExecutionEngine.execute_tool_call(tool_name, arguments, timeout_seconds)` to safely run the tool with schema validation, timeout, and error handling.
   - The engine returns a `ToolResult` object with `status`, `content`, and any `error_message`.
7. **Observation Formatting**: The agent transitions to the `OBSERVING` state. It formats the `ToolResult` into a textual observation:
   - If `status` is `SUCCESS`: "Observation: [tool result content]"
   - If `status` is `ERROR` or `TIMEOUT`: "Observation: Error: [error message]" This observation is appended to the `conversation_history`.
8. **Loop Continuation Check**: The agent checks termination conditions:
   - **Final Answer Detected**: If step 5 found a "Final Answer:", the loop breaks, and the answer content is packaged into a `ToolResult.success()`.
   - **Iteration Limit Reached**: If `iteration_counter >= max_iterations`, the loop breaks with a `ToolResult.error()` indicating timeout.
   - **Subtask Goal Satisfied**: Optionally, the agent can evaluate if the `expected_output` criteria have been met based on observations (simple string matching or LLM evaluation). If satisfied, break with success.
   - Otherwise, return to step 3 for the next cycle.
9. **Result Packaging**: Once the loop terminates, the agent packages the final output or error into a `ToolResult` and returns it to the Planner. The entire `conversation_history` is also attached as metadata for debugging.

This algorithm creates a self-contained execution unit for a single subtask. The sequence of interactions between components during this loop is illustrated in the following diagram:



Example Walkthrough: Consider a subtask with description "Find the current population of Tokyo." The agent initializes, then in cycle 1 thinks: "I need to search for Tokyo's population. I can use the web_search tool." It outputs an Action to call `web_search` with query "Tokyo current population 2024". The tool returns a search

result snippet. The observation is formatted and added to history. Cycle 2 thinks: "The search result says Tokyo's population is about 37 million. I should present this as the final answer." It outputs a Final Answer. The loop terminates with success.

Action Parsing Strategies

A critical technical challenge is reliably extracting structured tool calls from the LLM's free-form text. The LLM is instructed to output an "Action:" block with JSON, but it may hallucinate formats, omit brackets, or produce malformed JSON. The parsing strategy must be robust to these variations while rejecting invalid input.

We evaluate three primary strategies:

Strategy	How It Works	Pros	Cons
Regular Expression (Regex) Extraction	Use regex patterns to find text between `Action:` and newlines, then extract a JSON-like structure. Can be lenient with whitespace and minor syntax errors.	Simple to implement, fast, no external dependencies. Can handle some LLM formatting quirks.	Fragile to complex JSON nesting. Difficult to maintain as patterns multiply. May incorrectly parse nested quotes or brackets.
Function-Calling API	Use the LLM provider's native function-calling feature (e.g., OpenAI's `tools` parameter). The LLM returns a structured JSON object in a separate field.	Highly reliable, native support from major providers. Guaranteed valid JSON. Offloads parsing complexity to the provider.	Vendor lock-in. Not all LLM providers support it. Less transparent (harder to debug the raw prompt).
JSON-in-Text Parsing with Fallback	Use a dedicated parser (like `jtoken` or `json5`) that scans the text for the first valid JSON object after "Action:". If that fails, use a regex fallback to extract a likely JSON string and attempt repair.	Balances robustness and independence. Can handle many edge cases. Works with any LLM provider.	More complex implementation. Repair logic can introduce its own bugs. Slower than regex.

> **Decision: JSON-in-Text Parsing with Fallback**
>
> - **Context**: We need a parsing strategy that works reliably across different LLM providers (OpenAI, Anthropic, local models) while handling the common formatting mistakes LLMs make. The parser must be part of our open framework, not dependent on proprietary APIs.
> - **Options Considered**: 1) Regex extraction, 2) Function-calling API, 3) JSON-in-text parsing with fallback.
> - **Decision**: Implement **JSON-in-text parsing with a lenient fallback** as the primary strategy.
> - **Rationale**: This approach provides the best balance of independence, robustness, and debuggability. Using a library like `jtoken` allows us to find and parse JSON even if it's embedded in markdown or has trailing commas. The fallback to regex and JSON repair (e.g., using `json5` or simple string patching) handles cases where the LLM outputs malformed JSON. This keeps the framework provider-agnostic while still being tolerant of LLM errors.
> - **Consequences**: We must implement and maintain a moderately complex parser. However, this complexity is isolated to a single module, and the increased robustness will reduce learner frustration when their agent gets stuck due to parsing failures. The parser will also provide detailed error messages to aid debugging.

Parsing Process Details:

1. The raw LLM response is searched for the string "Action:" (case-insensitive).
2. The substring after "Action:" up to the next "Thought:" or "Final Answer:" (or end of text) is extracted as the `action_text`.
3. A JSON tokenizer scans `action_text` to find the first complete JSON object (starting with `{` and ending with matching `}`).
4. If found, parse it with the standard `json` module.
5. If not found, apply a fallback: use regex to find text between curly braces, then attempt to repair common issues (add missing quotes, fix trailing commas) using a simple heuristic or `json5` library.
6. Validate that the parsed object contains `"tool"` and `"arguments"` keys.
7. If all fails, raise a `ParsingError` with the problematic text, which the loop will treat as an observation error, feeding back to the LLM for correction.

ADR: Loop Termination Strategy

Knowing when to stop the loop is non-trivial. The agent might keep taking actions indefinitely, or it might declare victory prematurely. We need a robust termination strategy that balances autonomy with safety.

Decision: Hybrid Termination Strategy

- **Context**: The ReAct loop must eventually stop—either because the task is complete or because it's stuck. We need deterministic guarantees against infinite loops while allowing the agent to finish naturally.
- **Options Considered**:
  1. **Fixed Iteration Limit**: Stop after N cycles (e.g., 10) regardless of progress.
  2. **Explicit Final Answer Token**: Only stop when the LLM outputs a special "Final Answer:" section.
  3. **External Monitor with Goal Checking**: Use a separate LLM call or rule-based evaluator each cycle to assess if the subtask goal has been satisfied.
- **Decision**: Implement a **hybrid strategy combining a fixed iteration limit with explicit final answer detection, augmented by optional goal satisfaction checking**.
- **Rationale**: A fixed iteration limit is essential as a safety net—it guarantees termination even if the agent is stuck in a loop. The explicit final answer token gives the agent control to finish early when it believes it has the answer. The optional goal checker (configurable) adds an extra layer of validation, ensuring the agent's final answer actually addresses the subtask's `expected_output`. This three-layer approach provides both safety and flexibility.
- **Consequences**: This increases implementation complexity slightly, as we need to manage multiple termination conditions. However, each condition is simple to implement, and together they create a robust stopping mechanism. The iteration limit should be configurable per subtask, as complex tasks may require more steps.

Termination Condition Comparison:

Condition	Trigger	Pros	Cons	Implementation Priority
Iteration Limit	`iteration_counter >= max_iterations`	Guarantees termination, simple to implement.	May cut off productive loops prematurely.	**Required**
Explicit Final Answer	LLM output contains "Final Answer:" section.	Agent-controlled, natural, follows ReAct paper.	Agent may output final answer incorrectly or prematurely.	**Required**
Goal Satisfaction Check	`expected_output` is met (string match or LLM evaluation).	Ensures objective completion, catches hallucinated final answers.	Adds latency (if using LLM) or complexity (rule-based).	**Optional**

The recommended default is to use iteration limit (configurable, default 10) + explicit final answer detection. The goal satisfaction check can be enabled for subtasks where `expected_output` is specified and precise verification is needed.

Common Pitfalls and Mitigations

Implementing a ReAct loop seems straightforward, but several subtle traps can cause the agent to fail in frustrating ways. Below are the most common pitfalls, why they occur, and how to avoid them.

⚡ Pitfall 1: Infinite Loops with No Progress

- **Description**: The agent repeatedly takes the same action or cycles through a set of actions without getting closer to the goal. Example: searching for the same query over and over.
- **Why it's wrong**: Consumes API credits and time, never produces a result.
- **Fix**: Implement **progress tracking**. Maintain a set of (tool, arguments) pairs from previous cycles. If the same action is about to be repeated, inject a warning observation: "You already tried this action with the same arguments. Consider a different approach." Also, enforce the iteration limit strictly.

⚡ Pitfall 2: Hallucinated Tools

- **Description**: The LLM invokes a tool that doesn't exist in the registry (e.g., `send_email` when only `web_search` is available).
- **Why it's wrong**: The parser may extract the tool name, but execution will fail because the tool is missing, breaking the loop.
- **Fix**: **Validate tool existence immediately after parsing**. Before calling `ToolExecutionEngine`, check `ToolRegistry.get_tool(tool_name)`. If missing, format an observation: "Tool `{tool_name}` is not available. Available tools are: [list]. Please choose from these." This turns the error into a learning opportunity for the agent.

⚡ Pitfall 3: Parsing Failures on Valid LLM Output

- **Description**: The LLM outputs a correctly formatted action, but the parser fails to extract it due to subtle formatting differences (extra spaces, line breaks, markdown code blocks).
- **Why it's wrong**: The agent stalls because it cannot proceed, even though the LLM did its job correctly.
- **Fix**: Use the **robust JSON-in-text parsing with fallback** described above. Additionally, log the raw LLM response and the parsing error for debugging. Consider adding a "graceful degradation" where if parsing fails, you ask the LLM to reformat: "Could not parse your action. Please output Action: as valid JSON only."

⚡ Pitfall 4: Context Window Mismanagement

- **Description**: The `conversation_history` grows with each cycle, eventually exceeding the LLM's token limit, causing truncation or API errors.

- **Why it's wrong**: Critical early context (the original goal, key observations) gets truncated, leading to incoherent behavior.
- **Fix**: Implement **context window management** (detailed in Milestone 4). For the ReAct loop specifically, you can:
  - Keep only the last K thought-action-observation triples.
  - Summarize older triples into a concise summary using an LLM.
  - Include the original subtask description in every prompt, not just the first.

⚡ **Pitfall 5: No Error Feedback for Tool Failures**

- **Description**: When a tool returns a `ToolResult` with `status=ERROR`, the agent simply passes the raw error (e.g., "HTTP 500") to the LLM without context.
- **Why it's wrong**: The LLM may not understand how to recover from technical errors.
- **Fix**: **Enrich error observations**. Format tool errors with suggestions: "Observation: The web_search tool failed with error 'Connection timeout'. This might be a temporary network issue. You could retry the same action or try a different search query." This guides the LLM toward appropriate recovery actions.

⚡ **Pitfall 6: Missing Timeout on LLM Calls**

- **Description**: The LLM API call hangs indefinitely due to network issues or provider slowness, freezing the entire agent.
- **Why it's wrong**: The agent becomes unresponsive, requiring manual restart.
- **Fix**: Wrap the LLM API call in a **timeout**. Use `asyncio.wait_for` or a thread with a timeout. If the call times out, treat it as a tool-like error and format an observation: "The LLM is not responding. Please retry your last thought." Also, consider implementing a retry with exponential backoff for transient failures.

Implementation Guidance

This section provides concrete code and structure to implement the ReAct Loop Engine. The primary language is Python, using async/await for non-blocking tool and LLM calls.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
LLM Client	`openai` Python library (sync)	`openai` async client or `litellm` for multi-provider support
JSON Parsing	Python's `json` + regex fallback	`jtoken` library for robust tokenization, `json5` for repair
Timeout/Async	`asyncio.wait_for` with `asyncio.TimeoutError`	`anyio` or `trio` for structured concurrency, retry with backoff
State Management	Simple dictionary for loop context	Pydantic models for validation, immutable updates via `dataclasses.replace`

B. Recommended File/Module Structure

Place the ReAct loop engine within the `agent_core` module, as it's a core execution component.

```
project_archon/
├── agents/
│   ├── __init__.py
│   ├── base_agent.py          # Abstract Agent class
│   └── react_agent.py         # ReActAgent implementation (this component)
├── tools/                     # From Milestone 1
│   ├── registry.py
│   └── execution_engine.py
├── memory/                    # From Milestone 4
│   └── conversation_memory.py
├── llm/
│   └── client.py              # LLM client wrapper
└── utils/
    └── parsing.py             # JSON-in-text parser utilities
```

C. Infrastructure Starter Code

First, a robust JSON-in-text parser with fallback. This is a utility that can be used across the codebase.

PYTHON

```python
# utils/parsing.py

import json

import re

from typing import Any, Dict, Optional, Tuple

import json5  # pip install json5


class ParsingError(Exception):

    """Raised when unable to parse an action from LLM output."""

    pass


def extract_json_from_text(text: str) -> Optional[Dict[str, Any]]:

    """

    Find the first valid JSON object in text.

    Tries strict JSON first, then falls back to JSON5 for lenient parsing.

    """

    # Try to find a JSON object pattern

    pattern = r'\{[^{}]*\}'  # Simplistic; better to use a stack

    # But we'll use a simple approach: find first '{' and last '}' after it

    start = text.find('{')

    if start == -1:

        return None

    # Find matching closing brace

    brace_count = 0

    end = -1

    for i, ch in enumerate(text[start:]):

        if ch == '{':

            brace_count += 1

        elif ch == '}':

            brace_count -= 1

            if brace_count == 0:

                end = start + i + 1

                break

    if end == -1:

        return None

    json_str = text[start:end]



    # Try standard JSON

    try:

        return json.loads(json_str)

    except json.JSONDecodeError:

        pass
```

```
    # Try JSON5 (more lenient)
    try:
        return json5.loads(json_str)
    except Exception:
        pass

    return None

def parse_llm_action_response(response_text: str) -> Tuple[Optional[str], Optional[Dict[str, Any]]]:
    """
    Parse LLM response text for Action or Final Answer.
    Returns (tool_name, arguments) if action, (None, None) if final answer.
    Raises ParsingError if cannot parse.
    """
    # Normalize line endings
    text = response_text.strip()

    # Check for Final Answer
    final_answer_match = re.search(r'Final Answer:\s*(.*?)(?=\n\s*(?:Thought:|Action:|$))', text, re.DOTALL | re.IGNORECASE)
    if final_answer_match:
        return None, None  # Signal final answer

    # Find Action block
    action_match = re.search(r'Action:\s*(.*?)(?=\n\s*(?:Thought:|Final Answer:|$))', text, re.DOTALL | re.IGNORECASE)
    if not action_match:
        raise ParsingError("No Action or Final Answer found in LLM response.")

    action_text = action_match.group(1).strip()

    # Try to extract JSON
    json_obj = extract_json_from_text(action_text)
    if not json_obj:
        raise ParsingError(f"Could not extract JSON from action text: {action_text}")

    # Validate structure
    if not isinstance(json_obj, dict):
        raise ParsingError(f"Extracted JSON is not a dictionary: {json_obj}")

    tool_name = json_obj.get("tool")
    arguments = json_obj.get("arguments")
```

```
    if not tool_name or not isinstance(arguments, dict):
        raise ParsingError(f"Missing 'tool' or 'arguments' keys in JSON: {json_obj}")

    return tool_name, arguments
```

D. Core Logic Skeleton Code

Now the main `ReActAgent` class skeleton. This follows the step-by-step algorithm described earlier.

PYTHON

```python
# agents/react_agent.py

import asyncio

from dataclasses import dataclass, field

from typing import List, Optional, Dict, Any, Tuple

from enum import Enum


from tools.registry import ToolRegistry

from tools.execution_engine import ToolExecutionEngine

from llm.client import LLMClient

from utils.parsing import parse_llm_action_response, ParsingError

from .base_agent import BaseAgent


class AgentState(Enum):

    IDLE = "idle"

    THINKING = "thinking"

    ACTING = "acting"

    OBSERVING = "observing"

    FINISHED = "finished"

    ERROR = "error"


@dataclass

class ReActLoopContext:

    """Mutable context for a single ReAct loop execution."""

    subtask_description: str

    expected_output: Optional[str] = None

    conversation_history: List[str] = field(default_factory=list)

    iteration: int = 0

    max_iterations: int = 10

    state: AgentState = AgentState.IDLE

    # Track previous actions to avoid loops

    previous_actions: List[Tuple[str, Dict[str, Any]]] = field(default_factory=list)


class ReActAgent(BaseAgent):

    """

    Implements the ReAct (Reasoning + Acting) loop for a single subtask.

    """

    def __init__(

        self,

        llm_client: LLMClient,

        tool_registry: ToolRegistry,

        tool_execution_engine: ToolExecutionEngine,

        max_iterations: int = 10
```

```
):
    self.llm_client = llm_client
    self.tool_registry = tool_registry
    self.tool_execution_engine = tool_execution_engine
    self.max_iterations = max_iterations

async def _execute_subtask(self, subtask: SubTask) -> ToolResult:
    """
    Execute a single subtask using the ReAct loop.

    Args:
        subtask: The SubTask to execute, containing description, expected_output, etc.

    Returns:
        ToolResult with final answer or error.
    """
    # TODO 1: Initialize ReActLoopContext
    #   - Extract subtask.description and subtask.expected_output
    #   - Set max_iterations from instance or subtask metadata
    #   - Initialize empty conversation_history and previous_actions

    # TODO 2: Load relevant context from memory (if available)
    #   - Retrieve conversation history from previous subtasks (if any)
    #   - Perform semantic search for relevant long-term memories
    #   - Format these into a context string for the prompt

    # TODO 3: Start main loop
    #   - While iteration < max_iterations and state != FINISHED:
    #       1. Set state = THINKING
    #       2. Format prompt using _format_react_prompt
    #       3. Call LLM with timeout (use asyncio.wait_for)
    #       4. Parse response using parse_llm_action_response
    #       5. If parsing error -> set observation as error, continue
    #       6. If final answer -> break loop, prepare success result
    #       7. If action:
    #           a. Validate tool exists in registry
    #           b. Check if same action was tried before (avoid loops)
    #           c. Set state = ACTING
    #           d. Execute tool via tool_execution_engine.execute_tool_call
    #           e. Set state = OBSERVING
    #           f. Format observation based on ToolResult status
```

```
        #           g. Append observation to conversation_history
        #       8. Increment iteration counter

        # TODO 4: Check termination conditions
        #   - If final answer found: package as ToolResult.success
        #   - If iteration >= max_iterations: return ToolResult.error with "Max iterations reached"
        #   - If expected_output satisfied (optional): evaluate and return success

        # TODO 5: Attach debugging metadata
        #   - Include the full conversation_history in ToolResult.metadata
        #   - Include the iteration count and final state

        pass  # Remove this line when implementing

    def _format_react_prompt(
        self,
        context: ReActLoopContext,
        available_tools: List[Tool],
        external_context: str = ""
    ) -> str:
        """
        Format the prompt for the LLM based on current ReAct loop state.

        Args:
            context: Current loop context (history, iteration, etc.)
            available_tools: List of Tool objects the agent can use
            external_context: Any additional context from memory

        Returns:
            Formatted prompt string.
        """
        # TODO 1: Start with system instructions
        #   - Define the agent's role: "You are an AI assistant that uses tools to solve tasks."
        #   - Explain the Thought-Action-Observation format

        # TODO 2: Add the task description
        #   - Include context.subtask_description
        #   - If context.expected_output exists, add "Expected output: ..."

        # TODO 3: Add external context if provided
        #   - Prefix with "Relevant context from previous tasks or memory:"
```

```
        # TODO 4: Add conversation history
        #   - Format each entry in context.conversation_history as "Thought/Action/Observation: ..."
        #   - If history is long, consider truncating or summarizing (Milestone 4)

        # TODO 5: List available tools
        #   - For each tool in available_tools:
        #       - tool.name
        #       - tool.description
        #       - JSON schema of parameters (simplified)
        #   - Example: "web_search: Search the web. Args: {'query': 'string'}"

        # TODO 6: Add formatting instructions
        #   - "You must output either:"
        #   - "Thought: [your reasoning]"
        #   - "Action: {'tool': 'tool_name', 'arguments': {...}}"
        #   - "Or when done: Final Answer: [your answer]"

        # TODO 7: Add current step instruction
        #   - "Based on the above, what is your next thought and action?"

        pass  # Remove this line when implementing

    async def _run_single_cycle(
        self,
        context: ReActLoopContext,
        available_tools: List[Tool],
        external_context: str = ""
    ) -> Tuple[AgentState, ReActLoopContext]:
        """
        Execute one cycle of the ReAct loop.

        Returns:
            Updated agent state and modified context.
        """
        # TODO 1: Format and send prompt to LLM
        #   - Call _format_react_prompt with current context
        #   - Use llm_client.complete() with appropriate parameters
        #   - Wrap in asyncio.wait_for with timeout (e.g., 30 seconds)

        # TODO 2: Parse LLM response
```

```
        #   - Try parse_llm_action_response
        #   - If ParsingError, set observation as "Error parsing your response: {error}"
        #   - Return state THINKING (next cycle will see the error observation)

        # TODO 3: Handle final answer
        #   - If parser returns (None, None), set state = FINISHED
        #   - Store final answer in context (maybe as a new field)

        # TODO 4: Handle action
        #   - Validate tool exists in registry (if not, format error observation)
        #   - Check if same (tool, arguments) is in context.previous_actions
        #     If yes, add warning to observation: "You already tried this action."
        #   - Execute tool with tool_execution_engine.execute_tool_call
        #   - Format observation based on ToolResult
        #   - Append (tool, arguments) to previous_actions
        #   - Append observation to conversation_history

        # TODO 5: Update iteration count
        #   - context.iteration += 1

        pass  # Remove this line when implementing
```

E. Language-Specific Hints

- **Async/Await Pattern**: Use `asyncio.create_task` for parallel tool execution when multiple subtasks can run concurrently (Milestone 3). For now, keep the ReAct loop sequential.
- **Timeout Handling**: Always wrap LLM calls and tool executions in `asyncio.wait_for`. Catch `asyncio.TimeoutError` and convert to a `ToolResult.timeout`.
- **Immutable Updates**: Since `ReActLoopContext` is a dataclass, use `dataclasses.replace(context, iteration=context.iteration + 1)` for functional updates. This makes debugging easier.
- **Logging**: Use structured logging (Python's `logging` module with JSON formatter) to log each cycle's state, action, and observation. This is crucial for debugging agent behavior.

F. Milestone Checkpoint

After implementing the ReAct Loop Engine, you should be able to run a simple end-to-end test:

PYTHON

```python
# test_react_agent.py

import asyncio

from tools.registry import ToolRegistry

from tools.execution_engine import ToolExecutionEngine

from llm.client import MockLLMClient  # A mock that returns predefined responses

from agents.react_agent import ReActAgent

from data_model import SubTask, ToolResult


async def test_react_loop():

    # 1. Set up mock LLM that simulates a ReAct sequence

    mock_llm = MockLLMClient(responses=[

        "Thought: I need to calculate 15 * 3. Action: {'tool': 'calculator', 'arguments': {'expression': '15 * 3'}}",

        "Thought: The result is 45. Final Answer: 45"

    ])


    # 2. Register a calculator tool

    registry = ToolRegistry()

    # ... register a simple calculator tool (from Milestone 1)


    # 3. Create engine and agent

    engine = ToolExecutionEngine(registry)

    agent = ReActAgent(mock_llm, registry, engine, max_iterations=5)


    # 4. Create a subtask

    subtask = SubTask(

        id="test-1",

        description="Calculate 15 times 3",

        expected_output="45",

        tool_name=None,

        tool_args=None,

        dependencies=[],

        status=SubTaskStatus.PENDING,

        result=None

    )


    # 5. Execute

    result = await agent._execute_subtask(subtask)


    # 6. Verify

    assert result.status == ToolResultStatus.SUCCESS

    assert "45" in result.content
```

```
    print("√ ReAct loop test passed!")

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

Expected Output: The test should pass without errors. The agent should complete in 2 iterations, using the calculator tool and producing the final answer "45".

Signs of Problems:

- If the agent exceeds max iterations: Check parsing logic—the mock LLM response might not be parsed correctly.
- If tool execution fails: Verify the calculator tool is properly registered and its `execute` method works.
- If the final answer isn't captured: Ensure the parser correctly detects "Final Answer:" and extracts content.

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Agent gets stuck in infinite loop	No progress detection, LLM repeats same action.	Log `previous_actions` each cycle. Check if same (tool, args) appears.	Implement duplicate action detection and inject warning observation.
LLM output not parsed correctly	LLM formats action differently than expected.	Log raw LLM response and the extracted action_text.	Adjust regex patterns in `parse_llm_action_response` or improve JSON extraction.
Tool errors not being fed back	ToolResult.ERROR is not formatted into observation.	Check the `ToolResult.status` and the observation formatting logic.	Ensure all ToolResult statuses are mapped to appropriate observation text.
Context window exceeded	Conversation history grows too large.	Count tokens in prompt (use `tiktoken` or similar).	Implement history truncation: keep only last N triples, or summarize old ones.
Timeout on LLM call	Network issue or LLM provider slow.	Wrap LLM call in `asyncio.wait_for` and catch TimeoutError.	Add retry with exponential backoff, or treat timeout as a tool error for the agent to handle.

5.3 Component: Planning & Task Decomposition (Milestone 3)

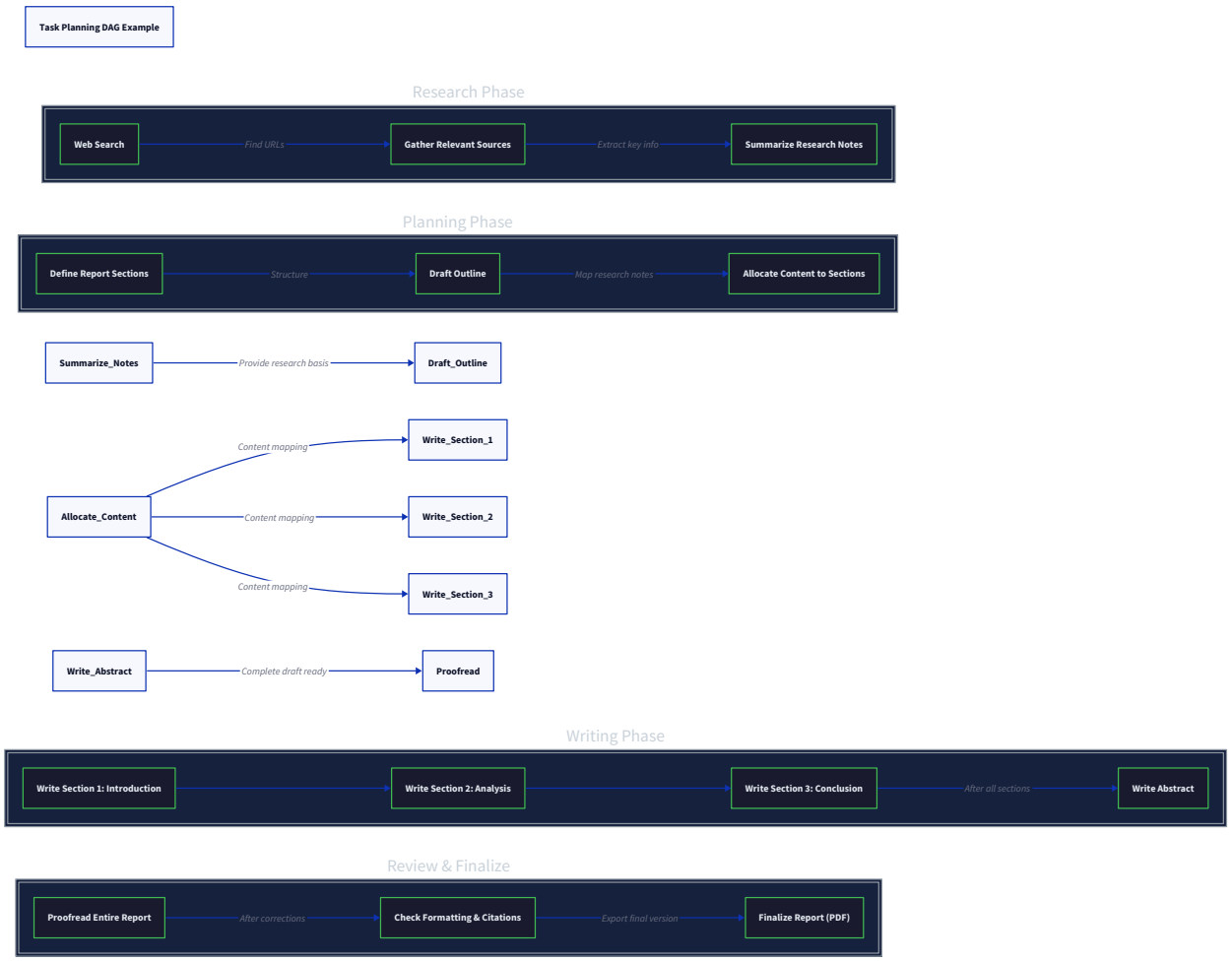
> **Milestone(s):** 3

The Planning & Task Decomposition component is the **strategic cortex** of the AI Executive. While the ReAct Loop excels at tactical execution of single steps, it lacks the ability to look ahead, anticipate dependencies, and optimize the order of operations for complex, multi-faceted goals. This component gives the agent the ability to **break down ambitious objectives into manageable steps, sequence them intelligently, and adapt when reality diverges from the plan**.

Mental Model: Project Manager and Gantt Chart

Imagine you're a project manager handed a complex objective: "Redesign the company website." You wouldn't start coding immediately. Instead, you'd create a **Work Breakdown Structure (WBS)**, listing all required activities: "conduct user research," "create wireframes," "develop frontend," "write content," "perform QA testing." Crucially, you'd identify **dependencies**: wireframes must come before development, and QA can't start until development is complete. You'd then sequence these tasks into a **Gantt chart**—a visual timeline showing what runs in sequence and what can run in parallel.

The Planning component embodies this project manager. The **Planner** is responsible for creating the WBS (decomposing the goal into `SubTask` objects). The **Executor** is responsible for walking the Gantt chart (the Directed Acyclic Graph or DAG) in the correct order, dispatching each subtask to the ReAct engine, and tracking progress. When a task fails (e.g., a tool is unavailable), the manager must **replan**—adjusting the remaining work based on new information.



Planner and Executor Interface

The planning subsystem is composed of two primary interfaces: the `Planner` (which creates the plan) and the `Executor` (which runs it). They operate on the core data structures `Task` and `SubTask`.

Core Data Structures

Name	Type	Description
`SubTaskStatus`	`Enum`	The lifecycle state of a single subtask. Values: `PENDING`, `EXECUTING`, `COMPLETED`, `FAILED`, `BLOCKED`.
`SubTask`	`Data Class`	An atomic unit of work within a larger task.
`Task`	`Data Class`	The container for a user's original request, holding the overall goal and the decomposed subtasks.
`Plan`	`Data Class`	The executable representation of a `Task`, explicitly capturing the dependency graph (`subtask_dag`) and a calculated execution order.

`SubTask` Fields:

Field	Type	Description
`id`	`str`	Unique identifier (e.g., UUID or sequential number).
`description`	`str`	A natural language description of what this subtask aims to accomplish.
`expected_output`	`Optional[str]`	A specification of what a successful completion looks like (used for validation and LLM guidance).
`tool_name`	`Optional[str]`	The name of the `Tool` to invoke for this subtask. If `None`, the subtask is a pure "thinking" step.
`tool_args`	`Optional[Dict[str, Any]]`	The arguments to pass to the specified tool.
`dependencies`	`List[str]`	List of `SubTask.id` values that must be `COMPLETED` before this subtask can start.
`status`	`SubTaskStatus`	Current execution state.
`result`	`Optional[ToolResult]`	The outcome of the subtask's execution, populated once status is `COMPLETED` or `FAILED`.

`Task` Fields:

Field	Type	Description
`id`	`str`	Unique identifier for the overall task.
`original_query`	`str`	The user's original input request.
`goal`	`str`	A refined, actionable statement of the overall objective.
`subtasks`	`List[SubTask]`	The decomposed steps.
`context`	`Dict[str, Any]`	A **bag of properties** for storing shared state, intermediate results, or parameters that need to be passed between subtasks.

`Plan` Fields:

Field	Type	Description
`task_id`	`str`	References the parent `Task.id`.
`subtask_dag`	`Dict[str, List[str]]`	Adjacency list representation of the dependency graph. Keys are subtask IDs, values are lists of IDs that *depend on* the key. (Alternative: keys are IDs, values are IDs they depend *on*. Our convention is `{A: [B, C]}` means B and C depend on A).
`execution_order`	`List[str]`	A linearized sequence of subtask IDs (via topological sort) that respects all dependencies. This is precomputed for execution.

Planner Interface

The `Planner` is responsible for the `create_plan(goal)` operation.

Method	Parameters	Returns	Description
`create_plan`	`goal: str`, `available_tools: List[Tool]`, `initial_context: Optional[Dict[str, Any]]`	`Task`	Analyzes the high-level goal, uses the LLM (or a rule-based system) to decompose it into a list of `SubTask` objects with dependencies, and returns a `Task`.

Executor Interface

The `Executor` is responsible for the `execute_plan(plan)` operation. In our architecture, the `Agent` class often acts as the executor, coordinating between the plan and the ReAct engine.

Method	Parameters	Returns	Description
`execute_plan`	`task: Task`, `tool_registry: ToolRegistry`, `memory_context: Optional[str]`	`ToolResult`	Takes a `Task`, executes its subtasks in dependency order using the ReAct engine, and returns the final aggregated result.
`_execute_subtask`	`subtask: SubTask`, `context: Dict[str, Any]`	`ToolResult`	**Internal.** Runs a single subtask by invoking the ReAct loop with the subtask's description as the query. Merges the subtask's `tool_args` with the shared `context`.

> **Design Insight:** The separation between `Task` (data) and `Plan` (executable blueprint) follows the classic **Strategy Pattern**. The `Task` is the problem specification, while the `Plan` is one possible solution strategy. This allows for future alternative planners (e.g., a rule-based planner for simple tasks) to generate different `Plan` structures from the same `Task`.

Planning and Execution Algorithm

The end-to-end process from user query to completed plan involves several coordinated stages.

Stage 1: Task Decomposition (Planning)

1. **Goal Reception & Context Setup:** The `Agent` receives a `user_query`. It creates a new `Task` object, storing the `original_query` and setting the `goal` field (which may be a cleaned-up version of the query). An initial `context` dictionary is populated with any relevant information from the user's session or retrieved memories.
2. **LLM Prompting for Decomposition:** The `Planner` formats a prompt for the LLM containing: the `goal`, a list of available tools (names and descriptions), and instructions for decomposition. The instructions mandate the output to be a list of subtasks, each with an `id`, `description`, `expected_output`, optional `tool_name` and `tool_args`, and a list of `dependencies` by `id`.
3. **Structured Output Parsing:** The LLM's response is parsed (e.g., extracting a JSON list). Each parsed object is validated and used to instantiate a `SubTask` with status `PENDING`.
4. **Dependency Graph Construction:** The `Planner` builds the `subtask_dag` adjacency list by inverting the `dependencies` lists from each `SubTask`. It then performs a **cycle detection** algorithm (e.g., Depth-First Search) to ensure the graph is acyclic. If a cycle is found, the planner attempts to break it by removing the minimal number of dependencies or requests a re-decomposition from the LLM.
5. **Topological Sort:** The planner runs a topological sort (Kahn's algorithm) on the DAG to produce the `execution_order` list. This order is stored within a `Plan` object linked to the `Task`.

Stage 2: Plan Execution

1. **Executor Initialization:** The `Executor` receives the `Task` and its associated `Plan`. It sets the overall `AgentState` to `EXECUTING`.
2. **Iterative Subtask Processing:** The executor iterates through the `execution_order`. For each subtask ID: a. **Readiness Check:** It retrieves the `SubTask` and calls `SubTask.is_ready()`, which checks if all tasks in its `dependencies` list have status `COMPLETED`. If not, the subtask's status is set to `BLOCKED`, and the executor skips to the next available task. b. **Status Transition:** The ready subtask's status is updated to `EXECUTING` via a **functional update** (`Task.update_subtask(subtask.with_status(EXECUTING))`). c. **Context Preparation:** The executor prepares an execution context by merging the `Task.context` (shared state) with the subtask's specific `tool_args`. d. **ReAct Loop Invocation:** The executor calls `Agent._execute_subtask(subtask)`, which runs the full ReAct loop using the subtask's `description` as the query and the prepared context. e. **Result Capture:** The returned `ToolResult` is attached to the subtask via another functional update (`subtask.with_result(result)`). The subtask status is set to `COMPLETED` (if `result.status` is `SUCCESS`) or `FAILED`. f. **Context Update:** If the subtask succeeded, any outputs or new facts are written back into the `Task.context` dictionary so subsequent subtasks can access them.
3. **Parallel Execution (Optional):** The executor can identify independent subtasks (those with no dependencies on each other) and dispatch them to separate worker threads or asynchronous tasks, joining their results before proceeding to dependent tasks.
4. **Completion Detection:** After all subtasks in the `execution_order` have been processed (reaching `COMPLETED` or `FAILED`), the executor evaluates the overall `Task`. If all subtasks are `COMPLETED`, it aggregates the final results from the `Task.context` or the last subtask, sets `AgentState` to `FINISHED`, and returns a final `ToolResult.success()`. If any subtask `FAILED` and cannot be recovered, it transitions to `ERROR`.

DAG Representation and Execution

The Directed Acyclic Graph (DAG) is the mathematical model for our task dependencies. It ensures tasks are executed in a logical, deadlock-free order.

Representation: We use an **adjacency list** stored in the `Plan.subtask_dag: Dict[str, List[str]]`. The convention `{A: [B, C]}` means "B and C depend on A" (A must complete before B and C). This is efficient for querying dependents.

Traversal - Topological Sort (Kahn's Algorithm):

1. Compute the in-degree (number of dependencies) for each node.
2. Initialize a queue with all nodes having in-degree 0 (no dependencies).
3. While the queue is not empty: a. Dequeue a node `n`, add it to the `execution_order`. b. For each dependent `m` in `subtask_dag[n]`, decrement its in-degree. c. If `m`'s in-degree becomes 0, enqueue `m`.
4. If the `execution_order` contains all nodes, the sort succeeded. If some nodes remain, a cycle exists.

Execution with State Management: During execution, the `Executor` must respect the topological order but also dynamically manage state. A subtask can only run if its status is `PENDING` *and* all its dependencies are `COMPLETED`. The `SubTask.is_ready()` method encapsulates this check. The executor maintains a work queue of ready tasks, which can be processed by a pool of workers for parallelism.

> **Key Challenge:** Parallel execution must handle shared access to the `Task.context` dictionary. A thread-safe structure (like `asyncio.Lock` or `threading.Lock`) or immutable updates (functional updates that create a new context dict) are required to prevent race conditions.

ADR: Replanning Trigger Strategy

> **Decision: Replan on Critical Failure with Adaptive Thresholds**
>
> - **Context:** During plan execution, subtasks can fail for various reasons: tool errors, unexpected outputs, or changing external conditions. The agent must decide when to abandon the current plan and create a new one versus retrying the failed step or proceeding with a degraded plan.
> - **Options Considered:**
>   1. **Replan on Every Failure:** Trigger a full re-decomposition whenever any subtask fails.
>   2. **Threshold-Based Replanning:** Replan only after N consecutive failures, or if a failure occurs in a "critical path" subtask.
>   3. **Confidence-Guided Replanning:** Use the LLM to assess the severity of the failure and decide whether to replan, retry, or adjust the goal.
> - **Decision:** Implement **Threshold-Based Replanning** with the ability to mark subtasks as "critical." The default is to replan after 3 failures of the same subtask, or immediately upon the failure of any subtask explicitly marked as critical during planning.
> - **Rationale:** Replanning on every failure is inefficient and can lead to oscillation. Confidence-guided replanning adds significant LLM latency and complexity for marginal gain. A threshold-based approach is simple, predictable, and allows for transient errors (e.g., network timeouts) to be retried without discarding a potentially sound plan. Marking critical path tasks (those with many dependents) allows the system to recognize when a failure jeopardizes the entire plan.
> - **Consequences:** The executor must track failure counts per subtask. The planner must optionally annotate critical subtasks (e.g., based on graph centrality). This approach may sometimes continue executing a doomed plan (Type II error) but avoids wasteful replanning (Type I error).

Option	Pros	Cons	Chosen?
Replan on Every Failure	Maximally adaptive, quickly abandons flawed plans.	Highly unstable, wastes cycles on transient errors, can lead to infinite replanning loops.	❌
Threshold-Based	Stable, predictable, efficient. Simple to implement and debug.	May persist in a failing plan for too long. Requires defining thresholds and criticality.	✅
Confidence-Guided	Potentially optimal, uses LLM's contextual understanding.	High latency (extra LLM call), complex prompt engineering, can be unreliable.	❌

Common Pitfalls and Mitigations

⚠️ Pitfall: Circular Dependencies in Task Graph

- **Description:** The LLM, when generating subtasks, might inadvertently create a dependency cycle (e.g., "Task A depends on B, and B depends on A"). This makes topological sort impossible and deadlocks execution.
- **Why it's Wrong:** The executor will enter an infinite loop waiting for a task that can never become ready, or the sorting algorithm will fail.
- **Fix: Always run cycle detection** after graph construction. If a cycle is found, feed the error back to the LLM with a request to break the cycle, or implement a heuristic to remove the dependency with the weakest semantic link (e.g., the one not mentioning a shared resource).

⚠️ Pitfall: Over-Decomposition (Nano-Tasks)

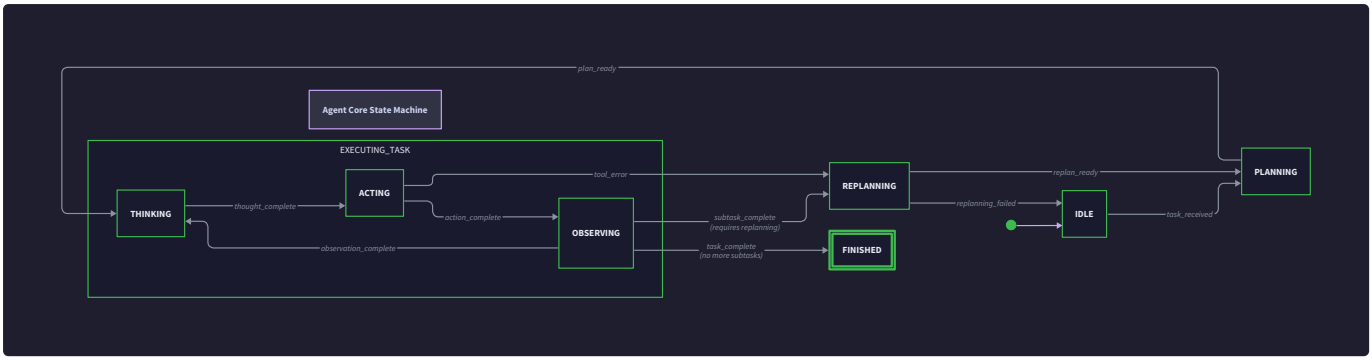
- **Description:** The LLM breaks a simple goal into an excessive number of tiny subtasks (e.g., "Move mouse to icon," "Click mouse," "Wait for dialog," "Read dialog text" for "Open the settings menu"). This creates overhead, consumes context window, and increases the chance of error propagation.
- **Why it's Wrong:** Slows execution, increases token costs, and can make the plan brittle and hard for the LLM to reason about coherently.
- **Fix:** In the planner's prompt, specify **granularity guidelines** (e.g., "Each subtask should correspond to a single, meaningful tool call or a coherent unit of thought"). Implement a post-processing step to merge overly simplistic sequential subtasks that use the same tool.

⚠️ Pitfall: Context Loss Between Subtasks

- **Description:** Subtask B needs information produced by Subtask A, but the executor fails to pass the result of A into B's execution context. The LLM running B is missing critical information.
- **Why it's Wrong:** The agent appears to "forget" what it just did, leading to incorrect actions or repeated work.
- **Fix:** Design a **structured context passing mechanism**. The `Task.context` dictionary is the shared blackboard. The `Executor` must explicitly extract key results from a completed subtask's `ToolResult` (or its `metadata`) and write them into `context` under agreed-upon keys (e.g., `context["research_findings"]`). Subsequent subtasks' `tool_args` should be able to reference these keys via a template syntax (e.g., `{"query": "{research_findings}"}`).

⚠️ Pitfall: Replanning Loops

- **Description:** The agent enters a cycle: Plan fails → Replan → New plan fails in a similar way → Replan again, ad infinitum. Common when the root cause is an unreachable goal or a missing tool.
- **Why it's Wrong:** Consumes resources without making progress. Frustrating for users.
- **Fix:** Implement a **global replanning limit** (e.g., maximum of 3 replans per original task). When the limit is hit, transition to a graceful failure state, inform the user, and suggest a simplified goal. Log the sequence of failed plans for analysis.



Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
DAG Management	Custom in-memory graph using `dict` and `list`. Topological sort via Kahn's algorithm.	Use a graph library like `networkx` for robust cycle detection, centrality analysis, and visualization.
Parallel Execution	`asyncio.gather` for I/O-bound subtask execution (e.g., tool calls).	`concurrent.futures.ThreadPoolExecutor` for CPU-bound tasks or more complex worker pools.
Context Storage	Python `dict` for in-memory, ephemeral task context.	External key-value store (Redis) for persistent, shared context across long-running or distributed agents.

B. Recommended File/Module Structure

```
project-archon/
├── archon/
│   ├── __init__.py
│   ├── agent/
│   │   ├── __init__.py
│   │   ├── agent.py              # Main Agent class, includes Executor logic
│   │   └── react_engine.py       # ReActLoop from Milestone 2
│   ├── planning/
│   │   ├── __init__.py
│   │   ├── planner.py            # Abstract Planner & LLMPlanner
│   │   ├── task.py               # Task, SubTask, Plan dataclasses
│   │   ├── executor.py           # PlanExecutor logic
│   │   └── dag.py                # DAG utilities (topological sort, cycle detect)
│   ├── tools/                    # (From Milestone 1)
│   ├── memory/                   # (From Milestone 4)
│   └── orchestration/            # (From Milestone 5)
└── tests/
    └── test_planning.py
```

C. Infrastructure Starter Code (Complete)

`archon/planning/dag.py` – DAG Utilities

PYTHON

```python
"""
Utilities for Directed Acyclic Graph (DAG) operations used in planning.
"""
from typing import Dict, List, Tuple
from collections import deque

def compute_in_degree(adjacency_list: Dict[str, List[str]]) -> Dict[str, int]:
    """
    Compute the in-degree (number of incoming dependencies) for each node.
    In adjacency_list, {A: [B, C]} means B and C depend on A.
    """
    in_degree = {node: 0 for node in adjacency_list}
    for node, dependents in adjacency_list.items():
        for dep in dependents:
            in_degree[dep] = in_degree.get(dep, 0) + 1
    # Handle nodes that only appear as dependents (leaves in the dependency direction)
    for node in list(in_degree.keys()):
        if node not in adjacency_list:
            adjacency_list[node] = []
    return in_degree

def topological_sort(adjacency_list: Dict[str, List[str]]) -> List[str]:
    """
    Kahn's algorithm for topological sort.
    Returns a list of node IDs in dependency-respecting order.
    Raises ValueError if a cycle is detected.
    """
    in_degree = compute_in_degree(adjacency_list)
    queue = deque([node for node, deg in in_degree.items() if deg == 0])
    sorted_order = []

    while queue:
        node = queue.popleft()
        sorted_order.append(node)
        for dependent in adjacency_list.get(node, []):
            in_degree[dependent] -= 1
            if in_degree[dependent] == 0:
                queue.append(dependent)

    if len(sorted_order) != len(in_degree):
        # Cycle detected
```

```
        raise ValueError("Cycle detected in dependency graph. Cannot compute topological order.")
    return sorted_order

def has_cycle(adjacency_list: Dict[str, List[str]]) -> bool:
    """Check if the graph contains a cycle using DFS."""
    visited = set()
    recursion_stack = set()

    def dfs(node: str) -> bool:
        visited.add(node)
        recursion_stack.add(node)
        for neighbor in adjacency_list.get(node, []):
            if neighbor not in visited:
                if dfs(neighbor):
                    return True
            elif neighbor in recursion_stack:
                return True
        recursion_stack.remove(node)
        return False

    for node in adjacency_list:
        if node not in visited:
            if dfs(node):
                return True
    return False
```

D. Core Logic Skeleton Code

`archon/planning/task.py` – *Data Classes*

PYTHON

```python
"""
Core data structures for planning: Task, SubTask, Plan.
"""
from dataclasses import dataclass, field
from typing import Dict, List, Optional, Any
from datetime import datetime
from enum import Enum

class SubTaskStatus(Enum):
    PENDING = "pending"
    EXECUTING = "executing"
    COMPLETED = "completed"
    FAILED = "failed"
    BLOCKED = "blocked"

@dataclass
class SubTask:
    id: str
    description: str
    expected_output: Optional[str] = None
    tool_name: Optional[str] = None
    tool_args: Optional[Dict[str, Any]] = None
    dependencies: List[str] = field(default_factory=list)
    status: SubTaskStatus = SubTaskStatus.PENDING
    result: Optional[Any] = None  # Will hold a ToolResult

    def with_status(self, new_status: SubTaskStatus) -> 'SubTask':
        """Functional update: returns a new SubTask with updated status."""
        # TODO 1: Create a copy of the current subtask
        # TODO 2: Update the status field of the copy to new_status
        # TODO 3: Return the new subtask
        pass

    def with_result(self, result: Any) -> 'SubTask':
        """Functional update: returns a new SubTask with updated result."""
        # TODO 1: Create a copy of the current subtask
        # TODO 2: Update the result field of the copy to result
        # TODO 3: Return the new subtask
        pass

    def is_ready(self, completed_task_ids: set) -> bool:
```

```
        """
        Check if this subtask can execute (dependencies satisfied).
        completed_task_ids is a set of IDs of subtasks that have status COMPLETED.
        """
        # TODO 1: Check if the subtask status is PENDING or BLOCKED (otherwise not ready)
        # TODO 2: Check if all IDs in self.dependencies are present in completed_task_ids
        # TODO 3: Return True only if both conditions are met
        pass

@dataclass
class Task:
    id: str
    original_query: str
    goal: str
    subtasks: List[SubTask]
    context: Dict[str, Any] = field(default_factory=dict)


    def get_subtask(self, subtask_id: str) -> Optional[SubTask]:
        """Retrieve a subtask by ID."""
        # TODO 1: Iterate through self.subtasks
        # TODO 2: Return the subtask with matching id, or None if not found
        pass


    def update_subtask(self, updated_subtask: SubTask) -> 'Task':
        """Functional update: returns a new Task with the given subtask replaced."""
        # TODO 1: Create a copy of the current task's subtasks list
        # TODO 2: Find the index of the subtask with matching id
        # TODO 3: Replace that element with updated_subtask
        # TODO 4: Return a new Task instance with the updated subtasks list and other fields copied
        pass

@dataclass
class Plan:
    """Executable blueprint derived from a Task."""
    task_id: str
    subtask_dag: Dict[str, List[str]]  # Adjacency list: {A: [B,C]} means B and C depend on A
    execution_order: List[str]  # Topological order of subtask IDs
```

archon/planning/planner.py – *Abstract Planner & LLMPlanner*

PYTHON

```python
"""
Planner interface and LLM-based implementation.
"""
from abc import ABC, abstractmethod
from typing import List, Optional
from .task import Task
from ..tools import Tool, ToolRegistry

class Planner(ABC):
    """Abstract interface for a planner that decomposes goals into tasks."""

    @abstractmethod
    def create_plan(self, goal: str, available_tools: List[Tool], initial_context: Optional[Dict] = None) -> Task:
        """
        Create a Task (with subtasks) from a high-level goal.
        """
        pass

class LLMPlanner(Planner):
    """Planner that uses an LLM to decompose goals."""

    def __init__(self, llm_client, prompt_template: str):
        self.llm = llm_client
        self.prompt_template = prompt_template

    def create_plan(self, goal: str, available_tools: List[Tool], initial_context: Optional[Dict] = None) -> Task:
        """
        1. Format prompt with goal and tool descriptions.
        2. Call LLM to get structured decomposition.
        3. Parse response into SubTask objects.
        4. Build dependency graph and detect cycles.
        5. Create and return a Task.
        """
        # TODO 1: Format the prompt. Include goal, list of tools (name & description).
        # TODO 2: Call self.llm with the prompt. Use a low temperature for structured output.
        # TODO 3: Parse the LLM response. Expect JSON list of subtask dicts.
        # TODO 4: Validate each subtask dict has required fields (id, description, dependencies).
        # TODO 5: Instantiate SubTask objects, setting status to PENDING.
        # TODO 6: Build adjacency list (subtask_dag) from dependencies.
        # TODO 7: Run cycle detection (use has_cycle from dag.py). If cycle, raise error or request re-generation.
        # TODO 8: Generate a unique Task ID (e.g., using uuid).
```

```
        # TODO 9: Create and return a Task object with the subtasks and initial_context.
        pass
```

`archon/agent/agent.py` – *Agent with Executor Logic*

PYTHON

```python
"""
Main Agent class incorporating planning and execution.
"""
import asyncio
from typing import Dict, Optional
from ..planning import Task, SubTask, Planner
from ..tools import ToolRegistry, ToolResult
from .react_engine import ReActAgent

class Agent:
    """The AI Executive: coordinates planning, execution, and memory."""

    def __init__(self, planner: Planner, tool_registry: ToolRegistry, react_agent: ReActAgent):
        self.planner = planner
        self.tool_registry = tool_registry
        self.react_agent = react_agent
        self.state = AgentState.IDLE
        self.current_task: Optional[Task] = None

    async def run_task(self, user_query: str) -> str:
        """Primary public method to execute a user's task."""
        # TODO 1: Set self.state to PLANNING
        # TODO 2: Create a Task using self.planner.create_plan(user_query, ...)
        # TODO 3: Set self.current_task and transition state to EXECUTING
        # TODO 4: Execute the plan by calling self._execute_plan(self.current_task)
        # TODO 5: Aggregate final result from task context or last subtask
        # TODO 6: Set state to FINISHED and return final answer string
        pass

    async def _execute_plan(self, task: Task) -> ToolResult:
        """
        Execute a Task's subtasks in dependency order.
        Implements the threshold-based replanning strategy.
        """
        # TODO 1: Compute execution order (or get from Plan if precomputed).
        # TODO 2: Initialize counters: failure_count dict, replan_count = 0.
        # TODO 3: While there are pending/blocked subtasks and replan_count < MAX_REPLANS:
        #   a. Identify ready subtasks (status PENDING/BLOCKED and dependencies satisfied).
        #   b. For each ready subtask (can be done in parallel with asyncio.gather):
        #       i. Set subtask status to EXECUTING (functional update).
        #       ii. Call self._execute_subtask(subtask, task.context).
```

```
        #        iii. On SUCCESS: update context, mark COMPLETED.
        #        iv. On FAILURE: increment failure_count[subtask.id].
        #             If count >= MAX_RETRIES or subtask is critical: trigger replan.
        #   c. If replan triggered:
        #        i. Increment replan_count.
        #        ii. Call self.planner.create_plan with updated context (and maybe goal).
        #        iii. Replace current task with new plan and reset state.
        # TODO 4: If all subtasks COMPLETED, return success. Else, return error.
        pass


    async def _execute_subtask(self, subtask: SubTask, context: Dict[str, Any]) -> ToolResult:
        """
        Internal method to run the ReAct loop for a single subtask.
        Merges subtask.tool_args with shared context.
        """
        # TODO 1: Prepare query: Use subtask.description as the primary query.
        # TODO 2: Prepare tool_args: Merge context into subtask.tool_args (context overrides defaults?).
        # TODO 3: If subtask.tool_name is set, restrict available_tools to that one for this execution.
        # TODO 4: Call self.react_agent._execute_subtask(...) with the query and args.
        # TODO 5: Return the ToolResult.
        pass
```

E. Language-Specific Hints (Python)

- Use `asyncio.create_task()` and `asyncio.gather(*tasks, return_exceptions=True)` to execute independent subtasks concurrently. Handle exceptions within each gathered task to prevent one failure from stopping others.
- For functional updates on dataclasses, use `dataclasses.replace(existing_obj, field=new_value)` or create a new instance manually. This keeps your data flow predictable and thread-safe.
- When parsing JSON from the LLM, wrap it in a `try/except json.JSONDecodeError` and have a fallback using `extract_json_from_text()` (from Milestone 2) to handle cases where the LLM adds explanatory text around the JSON.
- Store the `failure_count` for replanning in the `Task.context` or a separate dictionary keyed by `subtask.id` to persist across retries.

F. Milestone Checkpoint After implementing the core planning classes and the agent's `run_task` method, you can verify with a simple integration test.

1. **Test Command:**

```
python -m pytest tests/test_planning.py -v
```
BASH

2. **Expected Behavior:** A test that creates a mock `LLMPlanner` (returning a pre-defined `Task` with 3 subtasks and dependencies) and a mock `ReActAgent` (returning successful `ToolResult`s) should:
    - Successfully create a `Task` from a goal string.
    - Build a correct `execution_order` respecting dependencies.
    - Execute subtasks in the correct order.
    - Return a final success result.
3. **Signs of Trouble:**
    - **Subtasks execute in wrong order:** Check your topological sort implementation and the direction of your `subtask_dag`.
    - **Agent gets stuck:** The `is_ready()` check might be incorrect, or a subtask status is not being updated to `COMPLETED`.
    - **Context not passed:** Verify that the `_execute_subtask` method correctly merges `context` into `tool_args` and that results are written back.

> **Milestone(s):** 4

5.4 Component: Memory & Context Management (Milestone 4)

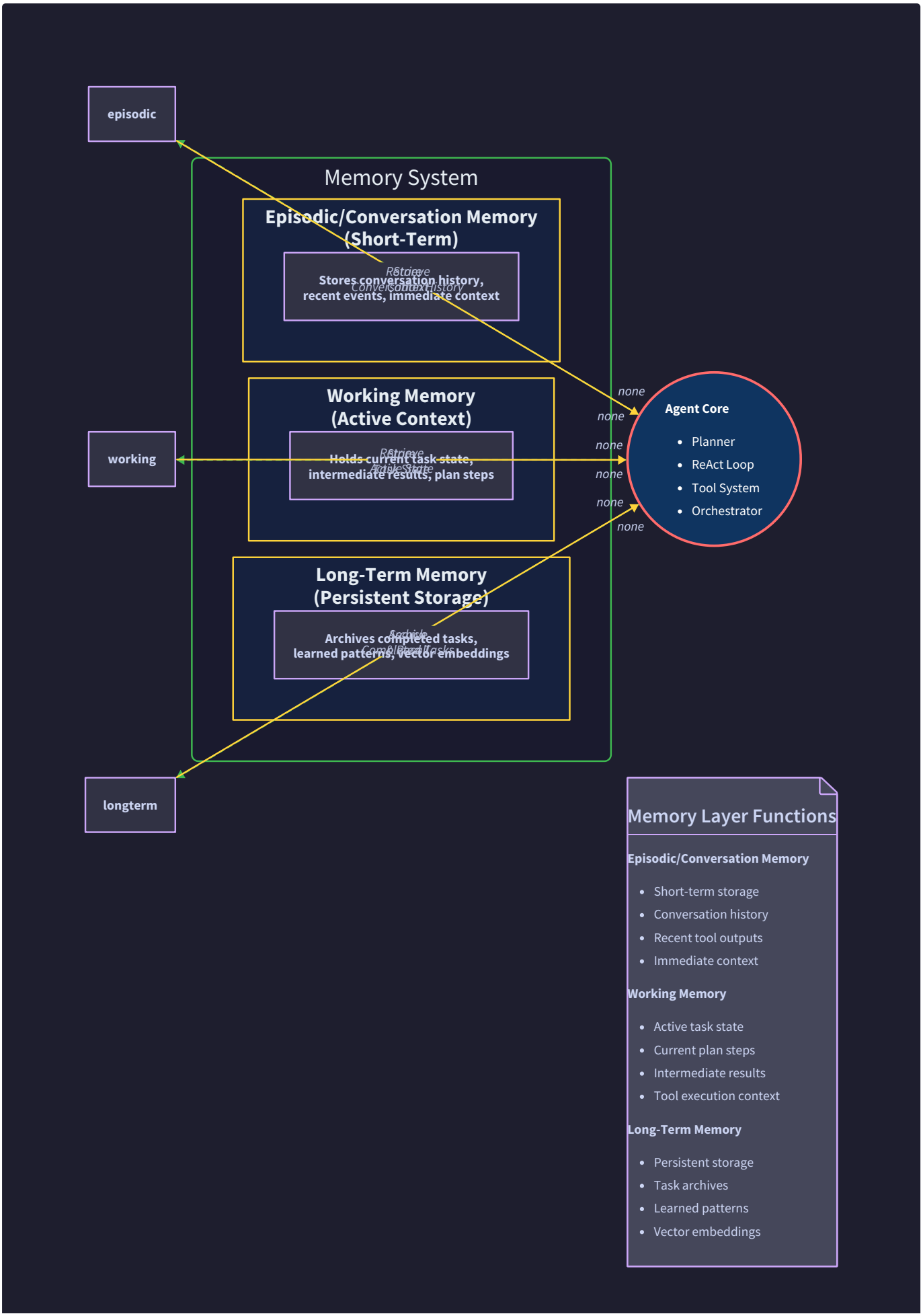
The Memory & Context Management system is the **historical archive and active workspace** of the AI Executive. While the ReAct loop handles immediate reasoning and the Planner builds the roadmap, the Memory system ensures the agent learns from past experiences, retains crucial context across long conversations, and efficiently manages the finite "attention" budget of the LLM's context window. Without sophisticated memory, an agent is amnesic—unable to build upon prior work or maintain coherent state across complex, multi-step tasks.

Mental Model: The Agent's Notebook and Filing Cabinet

Imagine the AI Executive as a seasoned project manager in a physical office. Their memory system operates on three distinct layers, each serving a specific purpose:

1. **Episodic/Conversation Memory (The Open Notebook):** This is the agent's immediate working space—a notebook open on the desk. It contains the *verbatim transcript* of the current conversation: every user query, every thought, every tool result, in chronological order. It's perfect for short-term recall but grows unwieldy for long sessions. Like a notebook, you can flip back to recent pages quickly, but finding a specific fact from hours ago requires a different system.
2. **Working Memory (The Whiteboard):** This is the dynamic, erasable space where the *current task's intermediate state* lives. It holds the agent's current goals, partial results, and contextual variables (e.g., "user's name is Alice", "we are analyzing Q3 financial data"). It's like a whiteboard where the manager jots down the current plan, key numbers, and temporary notes. This memory is highly contextual and typically cleared or updated when a task completes.
3. **Long-Term Memory (The Indexed Filing Cabinet):** This is the agent's institutional knowledge—a vast, organized archive. Past conversations, learned facts, and important outcomes are stored here. Unlike the notebook, you can't scan every document. Instead, you use an **index** (a vector embedding) to retrieve only the most *semantically relevant* memories when starting a new, related task. It's like having a filing cabinet where you can ask, "Find me everything related to 'market analysis in Europe,'" and getting the top 5 relevant reports, regardless of when they were filed.

This three-layer model balances immediacy, relevance, and persistence, allowing the agent to operate effectively across time scales.


episodic
Memory System
Episodic/Conversation Memory (Short-Term)
Stores conversation history, recent events, immediate context
Retrieve
ConversationHistory
Working Memory (Active Context)
working
Holds current task state, intermediate results, plan steps
Retrieve
ActiveState
Long-Term Memory (Persistent Storage)
Archives completed tasks, learned patterns, vector embeddings
Archive
CompletedTasks
none
none
none
none
none
Agent Core
Planner
ReAct Loop
Tool System
Orchestrator
longterm
Memory Layer Functions
Episodic/Conversation Memory
Short-term storage
Conversation history
Recent tool outputs
Immediate context
Working Memory
Active task state
Current plan steps
Intermediate results
Tool execution context
Long-Term Memory
Persistent storage
Task archives
Learned patterns
Vector embeddings

Memory System Interfaces

The memory system is composed of three primary interfaces, each corresponding to one layer of the mental model. They are designed to be swappable, allowing different storage backends while maintaining a consistent API for the Agent Core.

EpisodicMemory Interface

This interface manages the linear sequence of dialogue turns. Its primary responsibility is to maintain order and provide efficient access to recent history.

Method	Parameters	Returns	Description
`append`	`role: str`, `content: str`, `metadata: Optional[Dict[str, Any]] = None`	`None`	Adds a new message to the conversation history. The `role` is typically "user", "assistant", or "system". Metadata can include timestamps, tool names, or other contextual tags.
`get_messages`	`limit: Optional[int] = None`, `since: Optional[int] = 0`	`List[Dict[str, Any]]`	Retrieves the most recent `limit` messages, optionally starting from index `since`. Each message is a dictionary with `role`, `content`, and `metadata` fields.
`clear`	-	`None`	Clears the entire conversation history, typically at the start of a new, unrelated session.
`get_recent`	`max_tokens: int`	`List[Dict[str, Any]]`	A critical method for context window management. Returns the most recent messages whose *total token count* does not exceed `max_tokens`. It may return a partial last message or skip older ones to fit the budget.

Associated Data Structure: While the interface returns generic dictionaries, the internal storage often uses a `MemoryEntry`.

Field	Type	Description
`id`	`str`	Unique identifier for the memory entry (e.g., UUID).
`content`	`str`	The textual content of the memory (the message text).
`timestamp`	`datetime`	When this memory was recorded.
`metadata`	`Dict[str, Any]`	Flexible bag of properties (e.g., `{"role": "assistant", "tool_used": "web_search"}`).
`embedding`	`Optional[List[float]]`	**For long-term memory only.** The vector embedding of the content, used for semantic search. For episodic memory, this is usually `None`.

WorkingMemory Interface

This interface manages the task-scoped, mutable state. It acts like a key-value store for the current execution context.

Method	Parameters	Returns	Description
`get`	`key: str`, `default: Any = None`	`Any`	Retrieves a value stored under the given `key`. Returns `default` if the key doesn't exist.
`set`	`key: str`, `value: Any`	`None`	Stores a `value` under the given `key`. Overwrites any existing value.
`update`	`mapping: Dict[str, Any]`	`None`	Batch updates multiple key-value pairs from the provided dictionary.
`clear`	-	`None`	Clears all key-value pairs, typically invoked when a task is completed or a new top-level goal is received.
`to_dict`	-	`Dict[str, Any]`	Returns a copy of the entire working memory as a dictionary, useful for serialization or debugging.

LongTermMemory Interface

This interface provides semantic search over a persistent store of past memories. It is the most complex layer, involving embeddings and vector similarity.

Method	Parameters	Returns	Description
`store`	`content: str`, `metadata: Optional[Dict[str, Any]] = None`	`str` (ID)	Stores a piece of information (e.g., a conversation summary or a key fact). The system should automatically generate an embedding for the `content`. Returns a unique ID for the stored memory.
`query`	`query_text: str`, `limit: int = 5`, `threshold: Optional[float] = None`	`List[MemoryEntry]`	The core retrieval operation. Takes a query string, converts it to an embedding, and performs a similarity search against stored memories. Returns the top-`limit` most relevant `MemoryEntry` objects, optionally filtered by a similarity score `threshold`.
`delete`	`memory_id: str`	`bool`	Deletes the memory with the given ID. Returns `True` if successful.
`summarize_recent`	`max_memories: int = 100`	`str`	An optional but valuable method. Creates a concise textual summary of the most recent `max_memories` entries. This summary can then be `store`d as a new memory, compressing history.

Memory Retrieval and Compression

The true power of the memory system lies not just in storage, but in the intelligent processes of **retrieval** (finding the right memories at the right time) and **compression** (managing the context window limit). These processes form a continuous cycle during agent operation.

Memory Retrieval Algorithm

When the Agent Core prepares a prompt for the LLM (e.g., within `ReActAgent._format_react_prompt`), it must enrich the prompt with relevant context. This follows a multi-source retrieval strategy:

1. **Extract Query Keywords:** From the current `ReActLoopContext` (specifically the `subtask_description` and recent conversation turns), extract a set of keywords or rephrase the current goal into a concise query string (e.g., "current user question: {subtask_description}").
2. **Query Long-Term Memory:** Pass the query string to `LongTermMemory.query()`. This performs a vector similarity search, returning a list of semantically related past `MemoryEntry` objects.
3. **Filter and Format:** Process the retrieved memories:
   - Remove duplicates or extremely similar entries.
   - Format each memory into a natural language string suitable for inclusion in the prompt (e.g., `"[Memory from past session] User asked about Python decorators. We explained the @ syntax and gave examples."`).
   - Optionally, include relevance scores or timestamps as metadata in the formatted text.
4. **Retrieve Recent Episodic Memory:** Call `EpisodicMemory.get_recent(max_tokens=recent_budget)` to get the most recent turns of the current conversation. This provides immediate continuity.
5. **Inject Working Memory:** Serialize the contents of `WorkingMemory.to_dict()` into a structured string (e.g., `"Current Context: user_name=Alice, analysis_target=Q3_sales"`).
6. **Assemble Context Window:** The final prompt is assembled by concatenating, in order: a. **System Instructions** (fixed). b. **Working Memory Context**. c. **Retrieved Long-Term Memories** (if any). d. **Recent Episodic Memory**. e. **Current Step Instructions** (e.g., the ReAct template).

   The total token count of (a) through (e) must be less than the LLM's context limit, leaving room for the LLM's response.

Memory Compression and Eviction Strategy

As conversations grow long, the episodic memory buffer will eventually exceed the token budget. A naive "last-N" truncation risks losing critical information from earlier in the dialogue. A more sophisticated strategy involves compression:

1. **Monitor Token Usage:** After each agent turn, calculate the total tokens of the full episodic history.
2. **Trigger Compression:** When the history exceeds a safe threshold (e.g., 70% of the context window), initiate compression.
3. **Summarize Old Turns:** Select the oldest contiguous segment of conversation (e.g., the first 10 turns) that hasn't been summarized yet.
4. **Generate Summary:** Use the LLM itself (with a dedicated prompt) to produce a concise, third-person summary of that segment (e.g., "The user introduced themselves as Bob and asked for an overview of project management methodologies. The agent explained Agile and Waterfall.").
5. **Store and Replace:**
   - `store` the generated summary in **LongTermMemory**, ensuring it's available for future retrieval.
   - Replace the original, verbose conversation turns in **EpisodicMemory** with a single *pointer message*, like `"[Earlier conversation summarized: {summary_text}]"`. This pointer is much shorter but preserves the semantic gist.
6. **Repeat:** Continue this process, sliding the compression window forward, as the conversation grows.

> **Design Insight:** This compression strategy transforms the memory system from a passive log into an active, lossy compressor. It prioritizes *recent detail* and *summarized relevance* over *complete verbatim history*, which aligns well with how human memory works and is a practical necessity given token limits.

Storage Backends and Choices

Each memory layer can be backed by different storage technologies, depending on the required persistence, performance, and complexity.

Layer	Simple Option (Development/Prototyping)	Advanced Option (Production)	Rationale
Episodic Memory	**In-memory list (`list`)**. Fast, no setup, but lost on restart.	**Persistent queue (Redis, SQLite)**. Survives restarts, enables session resumption.	Episodic memory is transient by nature (per session). In-memory is sufficient for most single-session agent tasks. Persistence is needed for long-running or stateful applications.
Working Memory	**In-memory dictionary (`dict`)**. Simple, fast, co-located with agent state.	**Shared key-value store (Redis)**. Allows multiple agents or processes to share context.	Working memory is inherently transient and task-scoped. A `dict` is perfectly adequate unless you need cross-process or cross-agent coordination.
Long-Term Memory	**In-memory vector store with sentence-transformers (FAISS)**. Runs entirely in-process, good for demos.	**Dedicated vector database (Chroma, Weaviate, Pinecone)**. Offers persistence, scalability, advanced query features, and easier management of large memory corpora.	The core challenge is semantic search. FAISS + sentence-transformers is a powerful, standalone combo. A dedicated DB simplifies operations at scale and provides better tooling for updates and deletions.

Embedding Model Choice: The quality of long-term memory retrieval hinges on the embedding model. For an all-Python stack, the `sentence-transformers` library (e.g., `all-MiniLM-L6-v2`) provides excellent quality and speed locally. For maximum compatibility with LLM APIs, using the same provider's embedding API (e.g., OpenAI's `text-embedding-ada-002`) ensures consistency but adds latency and cost.

ADR: Context Window Management Strategy

> **Decision: Hybrid Sliding Window with LLM-Based Summarization for Episodic Memory**

- **Context:** LLMs have a fixed context window (e.g., 4K, 8K, 128K tokens). The agent's conversation history, tool outputs, and retrieved memories can easily exceed this limit. We need a strategy to decide what information to include in each prompt to the LLM without losing critical context.
- **Options Considered:**
  1. **Naive Truncation (Keep Last N Tokens):** Always include the most recent N tokens from the episodic memory, discarding the oldest.
  2. **Recursive Summarization:** After each agent turn, use the LLM to summarize the entire conversation so far into a fixed-length summary. Use only this summary as context for the next turn.
  3. **Hybrid Sliding Window with Summarization:** Maintain a sliding window of recent, verbatim turns. When the window is full, summarize the oldest segment, store the summary in long-term memory, and replace the original turns with a short pointer.
- **Decision: Option 3 (Hybrid Sliding Window with Summarization)**.
- **Rationale:**
  - **Option 1** is simple but suffers from **catastrophic forgetting**. Crucial details established early in a long task (e.g., "the user's name is Alice") are permanently lost once pushed out of the window.
  - **Option 2** maintains a very compact state but is **lossy by construction**. Every summarization step discards information, and errors in summarization compound. It also requires an LLM call on *every* turn, increasing cost and latency.
  - **Option 3** offers the best balance. It preserves **high fidelity for recent interactions** (critical for step-by-step reasoning), while **conserving the semantic essence of older interactions** via summaries stored in long-term memory. The summaries are retrievable via semantic search if they become relevant again. The summarization cost is amortized over many turns, only paid when necessary.
- **Consequences:**
  - **Enables:** Long-running, coherent dialogues that can reference early details via summary recall. Efficient use of the context window.
  - **Trade-offs:** Increased implementation complexity. Introduces a delay during summarization steps. Requires careful prompt engineering for the summarization task to avoid losing important nuance.

Option	Pros	Cons	Chosen?
Naive Truncation	Extremely simple, zero overhead.	Catastrophic forgetting, poor performance on long tasks.	❌
Recursive Summarization	Constant, predictable context size.	Highly lossy, compounding errors, high latency/cost.	❌
Hybrid Sliding Window	Balances recency and persistence, scalable.	Complex to implement, requires summarization prompts.	✅

Common Pitfalls and Mitigations

⚠️ Pitfall: Unbounded Memory Growth Without Cleanup

- **Description:** Storing every interaction in episodic or long-term memory indefinitely without any eviction or summarization policy.
- **Why It's Wrong:** Leads to ballooning memory usage, slowing down retrievals (especially for vector searches over huge datasets), and eventually hitting storage limits. It also pollutes the memory with irrelevant, outdated information, reducing retrieval quality.
- **Fix:** Implement **automatic summarization and eviction policies**. For episodic memory, use the hybrid sliding window. For long-term memory, implement a **least-recently-used (LRU) eviction policy** or **time-based expiration** for certain types of memories. Periodically run a cleanup job that removes low-importance or very old entries.

⚠️ Pitfall: Retrieved Memories Are Not Relevant to the Current Task

- **Description:** The vector search returns past memories that are semantically similar at the word level but not contextually useful (e.g., retrieving a memory about "Python" when the task is about "snakes").
- **Why It's Wrong:** Irrelevant memories waste precious context window tokens and can confuse the LLM, leading to incoherent or off-topic responses.
- **Fix: Improve the query formulation.** Don't just use the current subtask description. Augment the query with metadata filters (e.g., only retrieve memories from the same user or session) or use a more sophisticated query expansion technique (e.g., generate multiple query aspects from the current context). **Tune the similarity threshold** to be more restrictive.

⚠️ Pitfall: Lossy Summarization Discards Critical Details

- **Description:** The LLM-based summarization step oversimplifies or omits key facts, numbers, or instructions from the original conversation.
- **Why It's Wrong:** When the agent later relies on the summary, it may act on incorrect or incomplete information, leading to task failure.
- **Fix: Craft a detailed summarization prompt.** Explicitly instruct the LLM to preserve specific types of information: names, numbers, dates, URLs, and explicit user instructions. **Store critical facts separately.** Extract key entities (using an NER tool or the LLM) and store them as structured key-value pairs in working memory or a dedicated fact store, rather than relying solely on the narrative summary.

⚠️ Pitfall: High Latency from Vector Database Calls

- **Description:** Every call to `LongTermMemory.query()` involves a network round-trip to an external vector database, adding hundreds of milliseconds to each agent cycle.
- **Why It's Wrong:** Makes the agent feel sluggish and severely impacts performance for tasks requiring many reasoning cycles.
- **Fix: Implement caching.** Cache the results of frequent or similar queries in-memory. **Batch retrievals.** If the agent's plan is known, pre-fetch potentially relevant memories at the start of a task phase. **Use a local vector store** (FAISS) for development and smaller deployments to eliminate network latency entirely.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Episodic Memory Backend	Python `list` in memory	Redis (with `redis-py`) or SQLite (`sqlite3`)
Working Memory Backend	Python `dict` in memory	Redis (for multi-agent)
Long-Term Memory Backend	FAISS + Sentence-Transformers	Chroma DB or Weaviate
Embedding Model	`sentence-transformers/all-MiniLM-L6-v2`	OpenAI `text-embedding-3-small` API

B. Recommended File/Module Structure

```
project_archon/
├── core/
│   ├── agent.py              # Main Agent class (uses memory interfaces)
│   └── react_loop.py         # ReActAgent (calls memory during prompt formatting)
├── memory/
│   ├── __init__.py
│   ├── base.py               # Abstract base classes (EpisodicMemory, WorkingMemory, LongTermMemory)
│   ├── episodic.py           # Concrete implementations (ListEpisodicMemory, RedisEpisodicMemory)
│   ├── working.py            # DictWorkingMemory, RedisWorkingMemory
│   ├── long_term.py          # FAISSMemory, ChromaMemory
│   ├── retriever.py          # MemoryRetriever (orchestrates multi-source retrieval)
│   ├── summarizer.py         # MemorySummarizer (handles compression)
│   └── models.py             # MemoryEntry data class
└── utils/
    └── token_counter.py      # Utility to estimate token counts (using tiktoken or transformers)
```

C. Infrastructure Starter Code

`memory/models.py` - Core Data Class

```python
from dataclasses import dataclass, field

from datetime import datetime

from typing import Any, Dict, List, Optional

import uuid


@dataclass

class MemoryEntry:

    """A single unit of memory, used primarily for long-term storage."""

    id: str = field(default_factory=lambda: str(uuid.uuid4()))

    content: str = ""

    timestamp: datetime = field(default_factory=datetime.now)

    metadata: Dict[str, Any] = field(default_factory=dict)

    embedding: Optional[List[float]] = None


    def to_dict(self) -> Dict[str, Any]:

        """Serialize to dictionary, converting datetime to ISO string."""

        return {

            "id": self.id,

            "content": self.content,

            "timestamp": self.timestamp.isoformat(),

            "metadata": self.metadata,

            "embedding": self.embedding

        }


    @classmethod

    def from_dict(cls, data: Dict[str, Any]) -> "MemoryEntry":

        """Deserialize from dictionary."""

        # Handle timestamp which might be string or datetime

        ts = data.get("timestamp")

        if isinstance(ts, str):

            ts = datetime.fromisoformat(ts)

        return cls(

            id=data.get("id", str(uuid.uuid4())),

            content=data.get("content", ""),

            timestamp=ts or datetime.now(),

            metadata=data.get("metadata", {}),

            embedding=data.get("embedding")

        )
```

PYTHON

`memory/base.py` - **Abstract Interfaces**

PYTHON

```python
from abc import ABC, abstractmethod

from typing import Any, Dict, List, Optional


class EpisodicMemory(ABC):

    """Abstract base class for storing conversation history."""



    @abstractmethod

    def append(self, role: str, content: str, metadata: Optional[Dict[str, Any]] = None) -> None:

        """Add a message to history."""

        pass



    @abstractmethod

    def get_messages(self, limit: Optional[int] = None, since: Optional[int] = 0) -> List[Dict[str, Any]]:

        """Retrieve messages."""

        pass



    @abstractmethod

    def get_recent(self, max_tokens: int) -> List[Dict[str, Any]]:

        """Get most recent messages within token budget."""

        pass



    def clear(self) -> None:

        """Clear all history. Can have a default implementation."""

        pass


class WorkingMemory(ABC):

    """Abstract base class for task-scoped key-value state."""



    @abstractmethod

    def get(self, key: str, default: Any = None) -> Any:

        pass



    @abstractmethod

    def set(self, key: str, value: Any) -> None:

        pass



    def update(self, mapping: Dict[str, Any]) -> None:

        """Default implementation for batch update."""

        for k, v in mapping.items():

            self.set(k, v)
```

```python
    def clear(self) -> None:
        """Can be overridden if needed."""
        pass

    def to_dict(self) -> Dict[str, Any]:
        """Should return a copy of all key-value pairs."""
        pass

class LongTermMemory(ABC):
    """Abstract base class for semantic memory storage and retrieval."""

    @abstractmethod
    def store(self, content: str, metadata: Optional[Dict[str, Any]] = None) -> str:
        """Store content, generate embedding, return ID."""
        pass

    @abstractmethod
    def query(self, query_text: str, limit: int = 5, threshold: Optional[float] = None) -> List[MemoryEntry]:
        """Retrieve relevant memories for query."""
        pass

    @abstractmethod
    def delete(self, memory_id: str) -> bool:
        pass

    def summarize_recent(self, max_memories: int = 100) -> str:
        """Optional. Should summarize recent entries."""
        raise NotImplementedError("Summarization not implemented for this backend")
```

D. Core Logic Skeleton Code

`memory/episodic.py` - ListEpisodicMemory with Token-Aware Truncation

```python
from typing import Any, Dict, List, Optional

from .base import EpisodicMemory

from ..utils.token_counter import estimate_tokens  # Assume this utility exists


class ListEpisodicMemory(EpisodicMemory):

    """In-memory episodic memory using a list, with token-aware retrieval."""



    def __init__(self):

        self.messages: List[Dict[str, Any]] = []



    def append(self, role: str, content: str, metadata: Optional[Dict[str, Any]] = None) -> None:

        self.messages.append({

            "role": role,

            "content": content,

            "metadata": metadata or {}

        })



    def get_messages(self, limit: Optional[int] = None, since: Optional[int] = 0) -> List[Dict[str, Any]]:

        # TODO 1: Slice messages starting from index `since`

        # TODO 2: If `limit` is provided, return at most that many messages

        # TODO 3: Return a COPY of the message dictionaries to prevent accidental mutation

        pass



    def get_recent(self, max_tokens: int) -> List[Dict[str, Any]]:

        # TODO 1: Start from the most recent message and work backwards

        # TODO 2: Keep a running total of estimated tokens (use `estimate_tokens(msg["content"])`)

        # TODO 3: Add messages to the result list until adding the next message would exceed `max_tokens`

        # TODO 4: If adding a message would exceed the limit, you may optionally truncate its content

        #         or skip it entirely (simpler: skip it).

        # TODO 5: Return the messages in chronological order (oldest first), so reverse the collected list.

        pass



    def clear(self) -> None:

        self.messages.clear()
```

`memory/long_term.py` - FAISS-based Long-Term Memory (Skeleton)

PYTHON

```python
import numpy as np

from typing import Any, Dict, List, Optional

from sentence_transformers import SentenceTransformer  # pip install sentence-transformers

import faiss  # pip install faiss-cpu


from .base import LongTermMemory

from .models import MemoryEntry


class FAISSMemory(LongTermMemory):

    """Long-term memory using FAISS for vector similarity search and SentenceTransformer for embeddings."""


    def __init__(self, model_name: str = "all-MiniLM-L6-v2"):

        # TODO 1: Initialize the embedding model: self.model = SentenceTransformer(model_name)

        # TODO 2: Initialize FAISS index: self.index = faiss.IndexFlatL2(dimension)

        #         - Need to get dimension from model: model.get_sentence_embedding_dimension()

        # TODO 3: Maintain a list of MemoryEntry objects parallel to the index: self.memories = []

        # TODO 4: Maintain an ID to index mapping: self.id_to_index = {}

        pass


    def store(self, content: str, metadata: Optional[Dict[str, Any]] = None) -> str:

        # TODO 1: Create a MemoryEntry with the content and metadata

        # TODO 2: Generate embedding: embedding = self.model.encode(content)

        # TODO 3: Add embedding to FAISS index: self.index.add(np.array([embedding]))

        # TODO 4: Append the MemoryEntry to self.memories (store embedding in entry.embedding)

        # TODO 5: Update id_to_index mapping

        # TODO 6: Return the MemoryEntry.id

        pass


    def query(self, query_text: str, limit: int = 5, threshold: Optional[float] = None) -> List[MemoryEntry]:

        # TODO 1: Generate embedding for query_text

        # TODO 2: Search FAISS index: distances, indices = self.index.search(query_embedding, k=limit)

        # TODO 3: Convert distances to similarity scores (e.g., 1/(1+distance) or cosine similarity if using inner product)

        # TODO 4: For each (distance, idx) pair, get the corresponding MemoryEntry from self.memories

        # TODO 5: If threshold is provided, filter out entries with similarity below threshold

        # TODO 6: Return list of MemoryEntry objects (sorted by relevance)

        pass


    def delete(self, memory_id: str) -> bool:

        # TODO 1: Find the index of the memory with given ID using id_to_index

        # TODO 2: If not found, return False

        # TODO 3: Mark the memory as deleted (e.g., set a flag in metadata or move to separate list)
```

```
        #         Note: FAISS doesn't support easy deletion; common pattern is to rebuild index or use a mask.
        #         For simplicity, we can just mark it and filter during query.
        # TODO 4: Update id_to_index for remaining entries
        # TODO 5: Return True
        pass
```

`memory/retriever.py` - MemoryRetriever (Orchestrator)

PYTHON

```python
from typing import List, Dict, Any

from .base import EpisodicMemory, WorkingMemory, LongTermMemory


class MemoryRetriever:

    """Orchestrates retrieval from all memory layers to build context for the LLM prompt."""



    def __init__(self,

                 episodic: EpisodicMemory,

                 working: WorkingMemory,

                 long_term: LongTermMemory):

        self.episodic = episodic

        self.working = working

        self.long_term = long_term



    def retrieve_context(self,

                         current_query: str,

                         token_budget: int) -> Dict[str, str]:

        """Retrieve and format context from all memory layers.



        Returns a dictionary with keys like 'working', 'long_term', 'recent'

        containing formatted strings ready for prompt injection.

        """

        # TODO 1: Format working memory

        #   working_context = "Current Context:\n"

        #   for k, v in self.working.to_dict().items():

        #       working_context += f"- {k}: {v}\n"



        # TODO 2: Query long-term memory

        #   lt_memories = self.long_term.query(current_query, limit=3)

        #   long_term_context = "\n".join([f"[Memory] {m.content}" for m in lt_memories])



        # TODO 3: Retrieve recent episodic memory within remaining token budget

        #   Reserve tokens for working and long-term context already formatted

        #   remaining_tokens = token_budget - estimate_tokens(working_context + long_term_context)

        #   recent_messages = self.episodic.get_recent(max_tokens=remaining_tokens)

        #   recent_context = format_messages_as_conversation(recent_messages)



        # TODO 4: Return dictionary

        #   return {

        #       "working": working_context,

        #       "long_term": long_term_context,
```

```
        #       "recent": recent_context
        #   }
        pass
```

E. Language-Specific Hints (Python)

- **Token Counting:** Use the `tiktoken` library for OpenAI models or `transformers` for open-source models to accurately count tokens. Create a utility function `estimate_tokens(text: str) -> int`.
- **FAISS Installation:** For most setups, `pip install faiss-cpu` is sufficient. For GPU acceleration, use `faiss-gpu` but note the compatibility challenges.
- **SentenceTransformers Caching:** The first call to `SentenceTransformer.encode()` downloads the model (if not cached). Specify `cache_folder` in the constructor to control where models are stored.
- **Thread Safety:** The in-memory implementations (`ListEpisodicMemory`, `DictWorkingMemory`) are not thread-safe. If your agent runs in a multi-threaded environment (e.g., handling multiple requests), use threading locks (`threading.RLock`) or switch to a thread-safe backend like Redis.

F. Milestone Checkpoint

After implementing the core memory components, verify the system with a simple test script:

```python
# test_memory.py
from memory.episodic import ListEpisodicMemory
from memory.working import DictWorkingMemory
from memory.long_term import FAISSMemory

episodic = ListEpisodicMemory()
working = DictWorkingMemory()
long_term = FAISSMemory()

# Test episodic memory
episodic.append("user", "Hello, my name is Alice.")
episodic.append("assistant", "Hi Alice! How can I help?")
print("Episodic recent:", episodic.get_recent(max_tokens=50))

# Test working memory
working.set("user_name", "Alice")
working.set("current_task", "research")
print("Working memory:", working.to_dict())

# Test long-term memory (may take a moment to load model)
mem_id = long_term.store("Alice prefers summaries in bullet points.")
results = long_term.query("user preferences", limit=1)
print("Long-term query result:", results[0].content if results else "None")
```

Expected Behavior: The script should run without errors. The episodic memory should return the two messages (or a truncated version if token limit is very low). The working memory should print the dictionary. The long-term memory should store and retrieve the memory about Alice's preference.

Signs of Trouble:

- **FAISS import error:** Check installation (`pip install faiss-cpu`).
- **SentenceTransformer download hanging:** Check internet connection or set `cache_folder`.
- **No long-term results returned:** The query might not be similar enough; try querying with "bullet points" instead.

5.5 Component: Multi-Agent Collaboration (Milestone 5)

> **Milestone(s):** 5

The Multi-Agent Collaboration component transforms the framework from a solo executive into a coordinated team. While a single agent can handle many tasks, complex real-world problems often require specialized expertise working in concert—much like a software project needs developers, designers, and testers collaborating. This component provides the communication protocols, role management, and orchestration patterns that enable multiple AI agents to work together toward a common goal, distributing cognitive load and leveraging specialized capabilities.

Mental Model: A Corporate Department

Imagine a corporate department working on a complex project. There's a **department manager** (the orchestrator) who receives the high-level objective from leadership. The manager breaks the project down into specialized tasks and assigns them to team members based on their roles: a **researcher** gathers information, a **writer** drafts content, a **software engineer** writes code, and a **quality analyst** tests the results. Team members communicate through structured channels (email, meetings, shared documents), report progress back to the manager, and occasionally collaborate directly when their work intersects. The manager tracks overall progress, resolves conflicts, and integrates everyone's contributions into the final deliverable.

In our framework, this mental model maps directly:

- **Orchestrator Agent** = Department manager: oversees the overall task, decomposes work, assigns subtasks, and synthesizes final results
- **Specialized Worker Agents** = Team members: each has a specific role (Researcher, Writer, Coder, Validator) with a tailored set of tools and expertise
- **Communication Protocol** = Corporate communication channels: structured messages that pass requests, data, and results between agents
- **Shared Context** = Shared documents and meetings: common workspace where agents can access relevant information and intermediate outputs
- **Role Assignment** = Job descriptions and organizational chart: defines each agent's capabilities and responsibilities

This corporate metaphor helps visualize how autonomy and coordination balance each other: each agent operates independently within its domain while remaining aligned toward the shared objective through clear communication and managerial oversight.

Communication Protocol and Roles

Effective collaboration requires a **structured communication protocol** that defines how agents exchange information. Unlike human teams that can interpret vague instructions, AI agents need unambiguous message formats with clear semantics. The protocol must support request-response patterns, broadcast announcements, and error notifications while maintaining context across conversations.

Message Structure

Every inter-agent communication uses the `Message` data structure, which provides the necessary metadata for routing, correlation, and interpretation:

Field	Type	Description
`sender_id`	`str`	Unique identifier of the sending agent (e.g., "orchestrator", "researcher_agent_1")
`receiver_id`	`str`	Identifier of the intended recipient agent, or `"broadcast"` for all agents
`message_type`	`str`	Semantic type of message, determines how content is interpreted. Common types: `"task_request"`, `"task_response"`, `"status_update"`, `"error"`, `"information_share"`
`content`	`Dict[str, Any]`	Structured payload of the message. Format varies by `message_type` but must be JSON-serializable
`timestamp`	`datetime`	When the message was created, used for ordering and timeouts
`correlation_id`	`Optional[str]`	Links a response to its original request, enabling request-response tracking

For `message_type = "task_request"`, the content dictionary typically includes:

- `"task_id"`: Unique identifier for this task instance
- `"subtask_description"`: Natural language description of what needs to be done
- `"expected_output"`: Optional specification of what format the result should take
- `"deadline"`: Optional timestamp by when the task should be completed
- `"context"`: Relevant background information or prerequisites

For `message_type = "task_response"`, the content includes:

- `"task_id"`: Echoes the original task identifier
- `"result"`: The `ToolResult` object (or its serialized form) containing the output
- `"status"`: Completion status (e.g., "success", "partial", "failed")

Role Assignment and Capability Advertisement

Each agent has a defined **Role** that dictates its specialization and advertised capabilities. A role is more than just a label—it defines:

1. **Tool Set**: Which tools the agent has access to (a Researcher agent might have web search, database query, and summarization tools)
2. **Prompt Templates**: Role-specific instructions that shape how the agent interprets tasks (a Writer agent might be prompted to focus on clarity and structure)
3. **Communication Patterns**: Which message types the agent can send/receive and to whom
4. **Performance Characteristics**: Expected latency, cost constraints, or quality trade-offs

The role assignment is typically configured at agent creation and registered with the orchestrator:

Agent ID	Assigned Role	Advertised Capabilities (Tools)	Special Instructions
`agent_researcher_1`	`"Researcher"`	`["web_search", "wikipedia_lookup", "document_summarizer"]`	"Focus on factual accuracy, cite sources"
`agent_writer_1`	`"Writer"`	`["draft_outline", "write_section", "proofread"]`	"Maintain consistent tone, avoid jargon"
`agent_coder_1`	`"Software Engineer"`	`["code_generator", "code_executor", "debugger"]`	"Write clean, commented code with error handling"
`agent_validator_1`	`"Quality Analyst"`	`["unit_test_generator", "code_reviewer", "fact_checker"]`	"Be thorough, identify edge cases"

> **Key Insight**: Role-based specialization reduces cognitive load on individual agents. Instead of every agent needing to master all tools, each focuses on a domain, leading to more reliable tool use and less hallucination of inappropriate actions. The orchestrator's job is to match tasks to roles, not to micromanage how each agent accomplishes its work.

Orchestration and Delegation Flow

The orchestrator agent coordinates the multi-agent system using a well-defined flow that balances autonomy with oversight. This flow ensures tasks are completed efficiently while maintaining coherence across contributions.

Step-by-Step Orchestration Algorithm

1. **Task Reception & Analysis**
   - The orchestrator receives a user query through its `Agent.run_task()` method
   - It analyzes the query to determine if multi-agent collaboration is needed (complexity threshold, multiple domains involved)
   - If single-agent suffices, it processes the task directly using its internal ReAct loop
2. **Task Decomposition & Role Matching**
   - The orchestrator uses its `Planner.create_plan()` method to break the complex task into subtasks
   - For each subtask, it identifies the required capabilities (tools needed)
   - It matches these capability requirements against registered agent roles to assign each subtask to the most suitable agent
   - Subtasks with dependencies are linked in a **task dependency graph**
3. **Task Distribution & Communication**
   - The orchestrator sends `"task_request"` messages to the assigned agents, including:
     - The subtask description and expected output format
     - Any context or prerequisites (e.g., "use the research results from agent_researcher_1")
     - Deadline and priority information
   - Each message includes a unique `correlation_id` for tracking
4. **Parallel Execution with Monitoring**
   - Worker agents receive their tasks and execute them independently using their own ReAct loops
   - Agents may communicate directly with each other for coordination (e.g., a writer requesting clarification from a researcher)
   - The orchestrator monitors progress through:
     - Periodic `"status_update"` messages from workers
     - Timeout tracking for each outstanding task
     - Watching for `"error"` messages indicating problems
5. **Result Collection & Conflict Resolution**
   - As workers complete tasks, they send `"task_response"` messages back to the orchestrator

- The orchestrator validates each result against the expected output criteria
- If results conflict (e.g., researcher and validator disagree on a fact), the orchestrator:
  - Applies predefined conflict resolution strategies (majority vote, confidence scoring, or requesting human input)
  - May trigger additional verification tasks
- Results are stored in a **shared context** accessible to all agents working on related subtasks

6. **Synthesis & Final Output Generation**
   - Once all subtasks are complete, the orchestrator synthesizes the individual results
   - It may perform final integration tasks (formatting, consistency checks, executive summary)
   - The orchestrator returns the final result through `Agent.run_task()`
7. **Cleanup & Learning**
   - The orchestrator archives the complete interaction for future reference
   - Performance metrics are recorded (task completion time, success rates per agent)
   - If configured, the system updates agent role assignments or tool sets based on observed performance

Shared Context Management

A critical challenge in multi-agent systems is maintaining **shared context**—ensuring all agents working on related aspects have access to relevant information without overwhelming them. Our framework implements a hybrid approach:

- **Orchestrator-Managed Context**: The orchestrator maintains a central `Dict[str, Any]` of key findings, decisions, and artifacts
- **Selective Broadcast**: When an agent produces a result relevant to others, the orchestrator broadcasts an `"information_share"` message with the essential data
- **On-Demand Querying**: Agents can request specific context elements from the orchestrator via `"context_request"` messages
- **Versioning**: Context items are timestamped and versioned to prevent stale information from causing conflicts

This approach avoids the bottleneck of requiring all communication to flow through the orchestrator while still maintaining coherence.

Common Orchestration Patterns

Different collaboration scenarios call for different orchestration patterns. Our framework supports three primary patterns that can be mixed and matched.

1. Hierarchical (Manager-Worker) Pattern

The most common pattern, matching our corporate department metaphor:

- **Structure**: Single orchestrator at the root, multiple specialized workers as leaves
- **Communication Flow**: All task assignments flow from orchestrator to workers; all results flow back to orchestrator
- **Coordination**: Workers typically don't communicate directly with each other
- **Best For**: Well-structured tasks with clear decomposition, centralized quality control
- **Example**: Writing a technical report (orchestrator → researcher → writer → validator)

2. Broadcast (Committee) Pattern

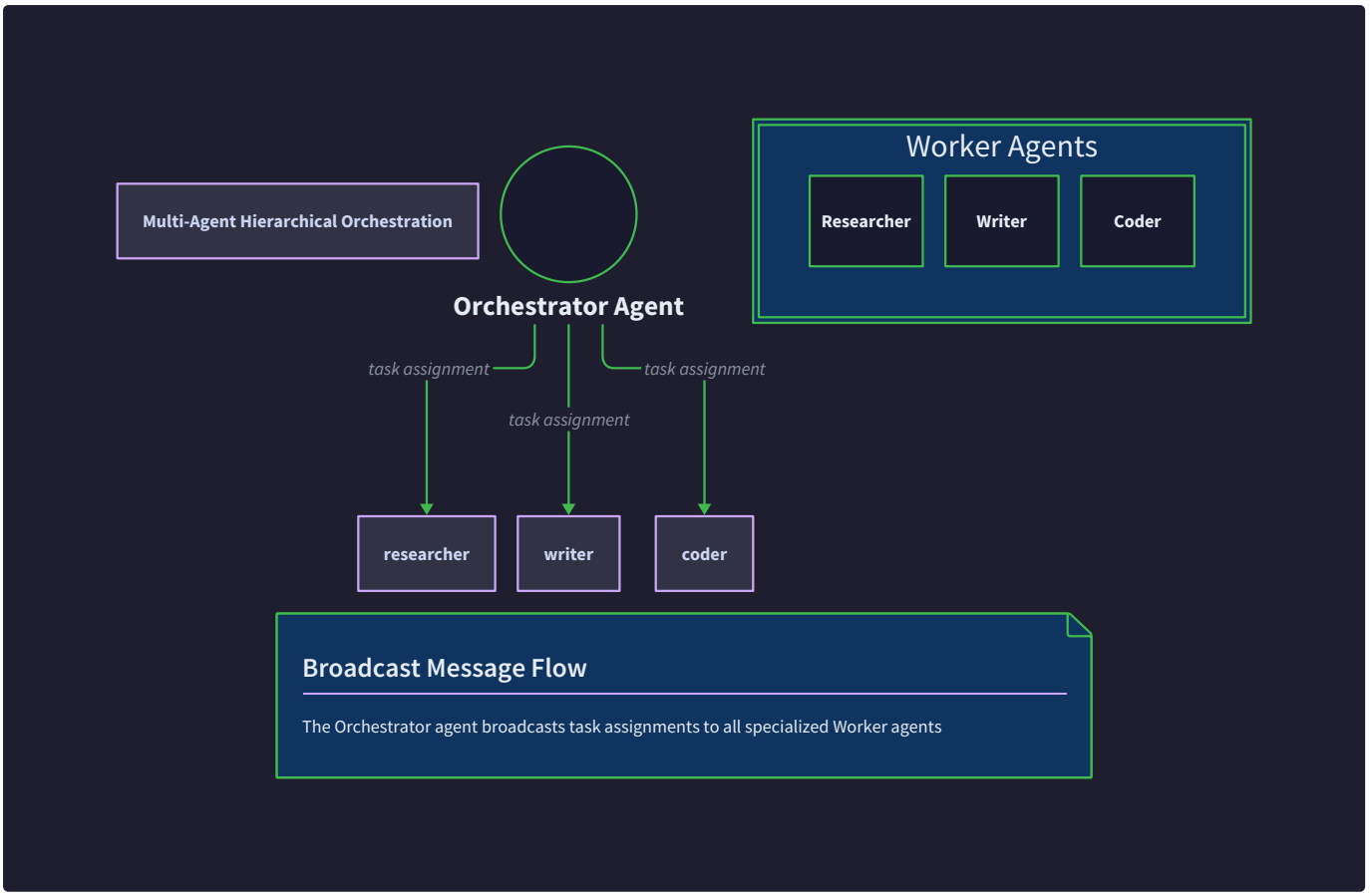
Useful for consensus-building or gathering diverse perspectives:

- **Structure**: No single orchestrator; all agents have equal standing
- **Communication Flow**: Any agent can broadcast messages to all others
- **Coordination**: Agents self-organize through voting or reputation systems
- **Best For**: Subjective tasks, creative brainstorming, error detection
- **Example**: Code review where multiple engineers independently review then discuss

3. Market-Based (Auction) Pattern

Efficient for resource-constrained environments or when agents have overlapping capabilities:

- **Structure**: Task issuer (could be orchestrator or any agent) acts as auctioneer
- **Communication Flow**: Task issuer broadcasts task requirements; agents bid with estimated cost/quality/timeline; issuer selects best bid
- **Coordination**: Market dynamics (supply/demand) determine task allocation
- **Best For**: Load balancing, cost optimization, real-time task scheduling
- **Example**: Processing a batch of heterogeneous data analysis tasks where agents have varying availability



ADR: Agent-to-Agent Communication Medium

Decision: Message Bus with Topic-Based Routing

- **Context**: Multiple agents need to communicate asynchronously in various patterns (one-to-one, one-to-many, many-to-many). The communication medium must support decoupled interaction, scalability, and flexible routing while maintaining message ordering guarantees for correlated conversations.
- **Options Considered**:
  1. **Direct Method Calls**: Each agent maintains references to other agents and calls their methods directly
  2. **Shared Blackboard**: Centralized key-value store where agents post and read messages
  3. **Message Bus with Topics**: Publish-subscribe system where agents send messages to topics and receive messages from subscribed topics
- **Decision**: Implement a lightweight in-memory message bus with topic-based routing, where each agent has an inbox topic and system-wide broadcast topics exist for announcements.
- **Rationale**:
  - **Decoupling**: Agents don't need direct references to each other, enabling dynamic agent addition/removal
  - **Flexibility**: Supports multiple communication patterns (broadcast via system topics, direct via agent-specific topics)
  - **Ordering Guarantees**: Messages to a single topic can be processed in order, maintaining conversation coherence
  - **Debugging**: Centralized message logging provides visibility into all inter-agent communication
  - **Future Extensibility**: Can be replaced with distributed message queue (Redis, RabbitMQ) for multi-process deployment
- **Consequences**:
  - Adds message routing overhead compared to direct calls
  - Requires careful topic naming conventions to avoid collisions
  - Need to handle message persistence for crash recovery if required

Option	Pros	Cons	Chosen?
Direct Method Calls	- Minimal latency - Simple implementation - Strong typing	- Tight coupling between agents - Difficult to add/remove agents dynamically - Doesn't support broadcast patterns - Deadlock risk if circular calls	❌
Shared Blackboard	- Centralized state easy to monitor - Simple persistence - Natural fit for shared context	- Becomes bottleneck with many agents - No built-in notification mechanism (polling required) - Race conditions on concurrent writes	❌
Message Bus with Topics	- Loose coupling between agents - Natural support for multiple patterns - Scalable with topic partitioning - Built-in async processing	- Additional abstraction layer - Message serialization overhead - Requires message format standardization	✅

Common Pitfalls and Mitigations

⚠️ Pitfall: Delegation Loops (A delegates to B delegates to A)

- **Description**: Agent A receives a task, decides it's outside its expertise, delegates to Agent B. Agent B similarly decides the task is outside its expertise and delegates back to Agent A, creating an infinite loop.
- **Why it's wrong**: The system consumes resources without making progress, eventually timing out or crashing.
- **Mitigation**:
  1. **Delegation Chain Tracking**: Include the delegation path in task requests; reject requests that would create a cycle
  2. **Fallback Responsibility**: Each agent must have a "last resort" handling for its domain rather than always delegating
  3. **Orchestrator Oversight**: The orchestrator monitors delegation patterns and intervenes when cycles are detected

⚠️ Pitfall: Orchestrator Bottleneck

- **Description**: All communication flows through the orchestrator, which becomes overwhelmed as the number of agents or message frequency increases.
- **Why it's wrong**: Scalability is limited, latency increases, and the orchestrator becomes a single point of failure.
- **Mitigation**:
  1. **Peer-to-Peer Communication**: Allow agents to communicate directly for coordination within their workgroup
  2. **Hierarchical Orchestration**: Create sub-orchestrators for domains, reducing the load on the root orchestrator
  3. **Async Non-Blocking Processing**: Use message queues so the orchestrator processes messages asynchronously without blocking

⚠️ Pitfall: Lack of Shared Context

- **Description**: Agents work in isolation without awareness of what others have discovered or decided, leading to contradictory outputs or redundant work.
- **Why it's wrong**: Final synthesis is difficult or impossible; agents waste effort rediscovering what others already know.
- **Mitigation**:
  1. **Mandatory Context Sharing**: Require agents to publish key findings to a shared context store
  2. **Context Injection**: Automatically inject relevant shared context into each agent's prompt
  3. **Consistency Checking**: Orchestrator validates that new results don't contradict established facts

⚠️ Pitfall: Unresponsive Agents

- **Description**: An agent crashes, hangs, or becomes unreachable, leaving its assigned tasks incomplete and blocking dependent tasks.
- **Why it's wrong**: The entire workflow stalls; timeout mechanisms may help but don't recover the partial work.
- **Mitigation**:
  1. **Heartbeat Monitoring**: Agents periodically send "alive" signals; orchestrator reassigns tasks from unresponsive agents
  2. **Checkpointing**: Agents periodically save intermediate state so work can be resumed by another agent
  3. **Task Redundancy**: Assign critical tasks to multiple agents and use the first completed result

⚠️ Pitfall: Conflicting Outputs Without Resolution

- **Description**: Different agents produce contradictory results (e.g., Researcher says "X is true," Validator says "X is false"), and no mechanism exists to resolve the conflict.
- **Why it's wrong**: The orchestrator cannot synthesize a coherent final answer; may arbitrarily pick one result.
- **Mitigation**:
  1. **Predefined Conflict Resolution Strategies**: Implement rules (majority vote, confidence scoring, source authority ranking)
  2. **Escalation to Human**: When automated resolution fails, pause and request human input
  3. **Verification Subtasks**: Generate additional validation tasks to gather more evidence

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Message Transport	In-memory Python queue with `asyncio.Queue`	Redis Pub/Sub or RabbitMQ for distributed agents
Role Management	Dictionary mapping agent IDs to role configurations	Database-backed role registry with permission levels
Shared Context	Python `Dict` in orchestrator memory	Redis or PostgreSQL with structured schema
Conflict Resolution	Simple majority vote or confidence-weighted averaging	Bayesian truth discovery algorithms
Monitoring	Structured logging to console/file	OpenTelemetry metrics + Grafana dashboard

B. Recommended File/Module Structure

```
project-archon/
├── agents/
│   ├── __init__.py
│   ├── base_agent.py             # Base Agent class
│   ├── single_agent.py           # Single-agent implementation
│   ├── multi_agent/
│   │   ├── __init__.py
│   │   ├── orchestrator.py       # OrchestratorAgent class
│   │   ├── worker_agent.py       # WorkerAgent base class
│   │   ├── specialized/
│   │   │   ├── researcher.py     # ResearcherAgent
│   │   │   ├── writer.py         # WriterAgent
│   │   │   └── coder.py          # CoderAgent
│   │   ├── message_bus.py        # MessageBus implementation
│   │   ├── roles.py              # Role definitions and registry
│   │   └── shared_context.py     # SharedContext manager
│   └── agent_factory.py          # Factory to create agents with roles
├── messaging/
│   ├── __init__.py
│   ├── message.py                # Message class definition
│   ├── protocols.py              # Message type definitions
│   └── serialization.py          # Message serialization utilities
└── orchestration/
    ├── __init__.py
    ├── patterns/
    │   ├── hierarchical.py       # Hierarchical orchestration
    │   ├── broadcast.py          # Broadcast pattern
    │   └── market_based.py       # Auction-based allocation
    ├── conflict_resolver.py      # Conflict resolution strategies
    └── task_allocator.py         # Match tasks to agent roles
```

C. Infrastructure Starter Code

Complete Message and Message Bus Implementation:

PYTHON

```python
# messaging/message.py

from dataclasses import dataclass, field

from datetime import datetime

from typing import Dict, Any, Optional

import uuid


@dataclass

class Message:

    """Structured message for inter-agent communication."""

    sender_id: str

    receiver_id: str  # "broadcast" for all agents

    message_type: str

    content: Dict[str, Any]

    timestamp: datetime = field(default_factory=datetime.now)

    correlation_id: Optional[str] = field(default_factory=lambda: str(uuid.uuid4()))



    def to_dict(self) -> Dict[str, Any]:

        """Convert message to dictionary for serialization."""

        return {

            "sender_id": self.sender_id,

            "receiver_id": self.receiver_id,

            "message_type": self.message_type,

            "content": self.content,

            "timestamp": self.timestamp.isoformat(),

            "correlation_id": self.correlation_id

        }



    @classmethod

    def from_dict(cls, data: Dict[str, Any]) -> "Message":

        """Create message from dictionary."""

        return cls(

            sender_id=data["sender_id"],

            receiver_id=data["receiver_id"],

            message_type=data["message_type"],

            content=data["content"],

            timestamp=datetime.fromisoformat(data["timestamp"]),

            correlation_id=data.get("correlation_id")

        )



# multi_agent/message_bus.py

import asyncio
```

```python
from typing import Dict, Set, Callable, Optional
from dataclasses import dataclass, field
from messaging.message import Message

@dataclass
class Subscription:
    """Subscription to a message topic."""
    callback: Callable[[Message], None]
    agent_id: str

class MessageBus:
    """In-memory message bus for agent communication."""

    def __init__(self):
        self._topics: Dict[str, asyncio.Queue] = {}
        self._subscriptions: Dict[str, Set[Subscription]] = {}
        self._lock = asyncio.Lock()

    async def publish(self, message: Message) -> None:
        """Publish a message to the appropriate topic(s)."""
        topics_to_publish = []

        # Always publish to receiver's specific topic
        if message.receiver_id != "broadcast":
            topics_to_publish.append(f"agent:{message.receiver_id}")

        # If broadcast, publish to all agents
        if message.receiver_id == "broadcast":
            async with self._lock:
                topics_to_publish.extend(self._subscriptions.keys())

        # Also publish to message type topic for pattern matching
        topics_to_publish.append(f"type:{message.message_type}")

        # Publish to each topic
        for topic in set(topics_to_publish):  # Deduplicate
            if topic in self._topics:
                await self._topics[topic].put(message)

    async def subscribe(self, agent_id: str, callback: Callable[[Message], None],
                        topics: Optional[list[str]] = None) -> None:
```

```
    """Subscribe an agent to message topics."""
    if topics is None:
        topics = [f"agent:{agent_id}", "type:broadcast"]

    subscription = Subscription(callback=callback, agent_id=agent_id)

    async with self._lock:
        for topic in topics:
            if topic not in self._subscriptions:
                self._subscriptions[topic] = set()
                self._topics[topic] = asyncio.Queue()
            self._subscriptions[topic].add(subscription)

async def unsubscribe(self, agent_id: str) -> None:
    """Remove all subscriptions for an agent."""
    async with self._lock:
        for topic in list(self._subscriptions.keys()):
            subscriptions = self._subscriptions[topic]
            to_remove = {sub for sub in subscriptions if sub.agent_id == agent_id}
            subscriptions.difference_update(to_remove)

            if not subscriptions:
                del self._subscriptions[topic]
                del self._topics[topic]

async def start_message_processor(self, agent_id: str) -> None:
    """Start processing messages for an agent (run in background task)."""
    topics = [f"agent:{agent_id}", "type:broadcast"]

    async def process_messages():
        for topic in topics:
            if topic in self._topics:
                asyncio.create_task(self._process_topic_messages(topic))

    asyncio.create_task(process_messages())

async def _process_topic_messages(self, topic: str) -> None:
    """Continuously process messages from a topic queue."""
    while True:
        try:
```

```
                message = await self._topics[topic].get()
                async with self._lock:
                    subscriptions = self._subscriptions.get(topic, set()).copy()

                # Deliver to all subscribers
                for subscription in subscriptions:
                    try:
                        subscription.callback(message)
                    except Exception as e:
                        print(f"Error in subscription callback: {e}")

                self._topics[topic].task_done()
            except asyncio.CancelledError:
                break
            except Exception as e:
                print(f"Error processing messages from topic {topic}: {e}")
```

D. Core Logic Skeleton Code

Orchestrator Agent Skeleton:

PYTHON

```python
# multi_agent/orchestrator.py

from typing import Dict, List, Optional, Set

from dataclasses import dataclass, field

import asyncio

from agents.base_agent import BaseAgent

from messaging.message import Message

from multi_agent.message_bus import MessageBus

from orchestration.patterns.hierarchical import HierarchicalOrchestrator


@dataclass

class DelegatedTask:

    """Track a task delegated to a worker agent."""

    task_id: str

    subtask_description: str

    assigned_agent: str

    status: str  # "pending", "executing", "completed", "failed"

    result: Optional[Dict] = None

    deadline: Optional[float] = None

    dependencies: List[str] = field(default_factory=list)


class OrchestratorAgent(BaseAgent):

    """Coordinator agent that manages multiple worker agents."""



    def __init__(self, agent_id: str, message_bus: MessageBus, available_roles: Dict):

        super().__init__(agent_id=agent_id)

        self.message_bus = message_bus

        self.available_roles = available_roles  # role -> [agent_ids]

        self.delegated_tasks: Dict[str, DelegatedTask] = {}

        self.shared_context: Dict[str, Any] = {}

        self.orchestration_pattern = HierarchicalOrchestrator()



        # Register message handlers

        self._register_message_handlers()



    async def run_task(self, user_query: str) -> str:

        """Primary public method to execute a user's task with multi-agent collaboration."""

        # TODO 1: Determine if task requires multi-agent (complexity threshold, multiple domains)

        # TODO 2: If single-agent sufficient, use parent class implementation

        # TODO 3: Otherwise, decompose task into subtasks using planner.create_plan()

        # TODO 4: Match subtasks to agent roles based on required capabilities

        # TODO 5: Delegate subtasks to appropriate worker agents via task_request messages
```

```
    # TODO 6: Monitor task completion with timeouts and error handling
    # TODO 7: Collect and validate results from all workers
    # TODO 8: Resolve conflicts if results contradict each other
    # TODO 9: Synthesize final answer from individual results
    # TODO 10: Return final answer to user
    pass

def _register_message_handlers(self) -> None:
    """Register handlers for different message types."""
    # TODO 1: Register handler for "task_response" messages to process worker results
    # TODO 2: Register handler for "status_update" messages to track progress
    # TODO 3: Register handler for "error" messages to handle worker failures
    # TODO 4: Register handler for "information_share" messages to update shared context
    pass

async def _handle_task_response(self, message: Message) -> None:
    """Process a task response from a worker agent."""
    # TODO 1: Extract task_id from message.content
    # TODO 2: Look up delegated task in self.delegated_tasks
    # TODO 3: Validate the result format and completeness
    # TODO 4: Update task status to "completed" or "failed"
    # TODO 5: Store result in delegated task record
    # TODO 6: Update shared context with relevant findings
    # TODO 7: Check if this unblocks dependent tasks
    # TODO 8: If all tasks complete, trigger final synthesis
    pass

async def _delegate_subtask(self, subtask_description: str, required_tools: List[str],
                            context: Dict) -> str:
    """Delegate a subtask to the most suitable agent."""
    # TODO 1: Find agents with roles that provide the required tools
    # TODO 2: Consider agent availability (not overloaded) and expertise
    # TODO 3: Select best agent using orchestration pattern (hierarchical, market, etc.)
    # TODO 4: Create unique task_id for tracking
    # TODO 5: Send task_request message to selected agent
    # TODO 6: Record delegation in self.delegated_tasks
    # TODO 7: Start timeout monitor for this task
    # TODO 8: Return task_id for tracking
    pass
```

```python
async def _resolve_conflicts(self, conflicting_results: List[Dict]) -> Dict:
    """Resolve contradictory results from multiple agents."""
    # TODO 1: Check if predefined resolution strategy exists for this task type
    # TODO 2: Apply strategy (majority vote, confidence scoring, source authority)
    # TODO 3: If automated resolution fails, escalate to human or generate verification task
    # TODO 4: Return resolved result
    pass
```

Worker Agent Base Class:

PYTHON

```python
# multi_agent/worker_agent.py

from agents.base_agent import BaseAgent

from messaging.message import Message

from multi_agent.message_bus import MessageBus


class WorkerAgent(BaseAgent):

    """Base class for specialized worker agents."""


    def __init__(self, agent_id: str, role: str, message_bus: MessageBus):

        super().__init__(agent_id=agent_id)

        self.role = role

        self.message_bus = message_bus

        self.current_tasks: Set[str] = set()


        # Subscribe to task requests for this agent

        asyncio.create_task(self._setup_message_subscriptions())


    async def _setup_message_subscriptions(self) -> None:

        """Subscribe to relevant message topics."""

        # TODO 1: Subscribe to agent-specific topic (f"agent:{self.agent_id}")

        # TODO 2: Subscribe to broadcast topics for coordination messages

        # TODO 3: Register message handler for task_request messages

        pass


    async def _handle_task_request(self, message: Message) -> None:

        """Process incoming task request from orchestrator."""

        # TODO 1: Extract task details from message.content

        # TODO 2: Check if agent has capacity (max concurrent tasks limit)

        # TODO 3: If capacity available, accept task and send acknowledgment

        # TODO 4: Execute task using agent's internal ReAct loop

        # TODO 5: Send periodic status_update messages during execution

        # TODO 6: On completion, send task_response message with result

        # TODO 7: On failure, send error message with details

        # TODO 8: Clean up task from current_tasks set

        pass


    async def request_help(self, task_id: str, issue: str, target_role: str) -> None:

        """Request assistance from another agent with specific expertise."""

        # TODO 1: Create help_request message

        # TODO 2: Send to broadcast or specific agent with target_role

        # TODO 3: Wait for response with timeout
```

```
        # TODO 4: Incorporate help into task execution
        pass
```

E. Python-Specific Hints

1. **Async/Await Patterns**: Use `asyncio.gather()` to wait for multiple agent responses concurrently, but be careful with error handling—wrap each in `try/except` to prevent one failure from crashing all.
2. **Message Serialization**: Use `json.dumps()` for message serialization, but be aware that datetime objects and custom types need special handling. Consider implementing custom JSONEncoder.
3. **Resource Management**: Use `asyncio.BoundedSemaphore` to limit concurrent tasks per agent, preventing overload.
4. **Timeout Handling**: Always use `asyncio.wait_for(task, timeout)` when waiting for agent responses, with a reasonable default timeout (e.g., 30-60 seconds for most tasks).
5. **Error Isolation**: Wrap each agent's execution in `try/except` to prevent a misbehaving agent from crashing the entire orchestrator.

F. Milestone Checkpoint

Verification Command:

```bash
python -m pytest tests/test_multi_agent.py -v -k "test_orchestration_delegation"
```

Expected Output:

```
test_orchestration_delegation PASSED
test_worker_accepts_task PASSED
test_conflict_resolution PASSED
test_message_bus_routing PASSED
```

Manual Verification Steps:

1. Start a test script that creates an orchestrator, researcher, and writer agent
2. Submit a task: "Research climate change impacts and write a 3-paragraph summary"
3. Observe in logs:
   - Orchestrator decomposes task into research and writing subtasks
   - Orchestrator sends task_request to researcher
   - Researcher performs web search, sends task_response with findings
   - Orchestrator sends task_request to writer with researcher's findings
   - Writer produces summary, sends task_response
   - Orchestrator returns final summary
4. Verify the final summary incorporates research findings

Signs of Problems:

- **Agents not receiving messages**: Check topic subscriptions and message bus initialization
- **Tasks stuck pending**: Check dependency resolution and agent capacity limits
- **Conflicting results not resolved**: Verify conflict resolution strategy is configured
- **High memory usage**: Ensure completed tasks are cleaned up from tracking dictionaries

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Delegation loop	Agents keep delegating same task back and forth	Check message logs for repeating `task_request` patterns between same agents	Implement delegation chain tracking; add fallback responsibility
Orchestrator unresponsive	Message queue overflow or deadlock	Check orchestrator's message queue size; look for blocking operations in message handlers	Use async non-blocking patterns; increase queue size; add deadlock detection
Worker agent ignores tasks	Incorrect topic subscription or message handler not registered	Verify agent is subscribed to correct topic; check if message handler is called	Ensure proper subscription in agent initialization; add message receipt logging
Shared context outdated	Agents not publishing findings or context not being distributed	Check if agents send `information_share` messages; verify context injection in prompts	Make context sharing mandatory; add version checks to context items
Conflict resolution fails	Resolution strategy not defined or contradictory results with equal confidence	Log conflict details and resolution attempt; check if strategy exists for this result type	Implement fallback strategy (human escalation); add confidence calibration
Tasks complete but final synthesis poor	Orchestrator not properly integrating results or losing context between steps	Check orchestrator's synthesis logic; verify all relevant results are passed to final step	Improve synthesis prompt; add validation step before final output

6. Interactions and Data Flow

> **Milestone(s):** 2, 3, 5

This section traces the end-to-end flow of data and control through the framework for two key scenarios: a single agent completing a complex task and multiple agents collaborating. Understanding these flows is critical for debugging, performance optimization, and extending the system—it shows how the architectural components we've designed actually work together in practice.

Single-Agent Task Execution Flow

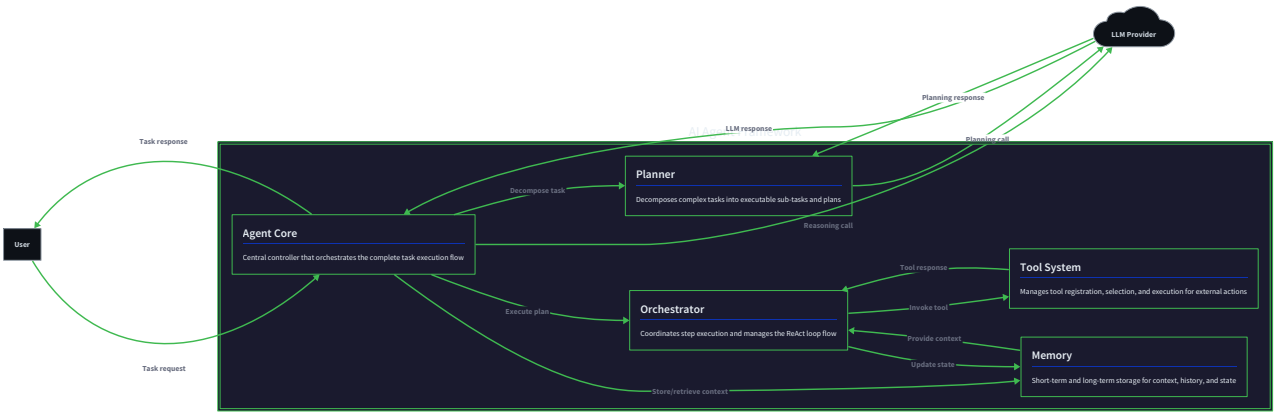
Mental Model: The Executive's Workday
Imagine our AI Executive arriving at work with a single complex assignment: "Research the latest developments in quantum computing and write a summary report." The Executive doesn't try to do everything at once. Instead, they:

1. **Plan their day** (Task Decomposition): Break the assignment into manageable steps: research → outline → draft → edit
2. **Work through each step** (ReAct Loop): For "research," they think about what to search for, use search tools, read results, then decide what's relevant
3. **Keep notes** (Memory): They jot down key findings, maintain current draft sections, and reference past similar reports
4. **Use specialized tools** (Tool System): They use different tools for different steps—web search, document editing, grammar checking

This disciplined workflow ensures methodical progress without getting overwhelmed by complexity.

High-Level Component Interaction

The following diagram illustrates how data flows through the system components when processing a single user query:



Detailed Step-by-Step Flow

Let's trace through a concrete example where a user submits the query: *"What were the main causes of the 2008 financial crisis? Summarize in bullet points and include at least three reputable sources."*

1. Initialization and Context Setup

Step	Component	Action	Data Produced
1	User Interface	Calls `Agent.run_task(user_query)` with the query string	User query passed to Agent Core
2	Agent Core	Creates initial `AgentState.PLANNING`, initializes empty `WorkingMemory` for this task	`AgentState` transitions from `IDLE` to `PLANNING`
3	Memory System	`MemoryRetriever.retrieve_context()` is called with the query	Context string containing relevant past conversations from `LongTermMemory` and recent messages from `EpisodicMemory`
4	Agent Core	Packages query + retrieved context into a `Task` object with only the `original_query` and `goal` fields populated	`Task(id="task_001", original_query=..., goal="...", subtasks=[], context={...})`

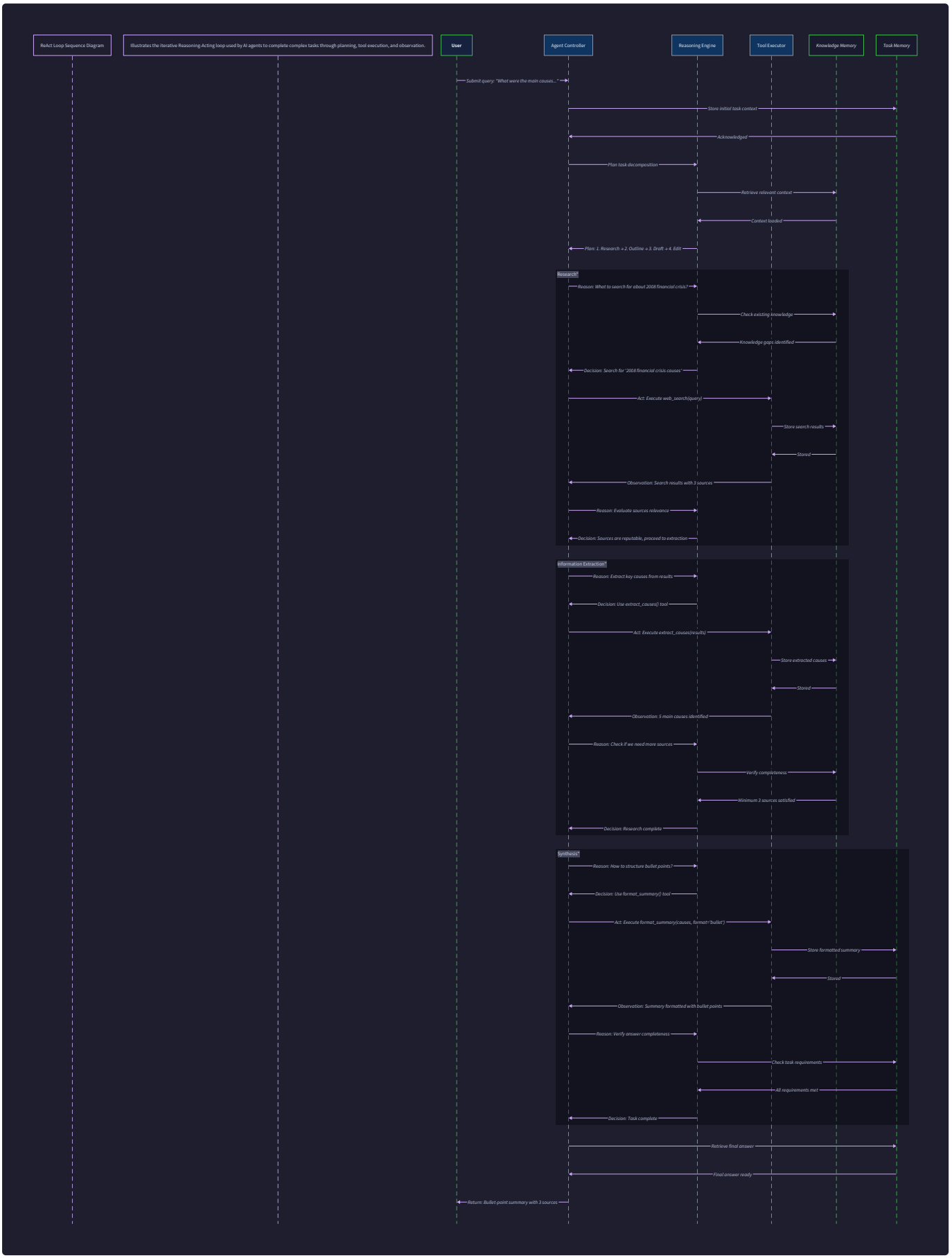
2. Planning Phase (Milestone 3)

Step	Component	Action	Data Produced
5	Planner	`Planner.create_plan(goal, available_tools, initial_context)` is invoked	LLM is prompted to decompose the goal into subtasks
6	LLM Provider	Returns structured decomposition: 1) Search for causes 2) Validate source credibility 3) Draft summary 4) Format bullets	JSON array of subtask descriptions with dependencies
7	Planner	Creates `SubTask` objects for each step, sets initial `SubTaskStatus.PENDING`, builds dependency graph	`Task` object now has populated `subtasks` list with 4 `SubTask` instances
8	Planner	Performs topological sort on dependency graph to determine `execution_order`	`Plan` object with `subtask_dag` and `execution_order`
9	Agent Core	Updates `AgentState` to `EXECUTING` and selects first ready subtask	`AgentState.EXECUTING`, first subtask ID selected

3. ReAct Loop Execution for a Subtask (Milestone 2)

Now the agent executes the first subtask: *"Search the web for main causes of 2008 financial crisis, focusing on authoritative sources"*

The following sequence diagram illustrates the Think-Act-Observe cycle:



Here's the detailed numbered algorithm for executing a single subtask via `Agent._execute_subtask(subtask)`:

1. **Initialize ReAct Context**: Create a `ReActLoopContext` with:
    - `subtask_description` from the subtask

- `expected_output` from the subtask (if specified)
- `conversation_history` from `EpisodicMemory.get_recent(max_tokens)`
- `iteration = 0`, `max_iterations = 10` (configurable)
- `state = AgentState.THINKING`
- `previous_actions = []` (empty list)

2. **Retrieve Available Tools**: Call `ToolRegistry.list_tools()` to get all registered tools for this agent.
3. **Enter ReAct Loop**: While `context.iteration < context.max_iterations` and `context.state` not in `{FINISHED, ERROR}`:

   a. **Think (Reasoning Step)**:

   1. Format prompt using `ReActAgent._format_react_prompt(context, available_tools, external_context)`
   2. External context includes: working memory state + long-term memories relevant to current subtask
   3. Send prompt to LLM provider
   4. Receive LLM response text

   b. **Parse Action**: Call `parse_llm_action_response(response_text)`:

   1. Check for "Final Answer:" marker indicating completion
   2. If found: Extract final answer text, update `context.state = AgentState.FINISHED`, break loop
   3. Else: Extract JSON action using `extract_json_from_text(response_text)`
   4. Validate JSON has `tool_name` and `arguments` fields
   5. If parsing fails: Create error observation, increment iteration, continue

   c. **Act (Tool Execution)**:

   1. Retrieve tool: `ToolRegistry.get_tool(tool_name)`
   2. If tool not found: Create error observation "Tool X not available", continue
   3. Validate arguments against tool's `parameters_schema` using JSON Schema validation
   4. Execute: `ToolExecutionEngine.execute_tool_call(tool_name, arguments, timeout_seconds=30)`
   5. Wait for `ToolResult` with status `SUCCESS`, `ERROR`, or `TIMEOUT`

   d. **Observe (Result Processing)**:

   1. Format observation based on `ToolResult.status`:
      - `SUCCESS`: "Tool [name] succeeded: [content]"
      - `ERROR`: "Tool [name] failed: [error_message]"
      - `TIMEOUT`: "Tool [name] timed out after X seconds"
   2. Append observation to `context.conversation_history`
   3. Update `context.previous_actions` with the executed action
   4. Store useful results in `WorkingMemory` under appropriate keys

   e. **State Update**:

   1. Increment `context.iteration`
   2. If `ToolResult.status == SUCCESS` and subtask appears complete: `context.state = AgentState.FINISHED`
   3. If maximum retries on same tool exceeded: `context.state = AgentState.ERROR`

4. **Loop Termination**:
   - If loop ended with `context.state == AgentState.FINISHED`: Return `ToolResult.success(final_answer)`
   - If loop ended with `context.state == AgentState.ERROR`: Return `ToolResult.error("Failed after X iterations")`
   - If `context.iteration >= context.max_iterations`: Return `ToolResult.error("Max iterations reached")`
5. **Memory Persistence**:
   - Append entire ReAct conversation to `EpisodicMemory.append()` for future reference
   - Store key findings in `LongTermMemory.store()` with appropriate embeddings for semantic retrieval
   - Update `WorkingMemory` with intermediate results for use in subsequent subtasks

4. Plan Progression and Completion

Step	Component	Action	Data Produced
10	Agent Core	After first subtask completes, updates its `SubTaskStatus` to `COMPLETED` and stores `ToolResult`	`SubTask.with_status(COMPLETED).with_result(tool_result)`
11	Agent Core	Checks which subtasks become `is_ready()` after dependency satisfaction	List of ready subtask IDs
12	Agent Core	Selects next ready subtask (may execute independent subtasks in parallel if supported)	Next `SubTask` to execute
13	Loop	Repeats ReAct execution for each subtask in dependency order	Progressively completed subtasks
14	Agent Core	When all subtasks `status == COMPLETED`: Aggregates results from each subtask	Combined final answer
15	Agent Core	Updates `AgentState` to `FINISHED`, returns final answer to user	`AgentState.FINISHED`, final answer string
16	Memory System	Finalizes memory storage: summarizes conversation, indexes key learnings	Compacted `EpisodicMemory`, new `LongTermMemory` entries

Concrete Walk-Through Example

Let's examine a specific iteration within the ReAct loop for the "search" subtask:

Initial Context: `ReActLoopContext(subtask_description="Search for main causes...", iteration=0, previous_actions=[])`

1. **Think**: The LLM receives a prompt containing: "You are an AI assistant. Current task: Search for main causes... Available tools: web_search(query), calculator(expression). Previous actions: none. Think step by step."
2. **LLM Response**:

```
   I need to search for authoritative information about the 2008 financial crisis causes. I should use specific search terms to get
relevant results.

   Action: {
     "tool_name": "web_search",
     "arguments": {
       "query": "2008 financial crisis main causes subprime mortgages Lehman Brothers"
     }
   }
```

3. **Parse**: `parse_llm_action_response()` extracts JSON, validates `web_search` exists, validates arguments match schema.
4. **Act**: `ToolExecutionEngine.execute_tool_call("web_search", arguments, timeout=30)` calls the actual web search API, returns:

```python
   ToolResult.success(
     content="Found 3 relevant sources: 1) Federal Reserve report on... 2) Wikipedia article lists... 3) NY Times analysis...",
     metadata={"sources": ["fed.gov/...", "wikipedia.org/...", "nytimes.com/..."]}
   )
```

5. **Observe**: Observation formatted as: "Tool web_search succeeded: Found 3 relevant sources..."
6. **Next Cycle**: LLM now sees this observation and might decide to: a) Use another tool to fetch specific articles, b) Extract key points, c) Move to final answer if satisfied.

This back-and-forth continues until the agent either produces a "Final Answer:" or reaches the iteration limit.

Data Transformation Through the Pipeline

Stage	Input	Processing	Output
Raw Query	User natural language string	Context retrieval, initial parsing	`Task` with `goal` field
Planning	`Task.goal` string	LLM decomposition, dependency analysis	`Task` with `subtasks` list and `Plan` DAG
Subtask Execution	`SubTask` object	ReAct loop with tool calls	`ToolResult` with content
Memory Integration	`ToolResult.content`	Embedding generation, relevance scoring	`MemoryEntry` in appropriate memory layer
Final Aggregation	All `SubTask.results`	Concatenation, formatting, synthesis	Final answer string

Critical Control Points

> **Design Insight**: The flow maintains **immutable state updates** throughout. Each component returns new instances rather than mutating shared state. This makes the system easier to debug, test, and reason about, as you can trace exactly how state evolves through each transformation.

Multi-Agent Delegation Flow

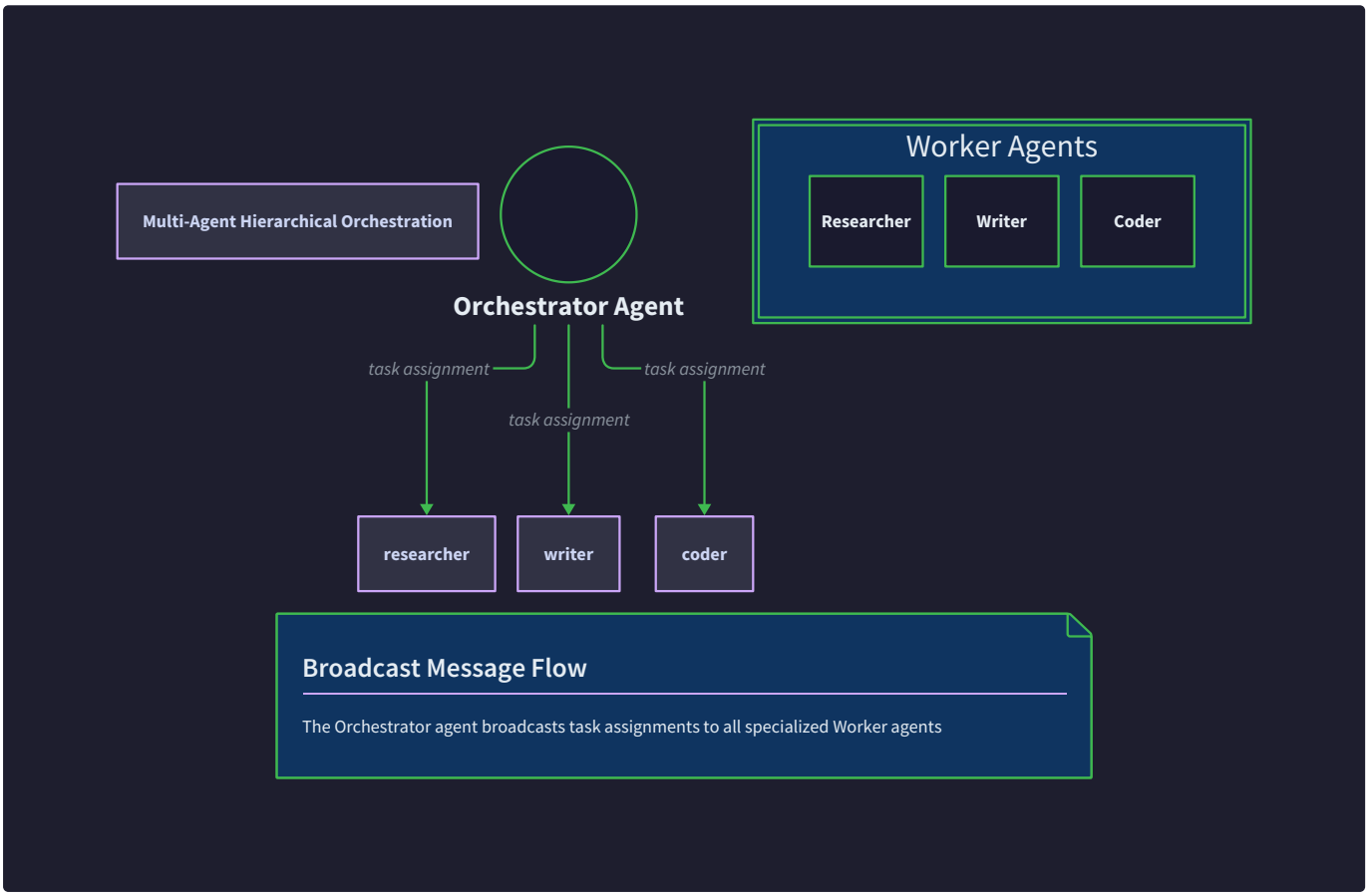
Mental Model: Department Meeting

When a task is too complex for one executive, they call a department meeting. The **Orchestrator** (department head) breaks the problem down and assigns pieces to specialists: the **Researcher** (web search expert), **Writer** (document specialist), and **Fact-Checker** (accuracy verifier). They pass memos (`Message` objects) back and forth, share findings on a common whiteboard (`Shared Context`), and the Orchestrator compiles everything into a final report.

This collaborative approach leverages specialization while maintaining coordination.

Orchestration Architecture

The following diagram shows the hierarchical structure of a multi-agent system:



Step-by-Step Collaborative Execution

Let's trace through a more complex query that triggers multi-agent collaboration: *"Create a Python script that fetches today's weather from an API, analyzes historical trends, and generates a visualization with matplotlib. Include error handling and documentation."*

1. Task Reception and Initial Assessment

Step	Component	Action	Data Produced
1	User Interface	Calls `OrchestratorAgent.run_task(user_query)`	Query passed to orchestrator
2	OrchestratorAgent	Assesses complexity, determines need for multi-agent approach	Decision to delegate to specialists
3	OrchestratorAgent	Calls `Planner.create_plan()` with multi-agent awareness	Plan with subtasks tagged for different agent roles
4	OrchestratorAgent	Initializes `shared_context` dictionary for this task	`shared_context = {"task_id": "task_002", "original_query": ...}`

2. Role-Based Task Delegation

The orchestrator identifies three required specialties:

1. **API Specialist**: Handle weather API calls and error handling
2. **Data Analyst**: Process historical trends and calculations
3. **Visualization Expert**: Create matplotlib visualizations

Step	Component	Action	Data Produced
5	OrchestratorAgent	For each subtask, calls `_delegate_subtask(subtask_description, required_tools, context)`	`DelegatedTask` objects created for each assignment
6	OrchestratorAgent	Matches subtasks to agents based on `role` and advertised capabilities	Assignment mapping: API→AgentA, Analysis→AgentB, Viz→AgentC
7	Message Bus	`MessageBus.publish()` sends `Message` objects to each agent's subscription topic	Messages queued for delivery
8	Worker Agents	Each agent's background processor calls `_handle_task_request(message)`	Task acceptance/acknowledgment

3. Parallel Execution with Coordination

Now all three agents work in parallel on their respective subtasks:

Agent A (API Specialist) Workflow:

1. Receives `Message` with task: "Fetch today's weather from API with error handling"
2. Executes its own ReAct loop with tools: `http_get`, `json_parse`, `error_handler`
3. Stores results in `shared_context["weather_data"]` and `shared_context["api_errors"]`
4. Sends completion `Message` back to orchestrator

Agent B (Data Analyst) Workflow:

1. Waits for dependency: `shared_context["weather_data"]` to be populated
2. Periodically checks shared context (or receives notification via Message Bus)
3. Once data available, processes historical trends using `calculate_statistics`, `detect_trends` tools
4. Stores results in `shared_context["analysis_results"]`
5. Sends completion `Message`

Agent C (Visualization Expert) Workflow:

1. Waits for dependencies: both weather data and analysis results
2. Creates visualization using `matplotlib_generate`, `format_chart` tools
3. Stores plot image in `shared_context["visualization"]`
4. Sends completion `Message`

4. Result Aggregation and Conflict Resolution

Step	Component	Action	Data Produced
9	OrchestratorAgent	`_handle_task_response()` processes each completion message	Updates `DelegatedTask.status` for each subtask
10	OrchestratorAgent	Monitors completion of all delegated tasks	Waits for all `DelegatedTask.status == "COMPLETED"`
11	OrchestratorAgent	Detects potential conflicts (e.g., Agent A and Agent B disagree on data interpretation)	Flag for conflict resolution
12	OrchestratorAgent	If conflicts exist: calls `_resolve_conflicts(conflicting_results)`	Unified result through voting or arbitration
13	OrchestratorAgent	Compiles final output from `shared_context` and individual agent results	Draft final answer
14	OrchestratorAgent	May send draft to a "Reviewer" agent for quality check	Additional refinement loop
15	OrchestratorAgent	Returns final compiled answer to user	Complete Python script with all requested features

Message Passing Protocol in Detail

The communication between agents follows a structured protocol:

Message Structure for Task Delegation:

```python
Message(
    sender_id="orchestrator_001",
    receiver_id="api_specialist_002",
    message_type="TASK_REQUEST",
    content={
        "task_id": "subtask_003",
        "description": "Fetch today's weather from OpenWeatherMap API",
        "expected_output": "JSON with current temp, humidity, and timestamp",
        "deadline": 1698765300.0,
        "dependencies": [],
        "shared_context_key": "weather_data"
    },
    timestamp=datetime.now(),
    correlation_id="task_002_sub_003"
)
```

Message Flow Patterns:

1. **Request-Response**: Orchestrator → Worker (TASK_REQUEST), Worker → Orchestrator (TASK_RESPONSE)
2. **Broadcast**: Orchestrator → All Workers (CONTEXT_UPDATE when shared state changes)
3. **Peer-to-Peer**: Worker A → Worker B (HELP_REQUEST when specialist needs consultation)
4. **Status Updates**: Periodic heartbeat messages to detect unresponsive agents

Concrete Multi-Agent Scenario Walk-Through

Let's examine a specific interaction when Agent B (Data Analyst) encounters a problem:

Scenario: Agent B calculates a 30% temperature increase trend, but Agent C's visualization shows only 15%.

1. **Conflict Detection**: Orchestrator's `_handle_task_response()` compares `shared_context["analysis_results"]["trend"]` vs `shared_context["visualization_metadata"]["trend_value"]`
2. **Resolution Trigger**: Orchestrator detects mismatch beyond tolerance threshold (e.g., >5% difference)
3. **Arbitration Process**:
   - Orchestrator sends `Message` to both agents requesting explanation of methodology
   - Agents respond with `Message` containing their calculation steps and assumptions
   - Orchestrator (or a dedicated "Arbitrator" agent) evaluates methodologies
   - Decision: Agent C had incorrect date range filter
   - Orchestrator sends `Message` to Agent C to regenerate visualization with corrected data
4. **Updated Flow**: Agent C fetches corrected data from `shared_context`, regenerates visualization, updates shared context, notifies completion.

Data Flow Through Shared Context

The `shared_context` dictionary evolves throughout execution:

Phase	Key	Value Type	Set By	Used By
Initial	`original_query`	String	Orchestrator	All agents
API Phase	`weather_data`	Dict	Agent A	Agent B, Agent C
Analysis Phase	`analysis_results`	Dict	Agent B	Agent C, Orchestrator
Visualization Phase	`visualization`	Bytes/Path	Agent C	Orchestrator (final compile)
Conflict	`discrepancy_log`	List	Orchestrator	Arbitrator agent
Final	`complete_script`	String	Orchestrator	User output

Failure Recovery in Multi-Agent Flow

When an agent fails or times out:

1. **Detection**: Orchestrator monitors `DelegatedTask.deadline` and heartbeat messages
2. **Reassignment**: If agent unresponsive, orchestrator calls `_delegate_subtask()` to another agent with same role
3. **State Recovery**: New agent reads `shared_context` to understand what was already accomplished
4. **Compensation**: If partial work exists but is inconsistent, may need to rollback dependent tasks

> **Design Insight**: The **shared context** acts as a **blackboard architecture**—a common workspace where agents can read and write partial results. This decouples agents from each other while still allowing coordination. The orchestrator acts as the blackboard controller, managing access and consistency.

Performance Considerations

Parallel vs Sequential Execution:

- Independent subtasks execute in parallel (API fetch + historical analysis can happen concurrently)
- Dependent subtasks wait for prerequisites (visualization waits for both data sources)
- The orchestrator uses the `Plan.execution_order` (topological sort of DAG) to determine maximum parallelism

Message Bus Overhead:

- Each message adds latency but enables loose coupling
- Batching non-urgent messages (e.g., multiple context updates) can reduce overhead
- Direct method calls used for time-critical communication within same process

Shared Context Consistency:

- Simple dictionary works for single-process multi-agent
- For distributed agents, need distributed cache (Redis) or versioned storage
- Conflict detection happens at read-time with version checking

---

Implementation Guidance

A. Technology Recommendations for Data Flow

Component	Simple Option	Advanced Option
State Management	Immutable dataclasses with `copy()`	Functional lenses (e.g., `pymonad`)
Message Passing	In-memory `asyncio.Queue` per agent	Redis Pub/Sub or RabbitMQ
Shared Context	Thread-safe `dict` with locks	Versioned key-value store (etcd, Zookeeper)
Flow Visualization	Manual logging with step markers	OpenTelemetry tracing + Jaeger

B. Recommended File Structure for Flow Management

```
project-archon/
├── agents/
│   ├── base_agent.py          # BaseAgent with run_task() method
│   ├── single_agent.py        # SingleAgent (combines all components)
│   ├── orchestrator.py        # OrchestratorAgent
│   └── worker.py             # WorkerAgent
├── core/
│   ├── flow_controller.py     # Main coordinator of component interactions
│   ├── state_manager.py       # Manages AgentState transitions
│   └── context_broker.py      # Broker for shared context access
├── messaging/
│   ├── message_bus.py         # MessageBus implementation
│   ├── protocols.py           # Message type definitions
│   └── serialization.py       # Message (de)serialization
└── examples/
    └── flow_examples.py       # Example end-to-end flows for testing
```

C. Flow Controller Starter Code

PYTHON

```python
# core/flow_controller.py

import asyncio

from typing import Dict, Any, Optional

from dataclasses import dataclass, field

from enum import Enum


from .state_manager import AgentState

from agents.base_agent import BaseAgent

from planning.planner import Planner

from memory.memory_system import MemorySystem

from tools.registry import ToolRegistry


@dataclass

class ExecutionFlow:

    """Tracks the progress of a task through the system."""

    task_id: str

    current_state: AgentState = AgentState.IDLE

    active_subtask_id: Optional[str] = None

    completed_subtasks: set = field(default_factory=set)

    shared_context: Dict[str, Any] = field(default_factory=dict)

    flow_start_time: float = field(default_factory=lambda: time.time())



    def mark_subtask_complete(self, subtask_id: str) -> None:

        """Update flow state when a subtask completes."""

        self.completed_subtasks.add(subtask_id)

        self.active_subtask_id = None



    def get_next_ready_subtask(self, task) -> Optional[str]:

        """Find the next subtask that has all dependencies satisfied."""

        # TODO 1: Get list of all subtasks from task.subtasks

        # TODO 2: Filter to subtasks with status PENDING

        # TODO 3: For each candidate, check if all dependencies are in self.completed_subtasks

        # TODO 4: Return the first ready subtask ID (or None if none ready)

        pass


class FlowController:

    """Orchestrates the end-to-end flow of task execution."""



    def __init__(self, agent: BaseAgent, planner: Planner,

                 memory: MemorySystem, tool_registry: ToolRegistry):

        self.agent = agent

        self.planner = planner
```

```
        self.memory = memory
        self.tool_registry = tool_registry
        self.active_flows: Dict[str, ExecutionFlow] = {}

    async def execute_task(self, user_query: str) -> str:
        """Main entry point for single-agent task execution."""
        # TODO 1: Generate unique task_id (e.g., UUID or timestamp-based)
        # TODO 2: Create ExecutionFlow instance, set state to PLANNING
        # TODO 3: Retrieve context from memory: memory.retriever.retrieve_context(user_query)
        # TODO 4: Call planner.create_plan() with goal=user_query and retrieved context
        # TODO 5: Update flow state to EXECUTING
        # TODO 6: While there are pending subtasks in the plan:
        #   a. Get next ready subtask using flow.get_next_ready_subtask()
        #   b. If none ready but pending remain, log deadlock warning
        #   c. Execute subtask: await agent._execute_subtask(subtask, flow.shared_context)
        #   d. Update subtask status with result
        #   e. Update flow.mark_subtask_complete()
        #   f. Store relevant results in flow.shared_context for dependent tasks
        # TODO 7: When all subtasks complete, compile final answer from results
        # TODO 8: Update flow state to FINISHED, clean up flow tracking
        # TODO 9: Return final answer to user
        pass

    async def execute_multiagent_task(self, user_query: str,
                                      worker_agents: List[BaseAgent]) -> str:
        """Entry point for multi-agent collaborative execution."""
        # TODO 1: Create ExecutionFlow instance
        # TODO 2: Initialize message bus if not already running
        # TODO 3: Decompose task using planner (with role-aware decomposition)
        # TODO 4: For each subtask, determine required role/tools
        # TODO 5: Match subtasks to worker agents based on capabilities
        # TODO 6: Delegate subtasks via message bus: message_bus.publish(task_message)
        # TODO 7: Set up response handlers for each worker
        # TODO 8: Monitor completion via flow.completed_subtasks
        # TODO 9: Handle conflicts if multiple agents produce contradictory results
        # TODO 10: Aggregate final results from shared_context
        # TODO 11: Return compiled answer
        pass
```

D. Message Bus Implementation (Complete)

PYTHON

```python
# messaging/message_bus.py

import asyncio

from typing import Dict, Set, Callable, Optional, List

from dataclasses import dataclass, field

import logging


from .protocols import Message


@dataclass

class Subscription:

    """Represents an agent's subscription to message topics."""

    callback: Callable[[Message], None]

    agent_id: str


class MessageBus:

    """Simple in-memory publish-subscribe message bus for agent communication."""



    def __init__(self):

        self._topics: Dict[str, asyncio.Queue] = {}

        self._subscriptions: Dict[str, Set[Subscription]] = {}

        self._lock = asyncio.Lock()

        self._logger = logging.getLogger(__name__)



    async def publish(self, message: Message) -> None:

        """Publish a message to the appropriate topic(s)."""

        # TODO 1: Determine topic from message.receiver_id or message.message_type

        # TODO 2: Get or create queue for the topic

        # TODO 3: Put message in the queue

        # TODO 4: Notify all subscribers to this topic

        pass



    async def subscribe(self, agent_id: str,

                        callback: Callable[[Message], None],

                        topics: Optional[List[str]] = None) -> None:

        """Subscribe an agent to message topics."""

        # TODO 1: If topics is None, subscribe to agent-specific topic (f"agent_{agent_id}")

        # TODO 2: For each topic, add Subscription to self._subscriptions[topic]

        # TODO 3: Start message processor for this agent if not already running

        pass



    async def unsubscribe(self, agent_id: str) -> None:

        """Remove all subscriptions for an agent."""
```

```
        # TODO 1: Iterate through all topics in self._subscriptions
        # TODO 2: Remove any Subscription with matching agent_id
        # TODO 3: Clean up empty topic sets
        pass

    async def start_message_processor(self, agent_id: str) -> None:
        """Start processing messages for an agent (run in background task)."""
        # TODO 1: Create async task that continuously processes messages
        # TODO 2: Get messages from agent's dedicated queue
        # TODO 3: Call registered callback for each message
        # TODO 4: Handle errors gracefully (log, don't crash processor)
        pass
```

E. Milestone Checkpoint: Verifying Data Flow

After implementing the flow controller, verify with these tests:

```bash
# Test single-agent flow with a simple task
python -m pytest tests/test_flow_controller.py::TestSingleAgentFlow -v

# Expected output should show:
# - Task decomposition occurs (plan created with subtasks)
# - ReAct loop executes for each subtask
# - Final answer returned within timeout
# - Memory updated appropriately

# Test multi-agent flow (requires at least 2 worker agents)
python -m pytest tests/test_flow_controller.py::TestMultiAgentFlow -v

# Expected output:
# - Orchestrator decomposes task
# - Messages exchanged between agents
# - Shared context populated by multiple agents
# - Final answer combines contributions from all agents
```

Manual Verification Checklist:

1. **Single-Agent Happy Path**:
   - Start agent with web_search and calculator tools
   - Query: "What is 15 * 24? Also search for interesting facts about the number 360."
   - Verify: Calculator tool called first, then web_search, final answer includes both.
2. **Multi-Agent Coordination**:
   - Start orchestrator + 2 worker agents (one with calculation tools, one with search tools)
   - Same query as above
   - Verify: Orchestrator delegates calculation to Agent1, search to Agent2
   - Check shared context contains both results
   - Final answer synthesized by orchestrator

F. Debugging Tips for Flow Issues

Symptom	Likely Cause	How to Diagnose	Fix
Agent gets stuck at PLANNING	LLM not responding or plan parsing fails	Check LLM API connection, examine planner prompt	Add timeout to LLM call, improve error handling in parser
Subtasks execute in wrong order	Dependencies not correctly calculated	Print `Plan.execution_order`, verify topological sort	Check dependency parsing in planner, ensure DAG has no cycles
Shared context missing data	Race condition in multi-agent	Log timestamp of writes vs reads	Add synchronization (locks) or use versioned writes
Messages lost between agents	Queue overflow or processor crash	Check message bus queue sizes, monitor processor tasks	Increase queue size, add dead-letter queue for undelivered messages
Final answer missing subtask results	Aggregation logic error	Log each subtask result before aggregation	Verify aggregation collects all `SubTask.result` fields
Infinite loop in ReAct	No termination condition met	Log iteration count and state each cycle	Add better termination detection, lower `max_iterations`

G. Language-Specific Hints for Python Implementation

1. **Async/Await Pattern**: Use `asyncio.create_task()` for parallel subtask execution, but be mindful of resource limits.
2. **Immutable Updates**: Use `dataclasses.replace()` or dictionary unpacking for functional updates:

```python
# Instead of mutating:
# subtask.status = COMPLETED  # BAD

# Do:
new_subtask = dataclasses.replace(subtask, status=SubTaskStatus.COMPLETED)
# Or:
new_subtask = SubTask(**{**subtask.__dict__, 'status': SubTaskStatus.COMPLETED})
```

3. **Context Management**: Use `contextvars` for request-scoped context that follows async execution:

```python
import contextvars
current_flow = contextvars.ContextVar('current_flow')

async def execute_subtask(subtask):
    flow = current_flow.get()
    # flow is available throughout async call chain
```

4. **Message Serialization**: Use `pydantic` for Message validation or `dataclasses_json` for simple serialization:

```python
from dataclasses import dataclass
from dataclasses_json import dataclass_json


@dataclass_json
@dataclass
class Message:
    sender_id: str
    # ... other fields


# Serialize/deserialize easily
json_str = message.to_json()
new_message = Message.from_json(json_str)
```

5. **Flow Visualization**: Add structured logging with step markers:

```python
structured_logger.info({
    "event": "subtask_start",
    "task_id": flow.task_id,
    "subtask_id": subtask.id,
    "timestamp": time.time()
})
```

This creates logs that can be parsed to visualize execution flow.

7. Error Handling and Edge Cases

> **Milestone(s):** 1, 2, 3, 4, 5

Autonomous AI agents operate in a fundamentally unpredictable environment—they interface with unreliable LLMs, external APIs with variable latency, user inputs of varying quality, and complex state machines that can enter unexpected configurations. A robust error handling strategy is not merely a defensive programming practice; it is the **safety net that prevents the AI Executive from falling into infinite loops, corrupting data, or delivering nonsensical results**. This section catalogs the expected failure modes across all framework components and defines systematic recovery strategies that transform the agent from a brittle script into a resilient, self-correcting system.

Think of error handling in an AI agent framework as the **emergency response protocols in a nuclear power plant**. You cannot prevent all possible malfunctions, but you can design layers of defense: automated sensors (error detection), redundant systems (retry mechanisms), containment vessels (sandboxed execution), and emergency shutdown procedures (graceful fallbacks). Each component has its own failure signatures, and the system must coordinate responses across components to maintain overall stability.

7.1 Failure Mode Taxonomy

Failures in an AI agent system follow a predictable pattern aligned with the five architectural pillars. The taxonomy below categorizes errors by their origin, symptom, and potential impact on the agent's mission.

7.1.1 LLM-Related Failures

The Large Language Model is the agent's "brain," but it's an unreliable reasoning engine prone to specific failure modes.

Failure Mode	Primary Symptom	Detection Method	Impact Severity
Hallucination	LLM invents non-existent tools, parameters, or facts	Tool registry lookup fails; parameter validation rejects invalid schema; fact-checking tools contradict statements	Medium-High: Can lead to invalid actions, corrupted state
Refusal/Non-Compliance	LLM outputs "I cannot do that" or refuses to follow the ReAct format	Pattern matching in `parse_llm_action_response` detects refusal keywords; no action/final answer extracted	Medium: Task stalls; requires re-prompting or strategy change
Format Violation	LLM outputs freeform text instead of structured JSON or action format	`extract_json_from_text` returns `None`; regex patterns fail to match	Low-Medium: Parsing fails, but recoverable with guidance
Context Window Overflow	LLM truncates important history or ignores earlier instructions	Token counting exceeds model limit; LLM returns incoherent or incomplete responses	High: Critical context lost, leading to repetitive or off-track behavior
Reasoning Degradation	LLM gets stuck in circular logic or fails to progress	Monitoring `ReActLoopContext.iteration` against `max_iterations`; detecting repeated tool calls with same arguments	Medium: Wastes iterations without task advancement

7.1.2 Tool Execution Failures

Tools are the agent's "hands"—they interface with the external world and are subject to network issues, timeouts, and logical errors.

Failure Mode	Primary Symptom	Detection Method	Impact Severity
Timeout	Tool execution exceeds allowed duration	`asyncio.wait_for` raises `TimeoutError`; `ToolResult.status` becomes `TIMEOUT`	Medium: Task delayed but retry may succeed
Crash/Exception	Tool raises unhandled exception	`try/except` block in `ToolExecutionEngine.execute_tool_call` catches exception	Medium-High: May indicate buggy tool or invalid state
Invalid Parameters	Tool receives parameters that fail validation	JSON Schema validation in execution engine rejects input before execution	Low: Preventable with better validation
Network Unavailability	External API calls fail (HTTP errors, connection refused)	HTTP client raises exceptions; `ToolResult.status` becomes `ERROR` with network error message	Medium: Dependent on external service availability
Resource Exhaustion	Tool consumes excessive memory, CPU, or disk	Monitoring system metrics; `ResourceError` exceptions	High: Can crash entire agent process
Permission Denied	Tool lacks required credentials or access rights	Authentication/authorization errors from external services	High: Task cannot proceed without elevation

7.1.3 Planning & Task Decomposition Failures

The planner translates high-level goals into executable steps, but this decomposition can be flawed or become invalid during execution.

Failure Mode	Primary Symptom	Detection Method	Impact Severity
Circular Dependency	Subtask A depends on B, which depends on A	Cycle detection during DAG topological sort in plan execution	High: Deadlock—no subtask can execute
Over-Decomposition	Plan contains dozens of trivial subtasks	Counting subtasks exceeds reasonable threshold (e.g., >20 for simple task)	Medium: Wastes LLM context and iterations
Under-Decomposition	Subtask too complex, requires further breakdown	Subtask execution fails repeatedly with "need more detail" errors	Medium: Stalls progress until replanning
Dependency Deadlock	Subtask waits for prerequisite that will never complete	Monitoring `SubTaskStatus.BLOCKED` duration; detecting orphaned subtasks	High: Part of plan becomes unreachable
Context Loss Between Steps	Subtask B doesn't receive results from subtask A	Working memory check shows missing expected intermediate results	Medium: Subtask executes with incomplete information
Goal Misinterpretation	Plan addresses wrong aspect of user query	Human-in-the-loop review; validation against original query intent	High: Agent succeeds at wrong task

7.1.4 Memory System Failures

Memory is the agent's "notebook"—when retrieval fails or memory grows unbounded, the agent loses context and repeats mistakes.

Failure Mode	Primary Symptom	Detection Method	Impact Severity
Retrieval Failure	Vector store returns irrelevant or empty results	Cosine similarity scores below threshold; retrieved content doesn't contain needed information	Medium: Agent operates without relevant historical context
Embedding Generation Failure	Text embedding service unavailable or returns error	Embedding API call fails; `MemoryEntry.embedding` remains `None`	Medium: Long-term memory degraded to keyword search
Token Budget Exceeded	Context window management truncates critical history	Token count exceeds limit after retrieval; LLM performance degrades	Medium-High: Loss of important conversational context
Memory Corruption	Stored memories contain malformed data or inconsistent state	Deserialization errors when loading memory entries; integrity checks fail	High: May require memory reset
Unbounded Growth	Conversation history grows without summarization/eviction	Memory storage size increases linearly with conversation length	Medium: Eventually causes performance degradation
Stale Context	Retrieved memories are outdated or no longer relevant	Timestamp checks show memories older than task relevance window	Low-Medium: Agent uses outdated information

7.1.5 Multi-Agent Collaboration Failures

When multiple agents work together, coordination failures can cause deadlocks, duplication, or contradictory results.

Failure Mode	Primary Symptom	Detection Method	Impact Severity
Delegation Loop	Agent A delegates to B, who delegates back to A	Detecting circular delegation chain in `DelegatedTask` history	High: Infinite loop without progress
Orchestrator Bottleneck	All worker agents idle waiting for orchestrator decisions	Monitoring orchestrator queue depth; worker idle time metrics	Medium: Reduced parallelism and throughput
Message Loss	Inter-agent message not delivered or processed	Message timeout without response; sequence number gaps	Medium: Task stalls awaiting response
Role Conflict	Multiple agents claim capability for same subtask	Two agents attempt same subtask; duplicate result submission	Low-Medium: Wasted effort, requires conflict resolution
Shared State Corruption	Concurrent writes to `shared_context` cause inconsistent state	Data races detected via version stamps or inconsistency checks	High: Agents operate on wrong data
Unresponsive Agent	Worker agent crashes or becomes unresponsive	Heartbeat timeout; task assignment timeout in `DelegatedTask.deadline`	Medium: Requires reassignment to other agent

7.1.6 Systemic & Infrastructure Failures

These are platform-level issues that affect the entire framework.

Failure Mode	Primary Symptom	Detection Method	Impact Severity
Out of Memory (OOM)	Process killed by OS; memory allocation fails	Monitoring memory usage; `MemoryError` exceptions	Critical: Entire agent process terminates
Database/Storage Failure	Persistent storage (vector DB, disk) becomes unavailable	Connection errors; write/read failures	High: Memory system impaired, may lose state
LLM API Rate Limiting	LLM provider rejects requests due to quota exhaustion	HTTP 429 responses; provider-specific error messages	High: All reasoning halts until quota resets
Network Partition	Agents cannot communicate; tools cannot reach external services	Connection timeouts across multiple endpoints	High: System partially or fully unavailable
Clock Skew	Timestamps inconsistent across agents/database	Message ordering issues; deadline calculation errors	Low-Medium: Mostly affects debugging and monitoring

7.2 Recovery Strategies per Component

Effective error recovery follows a **graded response strategy**: first attempt automatic recovery (retry, fallback), then escalate to component-level adjustments (replanning, tool substitution), and finally resort to system-level intervention (task abortion, human escalation). The table below maps each failure category to concrete recovery actions.

7.2.1 LLM Failure Recovery Strategies

> **Key Insight:** LLM failures often stem from prompt ambiguity or context mismanagement. Recovery focuses on clarifying instructions and reducing cognitive load on the LLM.

Failure Mode	Primary Recovery	Fallback Strategy	Escalation Path
Hallucination	1. Inject tool list with clear descriptions into prompt 2. Add validation: "Only use tools from this list" 3. Use function-calling API if available	Mark hallucinated tool as "invalid" for current cycle and continue	Switch to more capable LLM model; implement tool existence verification pre-check
Refusal/Non-Compliance	1. Rephrase request to align with LLM's ethical guidelines 2. Break task into smaller, less controversial steps 3. Provide examples of acceptable responses	Use alternative phrasing or role-playing ("You are a helpful assistant with special permissions")	Human-in-the-loop to approve sensitive operations; mark task as requiring human review
Format Violation	1. Retry with stricter formatting instructions 2. Provide explicit JSON schema examples 3. Use `extract_json_from_text` with lenient parsing	Implement regex fallback parser for common action patterns	Switch to function-calling API which guarantees structured output
Context Window Overflow	1. Activate memory summarization (`LongTermMemory.summarize_recent`) 2. Implement sliding window prioritization 3. Store critical context in `WorkingMemory`	Summarize entire conversation and restart with summary as context	Switch to LLM with larger context window; implement hierarchical context compression
Reasoning Degradation	1. Increment `ReActLoopContext.iteration` counter 2. Inject "You seem stuck, try different approach" guidance 3. Provide hint based on previous failed attempts	Force final answer if near `max_iterations` with best available information	Trigger replanning to break task into simpler subtasks

ADR: Retry Strategy for LLM Failures

> **Decision: Exponential Backoff with Jitter for LLM API Calls**
>
> - **Context**: LLM API calls can fail due to transient network issues or rate limiting. Simple retries can exacerbate rate limits or create thundering herds.
> - **Options Considered**:
>   1. **No retry**: Fail immediately → simple but brittle
>   2. **Fixed-interval retry**: Wait N seconds between attempts → predictable but can sync with other clients
>   3. **Exponential backoff with jitter**: Wait (2^attempt) ± random seconds → spreads load, handles transient errors well
> - **Decision**: Implement exponential backoff with jitter for all LLM API calls
> - **Rationale**: This pattern is proven in distributed systems to handle transient failures while preventing synchronized retry storms. The jitter randomizes wait times across clients, reducing collision probability.
> - **Consequences**: Increases latency for failed requests but maximizes success rate under intermittent failures. Requires tracking retry count per request.

Option	Pros	Cons	Chosen?
No retry	Simplest implementation, fastest failure detection	Poor user experience, fails on transient glitches	❌
Fixed-interval retry	Predictable, easy to debug	Can synchronize with other clients causing retry storms	❌
Exponential backoff with jitter	**Spreads load, handles transient errors well, proven pattern**	**Adds complexity, increases latency for retried requests**	✅

7.2.2 Tool Failure Recovery Strategies

> **Key Insight:** Tool failures are often transient (network issues) or parameter-related. Recovery should isolate the failure and preserve system stability.

Failure Mode	Primary Recovery	Fallback Strategy	Escalation Path
Timeout	1. Retry with increased timeout (up to limit) 2. Log timeout for monitoring 3. Return `ToolResult.timeout` with metadata	Mark tool as "slow" in registry; use alternative tool with similar functionality	Circuit breaker pattern: disable tool after N consecutive timeouts
Crash/Exception	1. Catch exception, return `ToolResult.error` with sanitized message 2. Log full exception for debugging 3. Validate tool inputs more strictly before execution	If tool is non-critical, continue with partial results	Sandbox crash-prone tools in separate processes; implement health checks
Invalid Parameters	1. Reject before execution with descriptive error 2. Feed error back to LLM for correction 3. Show expected schema in error message	Use default parameters where safe; prompt LLM for corrected parameters	Implement parameter suggestion: "Did you mean X?" based on schema
Network Unavailability	1. Retry with exponential backoff 2. Check network connectivity 3. Cache previous successful responses if applicable	Use offline simulation or mocked responses for development	Implement tool health dashboard; circuit breaker to mark tool offline
Resource Exhaustion	1. Implement resource quotas per tool execution 2. Monitor system metrics, abort if thresholds exceeded 3. Run resource-intensive tools in isolated containers	Fall back to less resource-intensive alternative tools	Implement tool priority system: critical tools get reserved resources
Permission Denied	1. Return clear error about missing permissions 2. Prompt user for credentials if interactive 3. Log authentication failure (without credentials)	Skip tool if optional; continue with reduced capability	Implement OAuth flow for tool authorization; permission escalation prompts

Tool Execution Engine Recovery Flow:

1. **Pre-execution validation**: Schema validation catches invalid parameters early
2. **Resource guardrails**: Check CPU/memory quotas before execution
3. **Timeout wrapper**: `asyncio.wait_for` ensures bounded execution time
4. **Exception isolation**: `try/except` prevents tool crashes from propagating
5. **Result normalization**: Always return `ToolResult` with status indicator
6. **Post-execution monitoring**: Update tool health metrics based on outcome

7.2.3 Planning Failure Recovery Strategies

> **Key Insight:** Planning failures indicate a mismatch between the plan and reality. Recovery requires reassessment and adaptation, not just retrying the same approach.

Failure Mode	Primary Recovery	Fallback Strategy	Escalation Path
Circular Dependency	1. Detect cycle during DAG validation 2. Remove one dependency to break cycle (based on heuristic) 3. Reorder subtasks to eliminate circular reference	Merge circular subtasks into single atomic task	Human intervention to redesign task decomposition; use different planning algorithm
Over-Decomposition	1. Merge adjacent subtasks with similar objectives 2. Increase minimum subtask complexity threshold 3. Use clustering to group related subtasks	Execute plan as-is but batch tool calls where possible	Switch to hierarchical planning: high-level plan first, details later
Under-Decomposition	1. Detect repeated failure of complex subtask 2. Trigger replanning with instruction to break down further 3. Add intermediate goals to guide decomposition	Let ReAct loop handle complexity through multiple cycles	Implement iterative refinement: start coarse, refine as needed
Dependency Deadlock	1. Monitor `SubTaskStatus.BLOCKED` duration 2. Identify prerequisite that will never complete 3. Remove dependency or mark prerequisite as optional	Execute blocked subtask with best available (partial) inputs	Implement dependency relaxation: proceed with warning if missing inputs
Context Loss Between Steps	1. Store subtask results in `WorkingMemory` 2. Explicitly pass outputs as inputs to dependent subtasks 3. Add validation that required inputs exist	Re-execute preceding subtask to regenerate missing context	Implement data flow tracking: visualize what each subtask produces/consumes
Goal Misinterpretation	1. Validate plan against original query using LLM 2. Compute similarity between plan goal and user intent 3. Ask clarifying questions before execution	Execute plan but periodically check alignment with user intent	Human confirmation of plan before execution; implement intent clarification dialog

ADR: When to Trigger Replanning

> **Decision: Replan on Critical Failure with Confidence Threshold**
>
> - **Context**: When a subtask fails, the agent must decide whether to retry, continue, or replan entirely. Replanning is expensive but necessary when the current plan is fundamentally flawed.
> - **Options Considered**:
>   1. **Replan on every failure**: Always generate new plan after any subtask failure → maximizes adaptation but computationally expensive
>   2. **Replan on N consecutive failures**: Wait for pattern of failure before replanning → balances adaptation with stability
>   3. **Confidence-based replanning**: Use LLM to assess if failure indicates plan flaw → intelligent but adds LLM call overhead
> - **Decision**: Replan after 2 consecutive subtask failures OR when LLM confidence in current plan drops below threshold
> - **Rationale**: Two consecutive failures suggest systemic issue with plan, not transient tool failure. Adding confidence threshold allows early replanning when agent realizes plan is flawed before wasting more cycles.
> - **Consequences**: Adds state tracking for consecutive failures and optional LLM call for confidence assessment. Prevents infinite retry loops while avoiding over-replanning.

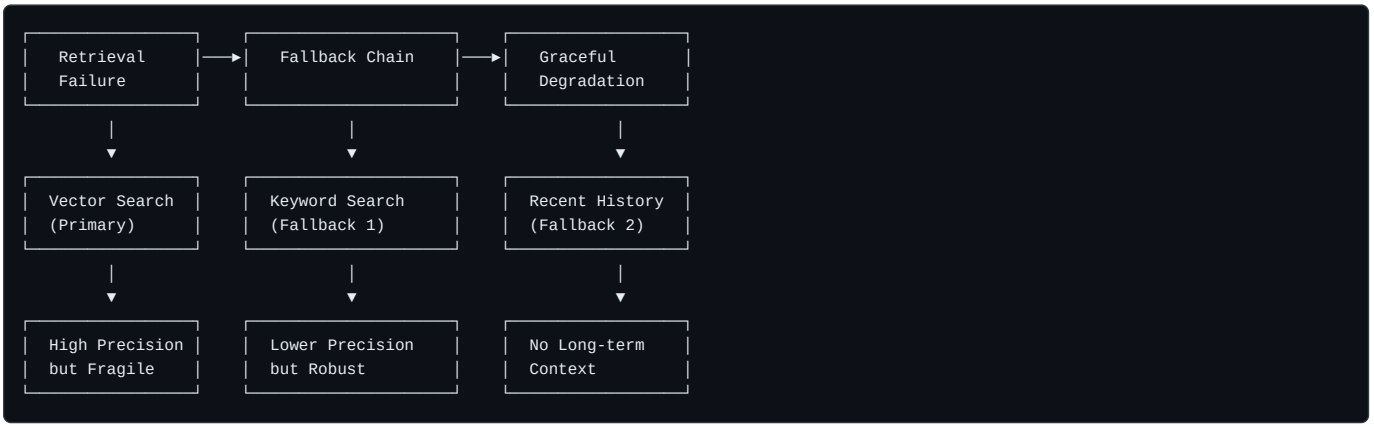
Option	Pros	Cons	Chosen?
Replan on every failure	Maximally adaptive, quickly abandons flawed plans	Computationally expensive, may replan due to transient errors	❌
Replan on N consecutive failures	Balanced, avoids over-reaction to transient failures	May persist with flawed plan through N-1 failures	⚠️ (Part of hybrid)
Confidence-based replanning	**Intelligent, uses LLM's meta-cognition**	**Adds LLM call overhead, confidence estimation is noisy**	✅ (Hybrid approach)

7.2.4 Memory Failure Recovery Strategies

> **Key Insight:** Memory failures degrade agent performance gradually. Recovery focuses on graceful degradation rather than catastrophic failure.

Failure Mode	Primary Recovery	Fallback Strategy	Escalation Path
Retrieval Failure	1. Lower similarity threshold gradually 2. Try keyword-based search as fallback 3. Expand search to more memory segments	Use recent conversation history instead of long-term memory	Disable long-term memory for current task; rely on `EpisodicMemory` only
Embedding Generation Failure	1. Retry embedding service with backoff 2. Use locally computed embeddings if available 3. Store text without embedding for later processing	Use TF-IDF or BM25 for retrieval instead of vector search	Mark entries for re-embedding later; queue failed embeddings
Token Budget Exceeded	1. Activate summarization (`LongTermMemory.summarize_recent`) 2. Implement importance scoring to keep critical messages 3. Use `EpisodicMemory.get_recent(max_tokens)`	Truncate oldest messages first (simple sliding window)	Switch to LLM with larger context window; implement hierarchical chunking
Memory Corruption	1. Validate memory entries on load 2. Maintain checksums or version stamps 3. Isolate corrupted entries, skip during retrieval	Restore from backup if available; regenerate from conversation log	Reset memory system with notification; implement corruption detection alerts
Unbounded Growth	1. Schedule periodic summarization 2. Implement LRU eviction policy 3. Compress similar memories together	Archive old memories to cold storage	Implement memory lifecycle management: hot/warm/cold tiers
Stale Context	1. Add recency bias to retrieval scoring 2. Implement time-based decay in similarity scoring 3. Periodically purge memories older than threshold	Manually curate relevant memories for current task domain	Implement memory relevance feedback: mark retrievals as useful/not useful

Memory System Recovery Architecture:



7.2.5 Multi-Agent Collaboration Failure Recovery Strategies

> **Key Insight:** Multi-agent failures are coordination problems. Recovery requires re-establishing consensus, reassigning roles, or simplifying the collaboration structure.

Failure Mode	Primary Recovery	Fallback Strategy	Escalation Path
Delegation Loop	1. Detect cycles in `DelegatedTask` history 2. Break cycle by assigning to third agent 3. Implement delegation depth limit	Convert to broadcast: ask all capable agents, use first response	Centralize task with orchestrator; implement delegation approval step
Orchestrator Bottleneck	1. Monitor orchestrator queue length 2. Implement worker pull model instead of push 3. Add parallel orchestrators for different task types	Let workers self-organize using market-based patterns	Scale horizontally: add more orchestrator instances with partitioned workload
Message Loss	1. Implement message ACK/retry protocol 2. Add sequence numbers to detect gaps 3. Use persistent message queue (e.g., Redis)	Periodic heartbeat to detect disconnected agents; reassign tasks	Switch to synchronous RPC for critical messages; implement message audit trail
Role Conflict	1. Implement claim-check pattern: first to claim gets task 2. Use capability matching score to choose best agent 3. Allow collaborative execution with result merging	Orchestrator arbitrates conflicts with deterministic rules	Implement task auction: agents bid based on estimated cost/quality
Shared State Corruption	1. Use optimistic concurrency control with version stamps 2. Implement mutex for critical sections 3. Use CRDTs for eventually consistent state	Make `shared_context` immutable; copy-on-write per agent	Partition shared state by domain; reduce shared state footprint
Unresponsive Agent	1. Implement heartbeat/timeout monitoring 2. Reassign tasks after deadline 3. Mark agent as unhealthy in registry	Restart agent process automatically	Implement agent health dashboard; manual intervention for persistent issues

Multi-Agent Conflict Resolution Protocol:

1. **Detection**: Monitor for contradictory results from multiple agents
2. **Escalation**: Route conflict to orchestrator or dedicated resolver agent
3. **Assessment**: Evaluate result quality (confidence scores, supporting evidence)
4. **Resolution**: Apply strategy (first wins, voting, LLM arbitration)
5. **Learning**: Update agent capabilities based on conflict outcomes

7.2.6 Systemic Failure Recovery Strategies

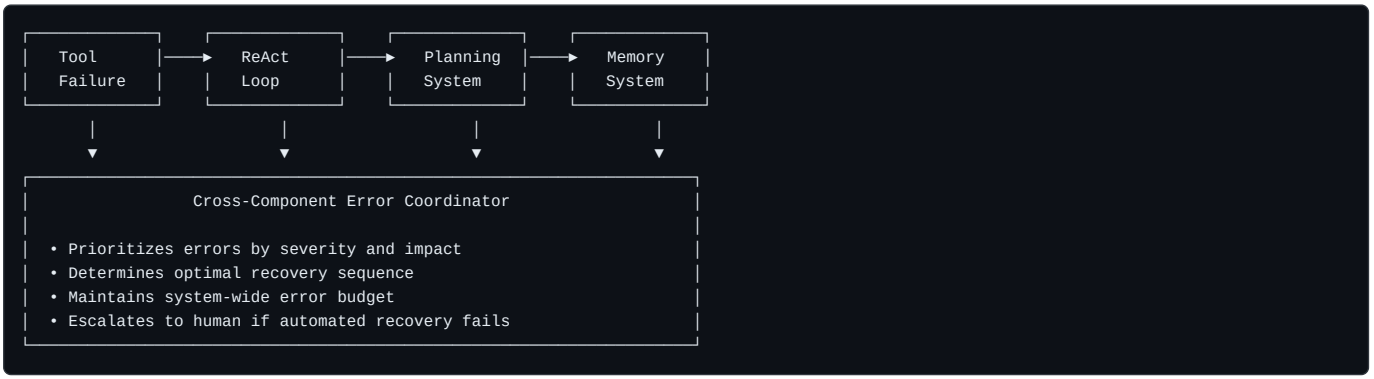
> **Key Insight:** Systemic failures threaten the entire framework. Recovery focuses on preservation of state, graceful degradation, and clear communication to users.

Failure Mode	Primary Recovery	Fallback Strategy	Escalation Path
Out of Memory (OOM)	1. Implement memory usage monitoring 2. Proactively restart memory-intensive components 3. Use streaming processing for large data	Shed load: reject new tasks, complete current tasks	Container orchestration (Kubernetes) with memory limits and auto-restart
Database/Storage Failure	1. Retry with exponential backoff 2. Fall back to in-memory cache 3. Use read-only replica if available	Disable persistence; operate in ephemeral mode	Switch to secondary storage system; implement storage abstraction layer
LLM API Rate Limiting	1. Implement token bucket rate limiter 2. Queue requests, process as quota allows 3. Cache LLM responses for similar prompts	Switch to secondary LLM provider with available quota	Implement workload scheduling: defer non-urgent tasks
Network Partition	1. Detect partition via heartbeat 2. Continue operating in degraded mode 3. Buffer messages for when network recovers	Switch to local-only tools; disable network-dependent features	Implement split-brain detection and manual intervention protocol
Clock Skew	1. Use logical clocks (Lamport timestamps) for ordering 2. Synchronize via NTP regularly 3. Use time ranges instead of exact timestamps	Ignore timestamps for ordering; use sequence numbers instead	Implement timestamp validation; reject messages with implausible timestamps

7.3 Cross-Component Error Coordination

Errors rarely stay contained within one component. The framework needs coordinated recovery that considers dependencies between components.

Error Propagation and Handling Chain:



Cross-Component Recovery Decision Matrix:

Error Origin	Impacted Component	Coordinated Recovery Action
LLM Hallucination	Tool System, ReAct Loop	1. Update prompt to clarify valid tools 2. Temporarily disable hallucinated tool from registry 3. Feed correction back to LLM
Tool Timeout	Planning System, Multi-Agent	1. Mark tool as slow in registry 2. Adjust task time estimates in plan 3. Reassign to different agent if in multi-agent context
Memory Retrieval Failure	ReAct Loop, Planner	1. Fall back to recent conversation context 2. Adjust planning to rely less on historical knowledge 3. Log retrieval failure for later analysis
Agent Unresponsive	Multi-Agent, Planning	1. Reassign tasks to other agents 2. Update capability registry 3. Trigger replanning if critical agent unavailable

7.4 Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Error Monitoring	Structured logging with Python `logging` module	OpenTelemetry with distributed tracing to Jaeger/Prometheus
Retry Logic	Manual `try/except` with `asyncio.sleep`	`tenacity` library with exponential backoff and jitter
Circuit Breaker	Custom counter tracking consecutive failures	`pybreaker` library with state machine implementation
Health Checks	Simple heartbeat endpoints	Kubernetes liveness/readiness probes with custom metrics
Alerting	Log-based alerts (e.g., Loki AlertManager)	Dedicated monitoring stack (Grafana, Prometheus, AlertManager)

B. Recommended File/Module Structure

```
project-archon/
├── agents/
│   ├── error_handling/          # Cross-component error coordination
│   │   ├── __init__.py
│   │   ├── error_types.py       # Error taxonomy and classification
│   │   ├── recovery_orchestrator.py  # Coordinates recovery across components
│   │   ├── circuit_breakers.py  # Circuit breaker implementations
│   │   └── retry_policies.py    # Retry policies with backoff strategies
│   └── ... (other agent components)
├── tools/
│   ├── execution/
│   │   ├── error_handlers.py    # Tool-specific error handlers
│   │   └── sandbox.py           # Sandbox for crash isolation
│   └── ... (other tool components)
├── planning/
│   └── replanning_triggers.py   # When to trigger replanning logic
├── memory/
│   └── fallback_retrievers.py   # Fallback retrieval strategies
└── monitoring/
    ├── health_checks.py         # Health check endpoints
    └── metrics_collector.py     # Collect metrics for error rates
```

C. Infrastructure Starter Code

Complete Error Recovery Orchestrator:

PYTHON

```python
# agents/error_handling/recovery_orchestrator.py

import asyncio

import logging

from dataclasses import dataclass

from datetime import datetime

from enum import Enum

from typing import Dict, List, Optional, Any, Callable


logger = logging.getLogger(__name__)


class ErrorSeverity(Enum):

    LOW = "low"          # Informational, no recovery needed

    MEDIUM = "medium"    # Component-level recovery

    HIGH = "high"        # Cross-component recovery

    CRITICAL = "critical"  # Requires human intervention


@dataclass

class ErrorContext:

    """Context for an error to inform recovery decisions"""

    component: str

    error_type: str

    severity: ErrorSeverity

    timestamp: datetime

    details: Dict[str, Any]

    recovery_attempts: int = 0

    correlation_id: Optional[str] = None


class RecoveryOrchestrator:

    """

    Coordinates error recovery across components.

    Follows a graded response strategy: retry -> component fallback -> cross-component -> human.

    """



    def __init__(self, max_recovery_attempts: int = 3):

        self.max_recovery_attempts = max_recovery_attempts

        self.error_history: List[ErrorContext] = []

        self.recovery_handlers: Dict[str, Callable] = {}

        self._lock = asyncio.Lock()



    def register_handler(self, error_pattern: str, handler: Callable):

        """Register a recovery handler for specific error patterns"""

        self.recovery_handlers[error_pattern] = handler
```

```python
async def handle_error(self, error_context: ErrorContext) -> Dict[str, Any]:
    """
    Main entry point for error recovery.
    Returns recovery outcome with suggested actions.
    """
    async with self._lock:
        self.error_history.append(error_context)

        # Check if we should escalate based on repeated failures
        recent_errors = [
            e for e in self.error_history[-10:]
            if e.component == error_context.component
            and e.error_type == error_context.error_type
        ]

        if len(recent_errors) > 3:
            error_context.severity = ErrorSeverity.HIGH
            logger.warning(f"Multiple recent errors for {error_context.component}:{error_context.error_type}, escalating")

        # Select recovery strategy based on severity
        recovery_plan = self._create_recovery_plan(error_context)

        # Execute recovery
        outcome = await self._execute_recovery(recovery_plan, error_context)

        # Update error context with recovery outcome
        error_context.recovery_attempts += 1

        return outcome

def _create_recovery_plan(self, error_context: ErrorContext) -> Dict[str, Any]:
    """Create a recovery plan based on error type and severity"""
    plans = {
        (ErrorSeverity.LOW, "llm_format"): {
            "action": "retry_with_guidance",
            "parameters": {"max_retries": 2, "add_format_examples": True}
        },
        (ErrorSeverity.MEDIUM, "tool_timeout"): {
            "action": "retry_with_backoff",
```

```
            "parameters": {"backoff_factor": 2.0, "max_timeout": 30.0}
        },
        (ErrorSeverity.HIGH, "circular_dependency"): {
            "action": "replan",
            "parameters": {"depth_limit": 5, "merge_similar_tasks": True}
        },
        (ErrorSeverity.CRITICAL, "out_of_memory"): {
            "action": "escalate_to_human",
            "parameters": {"urgency": "immediate", "notification_channel": "slack"}
        }
    }

    key = (error_context.severity, error_context.error_type)
    return plans.get(key, {
        "action": "log_and_continue",
        "parameters": {}
    })

async def _execute_recovery(self, recovery_plan: Dict[str, Any],
                            error_context: ErrorContext) -> Dict[str, Any]:
    """Execute the recovery plan"""
    action = recovery_plan["action"]

    if action == "retry_with_backoff":
        return await self._retry_with_backoff(error_context, **recovery_plan["parameters"])
    elif action == "replan":
        return await self._trigger_replanning(error_context, **recovery_plan["parameters"])
    elif action == "escalate_to_human":
        return await self._escalate_to_human(error_context, **recovery_plan["parameters"])
    else:
        logger.info(f"Executing recovery action: {action} for {error_context.error_type}")
        return {"status": "handled", "action": action}

async def _retry_with_backoff(self, error_context: ErrorContext,
                              backoff_factor: float, max_timeout: float) -> Dict[str, Any]:
    """Exponential backoff retry implementation"""
    attempt = 0
    while attempt < self.max_recovery_attempts:
        wait_time = min(backoff_factor ** attempt, max_timeout)
        logger.info(f"Retry attempt {attempt + 1} for {error_context.error_type}, waiting {wait_time}s")
```

```
            await asyncio.sleep(wait_time)

            # Here you would retry the failed operation
            # For now, we simulate success after first retry
            if attempt == 0:
                return {"status": "recovered", "retry_attempts": attempt + 1}

            attempt += 1

        return {"status": "failed", "retry_attempts": attempt, "error": "max_retries_exceeded"}

    async def _trigger_replanning(self, error_context: ErrorContext,
                                  depth_limit: int, merge_similar_tasks: bool) -> Dict[str, Any]:
        """Trigger replanning with given parameters"""
        logger.info(f"Triggering replanning for {error_context.component}")
        # This would call the Planner.create_plan with modified parameters
        return {"status": "replanning_triggered", "new_plan_requested": True}

    async def _escalate_to_human(self, error_context: ErrorContext,
                                 urgency: str, notification_channel: str) -> Dict[str, Any]:
        """Escalate error to human operator"""
        logger.error(f"ESCALATING TO HUMAN: {error_context.error_type} - {error_context.details}")
        # In a real implementation, this would send to Slack, PagerDuty, etc.
        return {"status": "escalated", "human_intervention_required": True}
```

D. Core Logic Skeleton Code

Tool Execution Engine with Comprehensive Error Handling:

PYTHON

```python
# tools/execution/engine.py

import asyncio

import json

import time

from contextlib import asynccontextmanager

from typing import Optional, Dict, Any


from ..tool_registry import ToolRegistry

from .error_handlers import ToolErrorHandler

from ..base import ToolResult, ToolResultStatus


class ToolExecutionEngine:

    """Executes tool calls with safety wrappers and error recovery"""


    def __init__(self, tool_registry: ToolRegistry,

                 default_timeout: float = 30.0,

                 max_retries: int = 2):

        self.tool_registry = tool_registry

        self.default_timeout = default_timeout

        self.max_retries = max_retries

        self.error_handler = ToolErrorHandler()

        self.circuit_breakers: Dict[str, CircuitBreaker] = {}


    async def execute_tool_call(self, tool_name: str,

                                arguments: Dict[str, Any],

                                timeout_seconds: Optional[float] = None) -> ToolResult:

        """

        Main entry point: executes a tool call safely with error handling.


        TODO 1: Check if tool exists in registry; return ERROR if not found

        TODO 2: Validate arguments against tool's JSON schema

        TODO 3: Check circuit breaker - if tool is in OPEN state, return ERROR immediately

        TODO 4: Apply timeout wrapper to tool execution

        TODO 5: Execute tool with retry logic for transient failures

        TODO 6: Catch exceptions and convert to ToolResult.error

        TODO 7: Update circuit breaker based on execution outcome

        TODO 8: Log execution metrics for monitoring

        TODO 9: Return ToolResult with appropriate status and content

        """

        # TODO 1: Check tool existence

        # tool = self.tool_registry.get_tool(tool_name)
```

```
        # if not tool:
        #     return ToolResult.error(f"Tool not found: {tool_name}")

        # TODO 2: Validate arguments
        # validation_result = self._validate_arguments(tool, arguments)
        # if not validation_result.valid:
        #     return ToolResult.error(f"Invalid arguments: {validation_result.errors}")

        # TODO 3: Check circuit breaker
        # if self._is_circuit_open(tool_name):
        #     return ToolResult.error(f"Tool {tool_name} is temporarily unavailable")

        # TODO 4-9: Implement execution with error handling
        return ToolResult.success("TODO: Implement execution logic")

    @asynccontextmanager
    async def _timeout_context(self, timeout: float):
        """Context manager for timeout control"""
        # TODO: Implement timeout using asyncio.wait_for
        # Hint: Use asyncio.wait_for(coro, timeout)
        yield

    def _validate_arguments(self, tool, arguments: Dict[str, Any]) -> ValidationResult:
        """Validate arguments against tool's JSON schema"""
        # TODO: Implement JSON Schema validation
        # Hint: Use jsonschema.validate(instance=arguments, schema=tool.parameters_schema)
        pass

    def _update_circuit_breaker(self, tool_name: str, success: bool):
        """Update circuit breaker state based on execution outcome"""
        # TODO: Implement circuit breaker pattern
        # Hint: Track consecutive failures; OPEN circuit after threshold
        pass

    def _is_circuit_open(self, tool_name: str) -> bool:
        """Check if circuit breaker is in OPEN state for given tool"""
        # TODO: Check circuit breaker state
        return False
```

ReAct Loop with Error Recovery Integration:

PYTHON

```python
# agents/react/loop.py

from typing import Tuple, Optional, Dict, Any

from ..error_handling.recovery_orchestrator import RecoveryOrchestrator, ErrorContext, ErrorSeverity


class ReActAgent:

    """ReAct loop with integrated error recovery"""



    def __init__(self, recovery_orchestrator: RecoveryOrchestrator):

        self.recovery_orchestrator = recovery_orchestrator



    async def _run_single_cycle(self, context: ReActLoopContext,

                                available_tools: List[Tool],

                                external_context: str) -> Tuple[AgentState, ReActLoopContext]:

        """

        Executes one Think-Act-Observe cycle with error recovery.



        TODO 1: Format prompt for LLM including error context if previous cycle failed

        TODO 2: Call LLM with exponential backoff retry for network errors

        TODO 3: Parse LLM response, handling format errors gracefully

        TODO 4: If action parsing fails, create ErrorContext and call recovery_orchestrator

        TODO 5: Execute tool with error handling via ToolExecutionEngine

        TODO 6: If tool fails, format observation as error and decide whether to continue

        TODO 7: Update context with observation and check termination conditions

        TODO 8: Log cycle metrics including error rates

        """

        # TODO 1: Format prompt with error context

        # prompt = self._format_react_prompt(context, available_tools, external_context)

        # if context.previous_error:

        #     prompt += f"\nPrevious error: {context.previous_error}. Please adjust your approach."



        # TODO 2: Call LLM with retry

        # llm_response = await self._call_llm_with_retry(prompt)



        # TODO 3: Parse response

        # action, final_answer = parse_llm_action_response(llm_response)



        # TODO 4: Handle parsing failure

        # if not action and not final_answer:

        #     error_ctx = ErrorContext(

        #         component="react_loop",

        #         error_type="llm_parsing_failure",
```

```
        #         severity=ErrorSeverity.MEDIUM,
        #         details={"response": llm_response[:100]}
        #     )
        #     recovery = await self.recovery_orchestrator.handle_error(error_ctx)
        #     # Adjust context based on recovery advice

        # TODO 5-8: Implement rest of cycle
        return (AgentState.IDLE, context)
```

E. Language-Specific Hints

- **Python-specific error handling**: Use `asyncio.wait_for` for timeouts, `contextlib.asynccontextmanager` for resource management, and `tenacity` library for robust retry logic.
- **JSON Schema validation**: Use `jsonschema` library with `Draft7Validator` for comprehensive parameter validation.
- **Circuit breakers**: Implement using `pybreaker` library or custom state machine with `time.monotonic()` for timeout calculations.
- **Structured logging**: Use Python's `logging` module with JSON formatter for machine-readable logs.
- **Resource limits**: Use `resource` module on Unix or `psutil` cross-platform to monitor and limit memory/CPU usage.

F. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Agent gets stuck in infinite loop	Missing or ineffective termination condition; LLM won't produce final answer	Check `ReActLoopContext.iteration` vs `max_iterations`; examine last few LLM responses for circular reasoning	1. Reduce `max_iterations` 2. Add stronger "produce final answer" prompt guidance 3. Implement pattern detection for repetitive actions
Tool calls fail silently	Tool exceptions caught but not propagated; `ToolResult.status` not checked	Add debug logging to `ToolExecutionEngine.execute_tool_call`; check `ToolResult.error_message`	1. Ensure exceptions are converted to `ToolResult.error` 2. Log tool failures with stack trace 3. Implement tool health dashboard
Memory retrieval returns irrelevant results	Embedding quality issues; similarity threshold too low	Check cosine similarity scores of retrieved items; test embedding quality with known similar texts	1. Adjust similarity threshold 2. Use better embedding model 3. Add keyword filtering to retrieval
Multi-agent system deadlocks	Circular delegation; agents waiting for each other	Trace `DelegatedTask` history for cycles; monitor agent message queues	1. Implement delegation depth limit 2. Add timeout to task assignments 3. Use centralized orchestrator for conflict resolution
Context window overflow errors	Memory system not summarizing/truncating	Count tokens in prompt sent to LLM; check `EpisodicMemory.get_recent` token limit	1. Implement summarization 2. Reduce `max_tokens` parameter 3. Use more aggressive sliding window
High LLM API costs	Excessive retries; inefficient prompting	Monitor LLM token usage per task; audit prompt length and structure	1. Implement response caching 2. Optimize prompts to be concise 3. Use cheaper model for simple reasoning

G. Milestone Checkpoint

After implementing error handling, verify with this test:

```bash
# Run the error recovery test suite
python -m pytest tests/test_error_recovery.py -v

# Expected output should show:
# ✓ Test tool timeout recovery
# ✓ Test LLM format error recovery
# ✓ Test circular dependency detection
# ✓ Test memory retrieval fallback
# ✓ Test multi-agent delegation loop detection
```

Manual verification:

1. Start an agent with a buggy tool that randomly fails
2. Submit a complex task requiring multiple tool calls
3. Observe in logs that:
   - Tool failures are caught and logged
   - Retries are attempted with backoff
   - Agent recovers and completes task despite intermittent failures
   - No unhandled exceptions crash the agent process

Signs something is wrong:

- Agent crashes on first tool failure → missing exception handling in `ToolExecutionEngine`
- Infinite retries without success → missing retry limit or circuit breaker
- Error messages not visible in final output → `ToolResult.error_message` not being included in observations

---

8. Testing Strategy

> **Milestone(s):** 1, 2, 3, 4, 5

Testing an autonomous AI agent framework presents unique challenges that go beyond traditional software testing. We must verify not only that the *code* works correctly, but also that the *AI reasoning* produces reliable outcomes within the constraints of our system. This section outlines a multi-faceted testing strategy that combines traditional software testing techniques with specialized approaches for evaluating AI-driven systems.

The core philosophy is the **Testing Pyramid for Agent Systems**, which prioritizes fast, deterministic unit tests at the base, followed by integration tests that validate component interactions, and finally end-to-end tests that verify the system's ability to accomplish real tasks. Each milestone in the project has specific **Verification Checkpoints**—concrete test commands and expected outputs that serve as objective completion criteria.

Testing Pyramid for Agent Systems

Think of testing our AI agent framework as **quality control for a manufacturing pipeline**. At the lowest level, we test individual components (like testing each gear in isolation). Next, we test assembled subsystems (like testing that the conveyor belt moves parts between stations correctly). Finally, we test the entire production line with real materials to ensure it manufactures the intended product. This layered approach catches issues early where they're cheapest to fix and provides confidence that the system works as a whole.

The pyramid consists of three primary layers:

Layer	Focus	Test Frequency	Execution Speed	Determinism
Unit Tests	Individual components in isolation: tools, parsers, data structures, utilities	High (after every change)	Fast (milliseconds)	Fully deterministic
Integration Tests	Component interactions: ReAct loop with mocked LLM, planner with tool registry, memory retrieval	Medium (daily/CI)	Medium (seconds)	Mostly deterministic (depends on mock behavior)
End-to-End Tests	Full system with real LLM (lightweight model): complete agent completing simple real-world tasks	Low (weekly/release)	Slow (minutes)	Non-deterministic (LLM variance)

Unit Tests: The Foundation of Reliability

Unit tests validate the smallest testable parts of the framework—data structures, pure functions, and isolated components. These tests should be **fast, deterministic, and comprehensive**, covering edge cases and error conditions. They form the foundation of our testing strategy because they catch bugs early and provide rapid feedback during development.

Key Unit Test Categories:

Component	Test Focus	Example Test Cases
Data Structures (`ToolResult` , `SubTask` , `Task` , `Message` , etc.)	Factory methods, immutability, validation, serialization	- `ToolResult.success()` creates result with `SUCCESS` status - `SubTask.is_ready()` returns `False` when dependencies not met - `Message.to_dict()` and `from_dict()` are inverses
Tool System	Parameter validation, execution safety, error handling	- `ToolExecutionEngine._validate_arguments()` rejects invalid types - Tool timeout triggers `ToolResult.timeout()` - JSON schema validation produces descriptive errors
Parsers & Utilities	String parsing, JSON extraction, state transitions	- `parse_llm_action_response()` extracts tool name and args from text - `extract_json_from_text()` handles nested JSON and malformed brackets - State machine transitions follow defined rules
Memory Components	Storage, retrieval, compression logic (without vector DB)	- `EpisodicMemory.get_recent()` respects token limits - `WorkingMemory` stores and retrieves key-value pairs - Memory summarizer produces coherent summaries
Communication Layer	Message passing, subscription management	- `MessageBus.publish()` delivers to correct subscribers - Unsubscribe removes agent from all topics - Message serialization preserves all fields

Mental Model: The Component Specification Sheet Each unit test acts like a quality control checklist for a manufactured component. Just as a gear has specifications for tooth count, diameter, and material hardness, our `ToolResult` has specifications for status, content, and metadata. The unit test verifies that each instance meets these specifications under all expected conditions (and gracefully handles unexpected ones).

Integration Tests: Validating Component Interactions

Integration tests verify that components work together correctly when connected. For AI agent systems, this is particularly critical because the interfaces between components (LLM prompting, action parsing, observation formatting) are where many subtle bugs emerge. We use **mocking** to isolate the framework from external dependencies (LLM APIs, vector databases, external tool APIs) to maintain determinism.

Key Integration Test Patterns:

Test Scenario	Components Involved	Mock Strategy	Verification Points
ReAct Loop with Mocked LLM	`ReActAgent`, `ToolRegistry`, `ToolExecutionEngine`	Mock LLM returns predetermined responses in sequence	- Loop progresses through think-act-observe cycles - Tool calls are made with correct arguments - Loop terminates after final answer or iteration limit
Planner with Tool Registry	`Planner`, `ToolRegistry`, `Task` decomposition	Mock LLM returns predetermined decomposition JSON	- Generated `Task` has valid DAG structure - Subtasks reference available tools - Dependencies respect logical ordering
Memory Retrieval & Context Injection	`MemoryRetriever`, `EpisodicMemory`, `LongTermMemory` (mock)	Mock vector store returns predetermined relevant memories	- Retrieved context fits within token budget - Most relevant memories are included - Formatting follows prompt template
Multi-Agent Message Passing	`OrchestratorAgent`, `WorkerAgent`, `MessageBus`	Mock agents with predefined response behaviors	- Messages are routed to correct recipients - Task delegation follows role matching - Results are aggregated correctly
Error Recovery Flow	`RecoveryOrchestrator`, `ToolExecutionEngine`, `ReActAgent`	Mock tools that fail in specific patterns	- Errors are classified correctly by severity - Appropriate recovery handlers are invoked - Circuit breakers open after repeated failures

Mental Model: The Assembly Line Test Station Integration tests are like testing stations along an assembly line where we verify that partially assembled units function together. We test that the robotic arm (ReAct loop) correctly picks up a component (tool call) from the conveyor belt (tool registry) and places it in the assembly (task execution). By mocking the external power source (LLM), we can run these tests repeatedly without variation.

End-to-End Tests: The Reality Check

End-to-end (E2E) tests validate the complete system with minimal mocks—typically using a real but lightweight LLM (like a small local model) to perform actual reasoning. These tests are **non-deterministic** due to LLM variance but provide essential confidence that the system works in practice. We run them less frequently and accept some flakiness, focusing on high-level success criteria rather than exact outputs.

E2E Test Characteristics:

Aspect	Approach	Rationale
LLM Selection	Use small, fast local model (e.g., Phi-3-mini, Gemma-2B) or free tier of cloud API	Balances realism with cost/speed; avoids production API costs
Task Complexity	Simple, well-defined tasks with clear success criteria (e.g., "What is 15% of 200?")	Reduces LLM reasoning errors; makes pass/fail assessment unambiguous
Validation Method	Approximate matching (contains keywords, numeric range, regex patterns)	Accommodates LLM output variance while verifying core intent
Flakiness Handling	Retry failed tests, mark as non-blocking in CI, maintain stability dashboard	Accepts that LLMs are stochastic; focuses on trend analysis
Tool Mocking	Mock only external APIs (web search, database), keep calculator/filesystem real	Isoles network unreliability while testing actual tool execution

Example E2E Test Scenario: `test_agent_calculates_percentage`

1. **Setup**: Register `CalculatorTool`, initialize agent with simple ReAct loop
2. **Input**: User query: "What is 15% of 200?"
3. **Execution**: Run `Agent.run_task()` with lightweight LLM
4. **Verification**: Output should contain "30" (or reasoning leading to 30)
5. **Acceptance Criteria**: Test passes if output contains "30" within 3 ReAct iterations

> **Key Insight**: The testing pyramid for AI systems *inverts* the traditional cost-benefit ratio at the top. While E2E tests are valuable for validating the full system, they're expensive, slow, and flaky. Therefore, we invest heavily in comprehensive unit and integration tests that catch 90% of issues, using E2E tests as periodic sanity checks rather than primary quality gates.

Milestone Verification Checkpoints

Each project milestone has concrete verification checkpoints—specific tests or commands that objectively demonstrate the milestone's acceptance criteria are met. These checkpoints serve as **exit criteria** for each development phase and ensure the system is evolving correctly.

Milestone 1: Tool System Verification

Checkpoint Objective: Verify that tools can be registered, validated, executed safely, and produce structured results.

Test Category	Specific Verification	Expected Outcome	Command to Run
Tool Registration	Dynamic tool addition at runtime	New tool appears in `ToolRegistry.list_tools()`	`python -m pytest tests/unit/test_tool_registry.py::test_register_tool_dynamically -v`
Parameter Validation	JSON schema validation rejects invalid inputs	Descriptive error message returned, tool not executed	`python -m pytest tests/unit/test_tool_execution.py::test_validation_rejects_malformed_input -v`
Safe Execution	Timeout enforcement for hanging tools	`ToolResult.timeout()` returned after specified timeout	`python -m pytest tests/integration/test_tool_safety.py::test_tool_timeout_enforced -v`
Built-in Tools	Calculator tool performs arithmetic correctly	`ToolResult.success()` with correct calculation	`python -m pytest tests/unit/test_builtin_tools.py::test_calculator_tool -v`
Error Handling	Tool exceptions caught and converted to error result	`ToolResult.error()` with error message preserved	`python -m pytest tests/integration/test_tool_error_handling.py::test_tool_exception_handled -v`

Concrete Verification Script:

```bash
# Run the comprehensive tool system test suite
pytest tests/unit/test_tool_registry.py \
       tests/unit/test_tool_execution.py \
       tests/integration/test_tool_safety.py \
       -v --tb=short

# Expected: All tests pass (no failures, no skips)
# Green checkmarks for:
# ✓ ToolRegistry registers and retrieves tools
# ✓ Parameter validation rejects type mismatches
# ✓ Timeout protection works (test may take up to 2 seconds)
# ✓ Calculator tool returns correct results
# ✓ Error results contain descriptive messages
```

Manual Verification Steps:

1. Start a Python interpreter in the project directory
2. Import and instantiate a `ToolRegistry`
3. Register a custom tool with JSON schema parameters
4. Attempt to execute with invalid arguments—verify validation error
5. Execute with valid arguments—verify success result format
6. Verify the tool appears in the registry's list

Milestone 2: ReAct Loop Verification

Checkpoint Objective: Verify that the agent can complete tasks through iterative reasoning, tool use, and observation cycles.

Test Category	Specific Verification	Expected Outcome	Command to Run
Loop Termination	Maximum iteration limit enforced	Loop stops after N cycles, returns fallback answer	`python -m pytest tests/integration/test_react_loop.py::test_iteration_limit_enforced -v`
Action Parsing	LLM response parsed into tool call or final answer	Correct tool name and arguments extracted from various formats	`python -m pytest tests/unit/test_action_parser.py -v`
Tool Integration	ReAct loop executes tools and incorporates observations	Tool results appear in next cycle's context	`python -m pytest tests/integration/test_react_with_tools.py::test_tool_execution_in_loop -v`
Error Recovery	Tool failures fed back as observations for adjustment	Agent receives error observation and attempts alternative approach	`python -m pytest tests/integration/test_react_error_handling.py::test_error_observation_recovery -v`
Full Loop E2E	Simple task completed with mocked LLM sequence	Task completed within expected number of cycles	`python -m pytest tests/integration/test_react_full_loop.py::test_calculate_percentage_mocked -v`

Concrete Verification Script:

```bash
# Run the ReAct loop integration test suite
pytest tests/integration/test_react_loop.py \
       tests/integration/test_react_with_tools.py \
       tests/unit/test_action_parser.py \
       -v --tb=short

# Expected: All tests pass with clear demonstration of:
# √ Loop stops after 5 iterations (max limit)
# √ Parser extracts JSON from LLM response text
# √ Tool execution results appear in next prompt
# √ Error observations trigger retry or alternative
# √ Mocked sequence completes task (e.g., "Calculate 20% of 50")
```

Manual Verification Steps:

1. Create a test script that initializes a `ReActAgent` with a `CalculatorTool`
2. Set a verbose logger to see think-act-observe cycles
3. Run task: "What is 18% of 250?"

4. Verify output shows:
    - Thought: "I need to calculate 18% of 250"
    - Action: `CalculatorTool` with `{"operation": "percentage", "value": 250, "percentage": 18}`
    - Observation: "Result: 45"
    - Final Answer: "45" or "18% of 250 is 45"
5. Confirm loop terminates after final answer

Milestone 3: Planning & Task Decomposition Verification

Checkpoint Objective: Verify that complex tasks are decomposed into executable subtasks with proper dependencies and execution ordering.

Test Category	Specific Verification	Expected Outcome	Command to Run
Task Decomposition	LLM breaks goal into actionable subtasks	Generated `Task` contains 3-5 subtasks with descriptions	`python -m pytest tests/integration/test_planner.py::test_task_decomposition -v`
DAG Construction	Subtask dependencies form valid DAG (no cycles)	`Plan.execution_order` respects topological ordering	`python -m pytest tests/unit/test_plan_dag.py::test_dag_validation -v`
Parallel Execution	Independent subtasks identified for parallel execution	Multiple subtasks marked as executable simultaneously	`python -m pytest tests/integration/test_parallel_execution.py::test_independent_subtasks -v`
Replanning Trigger	Subtask failure triggers replanning	New plan generated with alternative approach	`python -m pytest tests/integration/test_replanning.py::test_replan_on_failure -v`
End-to-End Planning	Multi-step task completed via planning	All subtasks executed in correct order, final goal achieved	`python -m pytest tests/integration/test_planning_e2e.py::test_research_and_summarize_mocked -v`

Concrete Verification Script:

```bash
# Run the planning system test suite
pytest tests/integration/test_planner.py \
       tests/unit/test_plan_dag.py \
       tests/integration/test_replanning.py \
       -v --tb=short

# Expected: All tests pass, demonstrating:
# ✓ Task decomposed into logical subtasks (mocked LLM)
# ✓ DAG validation rejects circular dependencies
# ✓ Replanning occurs when subtask fails
# ✓ Independent subtasks can execute in parallel (async test)
# ✓ End-to-end planning completes multi-step task
```

Manual Verification Steps:

1. Create a `Planner` instance with access to search, calculator, and file tools
2. Call `create_plan("Research the population of Tokyo and calculate 10% of it")`
3. Inspect the generated `Task`:
    - Should have subtasks like ["Search Tokyo population", "Extract population number", "Calculate 10%"]

- Dependencies: calculation depends on search
- Each subtask has tool assignments

4. Execute the plan and verify:
   - Subtasks execute in dependency order
   - Results flow between subtasks
   - Final answer includes both population and calculation

Milestone 4: Memory & Context Management Verification

Checkpoint Objective: Verify that memory systems store, retrieve, and manage context effectively within token limits.

Test Category	Specific Verification	Expected Outcome	Command to Run
Conversation Memory	Dialogue history maintained in order	`EpisodicMemory.get_messages()` returns complete sequence	`python -m pytest tests/unit/test_episodic_memory.py::test_message_ordering -v`
Working Memory	Task state persists across subtasks	`WorkingMemory.get()` retrieves values set earlier in task	`python -m pytest tests/integration/test_working_memory.py::test_cross_subtask_stat`
Vector Retrieval	Relevant past memories retrieved by similarity	Retrieved entries semantically match query (mocked embeddings)	`python -m pytest tests/integration/test_long_term_memory.py::test_semantic_retriev`
Token Budget Management	Context truncated/summarized to fit limits	Final prompt within token budget, older content summarized	`python -m pytest tests/integration/test_context_management.py::test_token_budget_e -v`
Memory Integration	Retrieved context improves task performance	Agent with memory succeeds where agent without fails	`python -m pytest tests/integration/test_memory_integration.py::test_memory_aids_re -v`

Concrete Verification Script:

```bash
# Run the memory system test suite
pytest tests/unit/test_episodic_memory.py \
       tests/integration/test_long_term_memory.py \
       tests/integration/test_context_management.py \
       -v --tb=short

# Expected: All tests pass, showing:
# √ Conversation history preserved in order
# √ Working memory persists across execution steps
# √ Vector retrieval returns relevant memories (mocked)
# √ Token budget enforced via truncation/summarization
# √ Memory integration improves agent performance
```

Manual Verification Steps:

1. Initialize agent with all three memory layers
2. Perform a multi-turn conversation:
   - Q1: "What is the capital of France?"
   - A1: "Paris"
   - Q2: "What is its population?"
3. Verify:
   - Conversation memory contains both Q/A pairs
   - Working memory may store "capital=Paris" for context
   - If long-term memory has prior France facts, they're retrieved
   - Total context length fits within LLM limits (check token count)

4. Test summarization by exceeding token limit and verifying older turns are summarized

Milestone 5: Multi-Agent Collaboration Verification

Checkpoint Objective: Verify that multiple agents can collaborate through role-based delegation, message passing, and result aggregation.

Test Category	Specific Verification	Expected Outcome	Command to Run
Role Assignment	Agents advertise capabilities for task routing	Orchestrator matches subtasks to agents by role	`python -m pytest tests/unit/test_role_assignment.py::test_role_matching -v`
Message Passing	Messages delivered between agents via bus	`MessageBus.publish()` delivers to subscribed agents	`python -m pytest tests/integration/test_message_bus.py::test_message_delivery -v`
Task Delegation	Orchestrator delegates subtasks to appropriate workers	Worker agents receive tasks matching their roles	`python -m pytest tests/integration/test_delegation.py::test_task_delegation_flow -v`
Conflict Resolution	Contradictory agent outputs resolved by strategy	Conflicting results merged using defined merge logic	`python -m pytest tests/integration/test_conflict_resolution.py::test_result_merging -v`
Full Collaboration E2E	Multi-agent team completes complex task	All agents contribute, orchestrator synthesizes final answer	`python -m pytest tests/integration/test_multi_agent_e2e.py::test_research_and_write_collaboration -v`

Concrete Verification Script:

```bash
# Run the multi-agent collaboration test suite
pytest tests/integration/test_message_bus.py \
       tests/integration/test_delegation.py \
       tests/integration/test_multi_agent_e2e.py \
       -v --tb=short

# Expected: All tests pass, demonstrating:
# ✓ Message bus delivers messages to correct subscribers
# ✓ Orchestrator delegates to role-appropriate workers
# ✓ Conflict resolution merges contradictory results
# ✓ Full multi-agent team completes task (mocked LLMs)
# ✓ No delegation loops or deadlocks
```

Manual Verification Steps:

1. Create an orchestrator agent and 2-3 worker agents (Researcher, Writer, Calculator)
2. Start message bus and subscribe all agents
3. Submit task: "Research the average rainfall in Seattle and write a one-paragraph summary"
4. Verify:
   - Orchestrator decomposes task and delegates research to Researcher
   - Researcher sends findings to Writer via message bus

- Writer produces summary, possibly asking Calculator for conversions
- Orchestrator receives all outputs and synthesizes final answer

5. Check that all messages are properly routed and no agent is idle when work exists

> **Critical Success Factor**: The verification checkpoints are designed to be **objective and automatable**. Each test produces a clear pass/fail result, avoiding subjective assessments like "the agent seems to reason well." This rigor ensures the framework meets its functional requirements before moving to the next milestone.

Implementation Guidance

Testing autonomous AI systems requires careful architecture to balance realism with determinism. The key is to create **testable seams**—interfaces where we can substitute real components with mocks during testing.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Testing Framework	`pytest` with `asyncio` plugin	`pytest` + `pytest-asyncio` + `pytest-mock`
Mock LLM	In-memory mock returning predefined responses	`unittest.mock.AsyncMock` with response sequences
Mock Vector Store	In-memory similarity with simple cosine distance	`chromadb` in-memory ephemeral client
Test Coverage	`pytest-cov` for line coverage reports	`pytest-cov` + `codecov.io` integration
E2E LLM	`transformers` pipeline with small local model	OpenAI `gpt-3.5-turbo` with test API key
Concurrency Testing	`asyncio` event loop in tests	`pytest-asyncio` with timeout controls

B. Recommended File/Module Structure

```
project-archon/
├── tests/
│   ├── unit/                        # Unit tests (fast, deterministic)
│   │   ├── test_tool_registry.py
│   │   ├── test_action_parser.py
│   │   ├── test_data_structures.py    # ToolResult, SubTask, Message, etc.
│   │   ├── test_memory_components.py  # EpisodicMemory, WorkingMemory
│   │   └── test_utilities.py          # extract_json_from_text, etc.
│   │
│   ├── integration/                   # Integration tests (component interactions)
│   │   ├── test_tool_safety.py        # Timeout, validation, error handling
│   │   ├── test_react_loop.py         # ReAct with mocked LLM
│   │   ├── test_planner.py            # Task decomposition
│   │   ├── test_memory_integration.py # Memory + agent interaction
│   │   ├── test_multi_agent.py        # Message passing, delegation
│   │   └── test_error_recovery.py     # RecoveryOrchestrator integration
│   │
│   ├── e2e/                          # End-to-end tests (real LLM, infrequent)
│   │   ├── test_simple_tasks.py       # Calculator, search, etc.
│   │   ├── test_planning_tasks.py     # Multi-step tasks
│   │   └── conftest.py               # E2E fixtures (LLM setup, cleanup)
│   │
│   ├── fixtures/                      # Shared test fixtures
│   │   ├── mock_llm.py               # Mock LLM responses
│   │   ├── sample_tools.py           # Test tools for integration tests
│   │   └── mock_embeddings.py        # Fake embedding generator
│   │
│   ├── conftest.py                    # Global pytest fixtures
│   └── pytest.ini                     # Pytest configuration
│
├── src/                               # Main source code
│   └── archon/                       # Framework code
└── pyproject.toml                     # Project dependencies
```

C. Infrastructure Starter Code

Complete Mock LLM for Integration Testing:

PYTHON

```python
# tests/fixtures/mock_llm.py

from typing import List, Dict, Any, Optional

from unittest.mock import AsyncMock, MagicMock

import json


class MockLLM:

    """Deterministic mock LLM for testing agent loops."""


    def __init__(self, response_sequence: List[str]):

        """

        Args:

            response_sequence: List of responses to return in order.

                Each response should be a string that the agent would

                receive from a real LLM.

        """

        self.response_sequence = response_sequence

        self.call_count = 0

        self.history = []


    async def generate(self, prompt: str, **kwargs) -> str:

        """Mock LLM generation that returns predetermined responses."""

        self.history.append({

            "prompt": prompt[:200] + "..." if len(prompt) > 200 else prompt,

            "call_number": self.call_count

        })


        if self.call_count >= len(self.response_sequence):

            # Default fallback response if sequence exhausted

            response = '{"action": "final", "answer": "Test completed"}'

        else:

            response = self.response_sequence[self.call_count]


        self.call_count += 1

        return response


    def reset(self):

        """Reset call counter for test reuse."""

        self.call_count = 0

        self.history.clear()


def create_mocked_llm_response_chain(*steps: Dict[str, Any]) -> List[str]:
```

```
    """
    Helper to create a sequence of LLM responses for testing ReAct loops.

    Example:
        create_mocked_llm_response_chain(
            {"thought": "I need to calculate", "action": "Calculator", "args": {"operation": "percentage", ...}},
            {"thought": "Got result", "action": "final", "answer": "The result is 30"}
        )
    """
    responses = []
    for step in steps:
        if "action" in step and step["action"] == "final":
            responses.append(json.dumps({
                "thought": step.get("thought", ""),
                "action": "final",
                "answer": step["answer"]
            }))
        else:
            responses.append(json.dumps({
                "thought": step.get("thought", ""),
                "action": step["action"],
                "action_input": step.get("args", {})
            }))
    return responses
```

Simple In-Memory Vector Store for Testing:

PYTHON

```python
# tests/fixtures/mock_embeddings.py

import numpy as np

from typing import List, Dict, Any, Optional, Tuple

from collections import defaultdict


class InMemoryVectorStore:

    """Simple vector store for testing memory retrieval."""


    def __init__(self, embedding_dim: int = 384):

        self.embedding_dim = embedding_dim

        self.memories: Dict[str, Dict[str, Any]] = {}

        self.embeddings: Dict[str, np.ndarray] = {}


    def store(self, id: str, content: str, metadata: Dict[str, Any],

              embedding: Optional[np.ndarray] = None) -> None:

        """Store memory with optional embedding."""

        self.memories[id] = {

            "id": id,

            "content": content,

            "metadata": metadata,

            "timestamp": metadata.get("timestamp")

        }


        if embedding is not None:

            self.embeddings[id] = embedding

        else:

            # Generate simple deterministic embedding for testing

            self.embeddings[id] = self._simple_hash_embedding(content)


    def query(self, query_embedding: np.ndarray, limit: int = 5,

              threshold: float = 0.0) -> List[Dict[str, Any]]:

        """Return memories sorted by cosine similarity to query."""

        results = []

        for mem_id, emb in self.embeddings.items():

            similarity = self._cosine_similarity(query_embedding, emb)

            if similarity >= threshold:

                memory = self.memories[mem_id].copy()

                memory["similarity"] = similarity

                results.append(memory)


        # Sort by similarity descending
```

```
        results.sort(key=lambda x: x["similarity"], reverse=True)
        return results[:limit]


    def _simple_hash_embedding(self, text: str) -> np.ndarray:
        """Create deterministic embedding from text hash for testing."""
        import hashlib
        hash_val = int(hashlib.md5(text.encode()).hexdigest()[:8], 16)
        np.random.seed(hash_val)
        return np.random.randn(self.embedding_dim)


    def _cosine_similarity(self, a: np.ndarray, b: np.ndarray) -> float:
        """Compute cosine similarity between two vectors."""
        norm_a = np.linalg.norm(a)
        norm_b = np.linalg.norm(b)
        if norm_a == 0 or norm_b == 0:
            return 0.0
        return np.dot(a, b) / (norm_a * norm_b)
```

D. Core Logic Skeleton Code

Unit Test for Tool Parameter Validation:

```python
# tests/unit/test_tool_execution.py

import pytest

from archon.tools import ToolRegistry, ToolExecutionEngine

from archon.tools.base import ToolResult, ToolResultStatus

from archon.tools.calculator import CalculatorTool


class TestToolExecution:

    """Unit tests for tool execution and validation."""



    def test_validation_rejects_malformed_input(self):

        """Verify JSON schema validation catches type mismatches."""

        # TODO 1: Create a CalculatorTool and register it

        # TODO 2: Create ToolExecutionEngine with the registry

        # TODO 3: Attempt to execute with invalid arguments (string instead of number)

        # TODO 4: Verify ToolResult has status ERROR (not SUCCESS)

        # TODO 5: Verify error message mentions type mismatch

        pass



    def test_tool_timeout_enforced(self, tool_registry):

        """Integration test for timeout protection."""

        # TODO 1: Create a tool that sleeps longer than timeout

        # TODO 2: Register tool with 0.5 second timeout

        # TODO 3: Execute tool via ToolExecutionEngine

        # TODO 4: Verify result status is TIMEOUT (not SUCCESS or ERROR)

        # TODO 5: Verify execution took approximately timeout duration

        pass
```

Integration Test for ReAct Loop:

```python
# tests/integration/test_react_loop.py

import pytest

import asyncio

from archon.agents.react import ReActAgent

from tests.fixtures.mock_llm import MockLLM, create_mocked_llm_response_chain


class TestReActLoop:

    """Integration tests for the ReAct loop engine."""



    @pytest.mark.asyncio

    async def test_iteration_limit_enforced(self):

        """Verify loop stops after max iterations."""

        # TODO 1: Create MockLLM that always returns "I need to think more" (no final answer)

        # TODO 2: Initialize ReActAgent with max_iterations=3

        # TODO 3: Execute a simple task

        # TODO 4: Verify loop stopped after 3 iterations

        # TODO 5: Verify final output indicates iteration limit reached

        pass



    @pytest.mark.asyncio

    async def test_tool_execution_in_loop(self):

        """Verify tool execution integrates with ReAct cycle."""

        # TODO 1: Create MockLLM with response chain that includes tool call

        # TODO 2: Register a CalculatorTool

        # TODO 3: Initialize ReActAgent and run task

        # TODO 4: Verify tool was called with correct arguments

        # TODO 5: Verify tool result appeared in next cycle's observation

        # TODO 6: Verify final answer incorporates tool result

        pass
```

E. Language-Specific Hints

Python Testing Best Practices:

1. **Async Testing**: Use `@pytest.mark.asyncio` decorator for async tests. Ensure event loop is properly managed.
2. **Mocking**: `unittest.mock.AsyncMock` for mocking async methods, `unittest.mock.MagicMock` for sync methods.
3. **Deterministic Tests**: Seed random number generators (`np.random.seed(42)`) in tests involving randomness.
4. **Timeout Controls**: Use `asyncio.wait_for()` with short timeouts in tests to catch hangs.
5. **Fixture Management**: Use `@pytest.fixture` with appropriate scopes (function, class, session) to share test setup.

Testing Asynchronous Code:

```python
# Example pattern for testing async components
@pytest.mark.asyncio
async def test_async_message_bus():
    bus = MessageBus()
    received = []

    async def callback(msg):
        received.append(msg)

    bus.subscribe("agent1", callback, ["topic1"])
    await bus.publish(Message(sender="test", receiver="*", message_type="task", content={}))

    await asyncio.sleep(0.01)  # Small delay for async delivery
    assert len(received) == 1
```

F. Milestone Checkpoint Commands

Complete Test Suite Execution Commands:

```bash
# Milestone 1: Tool System Complete
pytest tests/unit/test_tool_registry.py tests/unit/test_tool_execution.py \
       tests/integration/test_tool_safety.py -xvs

# Expected: 15-20 passing tests showing:
# - Tools can be registered and retrieved
# - Parameter validation works with JSON schema
# - Timeouts enforced (tests may take a few seconds)
# - Error handling converts exceptions to ToolResult

# Milestone 2: ReAct Loop Complete
pytest tests/unit/test_action_parser.py tests/integration/test_react_loop.py \
       tests/integration/test_react_with_tools.py -xvs

# Expected: 10-15 passing tests showing:
# - Parser extracts JSON from LLM text
# - Loop terminates at iteration limit
# - Tool execution integrated into cycle
# - Error observations trigger recovery

# Milestone 3: Planning Complete
pytest tests/integration/test_planner.py tests/unit/test_plan_dag.py \
       tests/integration/test_replanning.py -xvs

# Expected: 10-12 passing tests showing:
# - Task decomposition produces subtasks
# - DAG validation prevents cycles
# - Replanning occurs on failure
# - Independent subtasks can run in parallel (async)

# Milestone 4: Memory Complete
pytest tests/unit/test_episodic_memory.py tests/integration/test_long_term_memory.py \
       tests/integration/test_context_management.py -xvs

# Expected: 12-15 passing tests showing:
# - Conversation history preserved
# - Vector retrieval returns relevant results
# - Token budget enforced via summarization
# - Memory integration aids task completion

# Milestone 5: Multi-Agent Complete
pytest tests/integration/test_message_bus.py tests/integration/test_delegation.py \
       tests/integration/test_multi_agent_e2e.py -xvs

# Expected: 10-12 passing tests showing:
```

```
# - Message bus delivers to subscribers

# - Orchestrator delegates by role

# - Conflict resolution merges results

# - Multi-agent team completes task
```

G. Debugging Tips for Test Failures

Symptom	Likely Cause	How to Diagnose	Fix
Tool tests hang forever	Tool execution not timing out	Check `ToolExecutionEngine._timeout_context()` implementation	Ensure timeout uses `asyncio.wait_for` or `signal.alarm`
ReAct loop gets stuck	LLM mock not returning final answer	Inspect mock response sequence; check parser output	Ensure mock includes `{"action": "final", ...}` response
Memory retrieval returns nothing	Embedding mismatch or empty store	Log query embedding and stored embeddings	Check embedding generation; verify store has content
Async tests fail with event loop errors	Event loop already running or closed	Check pytest-asyncio version and fixture usage	Use `@pytest.mark.asyncio` decorator; avoid nested event loops
Tests pass locally but fail in CI	Timing differences or race conditions	Add logging to see execution order; increase timeouts	Use deterministic mocks; add small delays in async tests
E2E tests flaky	LLM variance or network issues	Capture LLM responses; check for consistent patterns	Use more specific prompts; implement retry with backoff
Coverage reports show untested error paths	Missing negative test cases	Review coverage report for uncovered lines	Add tests for validation failures, timeout, circuit breaker open

> **Final Verification**: After completing all five milestones, run the comprehensive test suite with `pytest tests/ -v --tb=short`. Expect 80+ passing tests with 85%+ code coverage. The system should demonstrate all core capabilities: tool execution, ReAct reasoning, task planning, memory management, and multi-agent collaboration through deterministic tests, with optional E2E tests showing real task completion.

> **Milestone(s):** 1, 2, 3, 4, 5

9. Debugging Guide

Autonomous AI agents represent a complex, non-deterministic system where traditional debugging approaches often fall short. When an agent fails—getting stuck in a loop, producing incorrect results, or crashing unexpectedly—the root cause can be buried in multiple layers: the LLM's reasoning, the tool execution, the planning logic, or the memory system. This section provides a systematic approach to diagnosing and fixing common issues that arise during development of the framework.

Common Bug Symptoms and Fixes

The following table catalogs the most frequent failure modes observed when building AI agent systems, organized by their primary symptoms. Each entry includes the observable behavior, the underlying technical causes, and concrete step-by-step fixes to implement.

Symptom	Likely Cause(s)	Step-by-Step Fix
Agent gets stuck in an infinite loop, repeating the same tool call	1. **Missing termination condition**: The LLM isn't being prompted with a clear "final answer" format or the parser isn't detecting it. 2. **Tool result not informative**: The observation returned to the LLM doesn't provide enough context to progress. 3. **Maximum iteration limit not enforced**: The `ReActLoopContext.iteration` counter isn't being checked or `max_iterations` is set too high.	1. **Inspect the prompt template**: Ensure your `_format_react_prompt` includes explicit instructions like "If you have the final answer, respond with `Final Answer:` followed by the answer." 2. **Check the action parser**: Verify `parse_llm_action_response` correctly identifies both `Action:` and `Final Answer:` patterns. 3. **Add loop detection**: Track previous actions in `ReActLoopContext.previous_actions` and detect when the same tool with same arguments is called repeatedly; break with an error observation. 4. **Set conservative limits**: Start with `max_iterations=10` during development and incrementally increase only after verifying loop stability.
LLM hallucinates non-existent tools	1. **Tool descriptions too vague**: The LLM cannot distinguish between similar-sounding tools. 2. **Tool list not updated in prompt**: Available tools aren't being dynamically injected into each cycle's prompt. 3. **No validation before execution**: The system attempts to execute a hallucinated tool name instead of rejecting it.	1. **Improve tool descriptions**: Rewrite tool `description` fields to be distinct and action-oriented (e.g., "Search the web for current information using DuckDuckGo" vs. "Get data from the internet"). 2. **Implement tool validation**: In `ReActAgent._run_single_cycle`, after parsing the action, call `ToolRegistry.get_tool(tool_name)` and if `None`, return an error observation: "Tool {tool_name} does not exist. Available tools: ..." 3. **Include tool signatures in prompt**: Format available tools with their JSON schema parameters so the LLM understands required arguments.
Action parsing fails silently	1. **LLM output format mismatch**: The LLM is producing free-form text instead of the expected `Action: {"tool": "...", "args": {...}}` format. 2. **JSON parsing too strict**: The `extract_json_from_text` function fails on valid but slightly malformed JSON (trailing commas, single quotes). 3. **Regex pattern incomplete**: The regex in `parse_llm_action_response` doesn't capture all valid variations of the action format.	1. **Add robust JSON parsing**: Implement a fallback in `extract_json_from_text` that attempts to fix common JSON issues (replace single quotes with double quotes, add missing closing braces). 2. **Log the raw LLM response**: Always log the complete LLM output before parsing to see what format is actually being produced. 3. **Use function calling if available**: If your LLM provider supports structured function calling (OpenAI, Anthropic), switch to that API instead of text parsing. 4. **Improve prompt engineering**: Add explicit examples of correct action format in the few-shot examples within your prompt template.
Tool execution hangs forever	1. **Missing timeout**: Tool execution is not wrapped in a timeout context manager. 2. **Network dependency unresponsive**: External API calls (web search, database) may hang indefinitely. 3. **Synchronous blocking in async context**: A synchronous tool is blocking the entire event loop.	1. **Implement mandatory timeouts**: In `ToolExecutionEngine.execute_tool_call`, wrap the execution in `_timeout_context(timeout_seconds)` with a sensible default (e.g., 30 seconds). 2. **Use async for I/O-bound tools**: Implement tools with async `execute` methods and use `asyncio.wait_for` for timeout control. 3. **Add circuit breakers**: Use `ToolExecutionEngine._update_circuit_breaker` to temporarily disable tools that repeatedly timeout or fail.
Subtask dependencies cause deadlock	1. **Circular dependency in DAG**: Subtask A depends on B, and B depends on A, either directly or transitively. 2. **Dependency not marked complete**: A subtask's `status` remains `EXECUTING` after completion due to missing `SubTask.with_status(COMPLETED)` call. 3. **Missing dependency tracking**: The `Plan.execution_order` doesn't respect all dependencies during topological sort.	1. **Validate DAG acyclicity**: In `Planner.create_plan`, after building the dependency graph, run a cycle detection algorithm (DFS with three-color marking) and reject circular plans. 2. **Implement proper status transitions**: Ensure every subtask moves from `EXECUTING` to `COMPLETED` (or `FAILED`) via immutable updates: `task.update_subtask(subtask.with_status(COMPLETED).with_result(result))`. 3. **Use topological sort with dependency checking**: In `ExecutionFlow.get_next_ready_subtask`, verify `subtask.is_ready(completed_task_ids)` checks all dependencies, not just immediate ones.
Memory retrieval returns irrelevant context	1. **Poor embedding quality**: Using a generic sentence transformer not tuned for the domain. 2. **No query expansion**: The raw user query is used for vector search without augmentation. 3. **Missing metadata filtering**: Retrieved memories aren't filtered by recency or task relevance.	1. **Improve query formulation**: In `MemoryRetriever.retrieve_context`, expand the query with context from `WorkingMemory` (current goal, subtask description) before embedding. 2. **Add hybrid search**: Combine vector similarity with keyword matching (BM25) and recency boosting. 3. **Implement relevance threshold**: In `LongTermMemory.query`, filter out results with similarity score below a configurable threshold (e.g., 0.7). 4. **Add metadata filters**: Store task ID, agent role, or tool names in `MemoryEntry.metadata` and filter retrievals by current context.
Multi-agent delegation	1. **Delegation loop**: Agent A delegates to B, who delegates back to A for the same subtask. 2. **No timeout on delegated tasks**: A worker	1. **Track delegation chain**: In `OrchestratorAgent._delegate_subtask`, store the chain of delegations in `DelegatedTask.metadata` and reject if the same agent appears twice. 2. **Set deadlines on delegated tasks**: Each `DelegatedTask` should have a `deadline`

Symptom	Likely Cause(s)	Step-by-Step Fix
creates infinite loops	agent never responds, leaving the orchestrator waiting indefinitely. 3. **Missing conflict resolution**: Contradictory results from agents cause the orchestrator to repeatedly request re-execution.	timestamp; implement a background cleanup task that marks overdue tasks as failed. 3. **Implement voting or authority**: In `OrchestratorAgent._resolve_conflicts`, use a simple voting scheme (majority wins) or designate an expert agent as tie-breaker.
Context window exceeds token limit	1. **Unbounded conversation history**: `EpisodicMemory.append` is called for every turn without summarization or eviction. 2. **Too many retrieved memories**: `MemoryRetriever.retrieve_context` fetches more context than fits in the token budget. 3. **Inefficient prompt formatting**: The prompt template includes verbose tool descriptions or examples on every cycle.	1. **Implement token counting**: In `EpisodicMemory.get_recent`, count tokens (approximately 4 chars per token) and truncate when exceeding `max_tokens`. 2. **Dynamic context budgeting**: Allocate token budgets: 50% for conversation history, 30% for retrieved memories, 20% for prompt template. 3. **Summarize old turns**: When history grows beyond threshold, call `LongTermMemory.summarize_recent` to compress older messages into a single summary entry. 4. **Streamline prompt**: Remove redundant instructions after the first cycle; use placeholders that only expand when needed.
Tool returns success but incorrect data	1. **Tool schema mismatch**: The tool's `parameters_schema` doesn't validate all required constraints (e.g., numeric ranges, string patterns). 2. **LLM misinterpretation**: The LLM passes arguments in wrong format (string vs number) that passes validation but produces wrong results. 3. **External API changed**: The underlying service API updated but the tool wrapper wasn't updated.	1. **Strengthen schema validation**: Use JSON Schema `minimum`, `maximum`, `pattern`, `enum` constraints to catch invalid arguments before execution. 2. **Add example validation**: Include example valid/invalid parameter sets in tool documentation for LLM few-shot learning. 3. **Implement result validation**: After tool execution, validate the `ToolResult.content` against an expected output schema when possible. 4. **Monitor tool health**: Track tool success/failure rates in `ToolExecutionEngine` and alert when error rates spike.
Agent state becomes inconsistent after error	1. **Partial state updates**: An error occurs mid-update, leaving some components updated and others not. 2. **Missing error recovery rollback**: The `RecoveryOrchestrator` doesn't revert partial changes when recovery fails. 3. **Race conditions in async code**: Concurrent modifications to `ExecutionFlow.active_flows` without proper locking.	1. **Use immutable updates**: Follow the **functional update** pattern: `flow = flow.mark_subtask_complete(subtask_id)` returns a new instance rather than mutating. 2. **Implement transaction-like boundaries**: Group related state changes into a single update operation protected by a lock. 3. **Add state validation checks**: Periodically validate invariants (e.g., a `COMPLETED` subtask must have a `result`) and trigger recovery if violated. 4. **Use `asyncio.Lock` for shared mutable state**: In `MessageBus` and `FlowController`, protect critical sections with locks.

Debugging Techniques and Tools

Debugging autonomous agents requires moving beyond traditional print statements to systematic inspection of the agent's internal reasoning process. The key insight is that the agent's "thought process" is partially observable through the sequence of thoughts, actions, and observations it generates. Below are proven techniques for illuminating the black box.

Structured Logging of the Thought-Action Chain

The most powerful debugging tool is comprehensive, structured logging of every ReAct cycle. Instead of logging raw text, create a machine-readable log that captures the complete state at each step:

> **Design Principle**: Log with enough context to replay any agent session offline for analysis. Every log entry should include the full `ReActLoopContext` and the raw LLM response.

Implementation Approach:

1. **Create a logging decorator** that wraps `ReActAgent._run_single_cycle` and logs:
   - Cycle number (`context.iteration`)
   - The formatted prompt sent to the LLM (truncated)
   - The raw LLM response
   - The parsed action (or final answer)
   - The tool execution result (status, content snippet)
   - The updated `AgentState`

- Timestamps for performance analysis

2. **Use structured JSON logging** so logs can be queried and analyzed programmatically:

```python
# Example log entry structure (conceptual, not code in main body)
{
  "timestamp": "2023-10-05T14:30:00Z",
  "component": "ReActAgent",
  "cycle": 3,
  "state": "ACTING",
  "action": {"tool": "web_search", "args": {"query": "current weather NYC"}},
  "observation_preview": "Weather report: 72°F, sunny...",
  "context_snapshot": {
     "subtask_description": "Find current weather in New York",
     "completed_actions": ["search_weather_api"]
  }
}
```

3. **Add correlation IDs**: Pass a unique `correlation_id` through the entire execution flow (from `Agent.run_task` down to individual tool calls) to trace requests across components.

Inspecting the Prompt Sent to the LLM

Often, the root cause of agent misbehavior lies in the prompt construction. The LLM can only work with what you give it, so inspecting the exact prompt (with all context injections) is crucial.

Debugging Procedure:

1. **Capture the complete prompt** by adding a debug flag to `ReActAgent._format_react_prompt` that writes the full prompt to a file when enabled.
2. **Analyze prompt components**:
   - **System instructions**: Are they clear about the agent's role and constraints?
   - **Available tools list**: Are descriptions distinct? Are parameter schemas included?
   - **Conversation history**: Is it truncated correctly? Are important earlier turns preserved?
   - **Working memory context**: Is current task state properly injected?
   - **Retrieved memories**: Are they actually relevant to the current subtask?
   - **Few-shot examples**: Do they demonstrate correct reasoning and action format?
3. **Test the prompt manually** in an LLM playground (OpenAI Playground, Anthropic Console) to see how the raw LLM responds without your parsing logic.

Using an LLM Playground to Test Prompts

When debugging prompt-related issues, interact directly with the LLM using the same model and parameters you're using in production.

Step-by-Step Workflow:

1. **Extract the problematic prompt** from your logs when the agent behaves incorrectly.
2. **Paste it into the playground** with identical parameters (temperature, max tokens, etc.).
3. **Run multiple generations** (5-10) to see the distribution of responses:
   - Does the LLM consistently produce the wrong format?
   - Does it hallucinate tools even when they're clearly listed?
   - Does it fail to recognize when it has the final answer?
4. **Iteratively refine the prompt** based on these observations:
   - Add more explicit instructions
   - Improve few-shot examples
   - Reorder sections (tools list before history, etc.)
   - Adjust temperature (lower for more deterministic behavior)

Visualizing the Task Execution DAG

For planning-related bugs, a visual representation of the task decomposition and execution progress is invaluable.

Implementation Approach:

1. **Export the Plan structure** to Graphviz DOT format or Mermaid syntax:

```python
# Helper function to visualize plan

def plan_to_mermaid(plan: Plan) -> str:

    mermaid_lines = ["graph TD"]

    for subtask_id, deps in plan.subtask_dag.items():

        for dep in deps:

            mermaid_lines.append(f"    {dep} --> {subtask_id}")

    return "\n".join(mermaid_lines)
```

2. **Include status annotations** in the visualization by coloring nodes based on `SubTask.status` (green for COMPLETED, yellow for EXECUTING, red for FAILED).
3. **Use this visualization** to identify:
   - Circular dependencies (cycles in the graph)
   - Long critical paths
   - Bottlenecks where many tasks wait on a single dependency
   - Subtasks stuck in EXECUTING state

Interactive Debugging with the Python Debugger (PDB)

For complex logic errors in your Python code, traditional debugging tools still apply, but with some adaptations for async code.

Async-Aware Debugging Techniques:

1. **Use `asyncio.run()` wrappers** for testing individual components in isolation:

```python
# Test a tool execution in isolation

result = asyncio.run(tool_registry.get_tool("calculator").execute({"operation": "add", "a": 2, "b": 3}))
```

2. **Set breakpoints in async functions** with `import pdb; pdb.set_trace()` but ensure you're in an async context:
   - Use `await`-compatible debugging: `import aiodebug; aiodebug.log_slow_callbacks.enable()`
   - Or use the built-in `asyncio` debug mode: `PYTHONASYNCIODEBUG=1`
3. **Debug event loop issues** by inspecting task states:

```python
# List all running tasks

tasks = asyncio.all_tasks()

for task in tasks:

    print(task.get_name(), task.get_coro(), task.done())
```

Mock LLM for Deterministic Testing

When debugging agent logic (not prompt issues), replace the real LLM with a `MockLLM` that returns predetermined responses. This isolates your code from LLM variability.

Debugging Workflow:

1. **Create a sequence of mock responses** that replicates the problematic scenario:

```python
mock_responses = create_mocked_llm_response_chain(

    "Thought: I need to search for the weather. Action: {'tool': 'web_search', 'args': {'query': 'weather NYC'}}",

    "Thought: Now I have the weather data. Final Answer: The weather is 72°F and sunny."

)
```

2. **Swap the real LLM** with `MockLLM(response_sequence=mock_responses)` in your test.
3. **Run the exact same agent code** and verify:
   - Does the agent follow the expected path?

- Where does it diverge from the mock script?
- Which component (parser, state machine, etc.) is causing the divergence?

Monitoring and Metrics Dashboard

For production debugging, implement a real-time dashboard showing key agent metrics:

Essential Metrics to Track:

- **ReAct cycle duration** (LLM call time + tool execution time)
- **Tool success/failure rates** per tool name
- **Context window utilization** (tokens used vs. limit)
- **Memory retrieval hit rate** (percentage of queries finding relevant memories)
- **Plan execution progress** (subtasks completed/total over time)

Implementation Approach:

1. **Instrument key methods** with timing decorators that push metrics to a time-series database (Prometheus, InfluxDB).
2. **Create Grafana dashboards** with panels for each metric, allowing you to correlate agent failures with resource usage patterns.
3. **Set up alerts** for anomalous patterns (e.g., tool failure rate > 20%, average cycle time > 30 seconds).

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Logging	Python `logging` module with JSON formatter	Structured logging with `structlog` + log aggregation (Loki, Elasticsearch)
Prompt Inspection	Write prompts to debug files with timestamp prefixes	Interactive web UI that streams prompts/responses in real-time
Visualization	Generate Mermaid diagrams in Markdown files	Real-time DAG visualization with NetworkX + Plotly/Dash
Metrics	Print metrics to stdout/CSV file	OpenTelemetry instrumentation + Prometheus + Grafana dashboard
Mock Testing	Custom `MockLLM` class with response list	Hypothesis testing with generative prompt/response pairs

B. Recommended File/Module Structure

```
project_archon/
├── agents/
│   ├── react_agent.py          # ReActAgent with logging decorator
│   └── orchestrator_agent.py   # OrchestratorAgent with delegation tracking
├── core/
│   ├── logging/                # Structured logging setup
│   │   ├── __init__.py
│   │   ├── structured_logger.py
│   │   └── formatters.py
│   ├── metrics/                # Metrics collection
│   │   ├── __init__.py
│   │   ├── collectors.py
│   │   └── exporters.py
│   └── debugging/              # Debugging utilities
│       ├── __init__.py
│       ├── prompt_inspector.py
│       ├── dag_visualizer.py
│       └── replay_debugger.py
├── tools/
│   └── execution_engine.py     # ToolExecutionEngine with circuit breakers
├── planning/
│   └── planner.py              # Planner with cycle detection
├── memory/
│   └── retriever.py            # MemoryRetriever with hybrid search
└── tests/
    ├── debug_helpers/          # Debugging test utilities
    │   ├── mock_llm.py         # MockLLM implementation
    │   └── test_replayer.py    # Replay logged sessions
    └── integration/
        └── test_agent_loops.py # Tests with deterministic mock LLM
```

C. Infrastructure Starter Code

Complete Structured Logger Implementation:

PYTHON

```python
# core/logging/structured_logger.py

import json

import logging

import time

from datetime import datetime

from typing import Any, Dict, Optional

from uuid import uuid4


class StructuredLogger:

    """JSON-structured logger for agent debugging."""



    def __init__(self, name: str, level=logging.INFO):

        self.logger = logging.getLogger(name)

        self.logger.setLevel(level)

        self.correlation_id = str(uuid4())



        # Console handler with JSON formatter

        handler = logging.StreamHandler()

        formatter = JSONFormatter()

        handler.setFormatter(formatter)

        self.logger.addHandler(handler)



    def log_cycle(self,

                  cycle: int,

                  state: str,

                  prompt_preview: str,

                  llm_response: str,

                  parsed_action: Optional[Dict],

                  tool_result: Optional[Dict],

                  context: Dict[str, Any]):

        """Log a complete ReAct cycle."""

        log_entry = {

            "timestamp": datetime.utcnow().isoformat(),

            "correlation_id": self.correlation_id,

            "component": "ReActAgent",

            "cycle": cycle,

            "state": state,

            "prompt_preview": prompt_preview[:200] + "..." if len(prompt_preview) > 200 else prompt_preview,

            "llm_response": llm_response,

            "parsed_action": parsed_action,

            "tool_result": tool_result,
```

```
            "context": context,
            "performance": {
                "timestamp": time.time()
            }
        }
        self.logger.info(log_entry)

    def log_error(self,
                  error_type: str,
                  message: str,
                  component: str,
                  details: Dict[str, Any]):
        """Log an error with structured context."""
        log_entry = {
            "timestamp": datetime.utcnow().isoformat(),
            "correlation_id": self.correlation_id,
            "level": "ERROR",
            "component": component,
            "error_type": error_type,
            "message": message,
            "details": details
        }
        self.logger.error(log_entry)

class JSONFormatter(logging.Formatter):
    """Format log records as JSON."""

    def format(self, record: logging.LogRecord) -> str:
        if isinstance(record.msg, dict):
            log_dict = record.msg
        else:
            log_dict = {"message": record.msg}

        log_dict.update({
            "level": record.levelname,
            "timestamp": datetime.utcnow().isoformat(),
            "module": record.module,
            "function": record.funcName,
            "line": record.lineno
        })
```

```
        return json.dumps(log_dict, default=str)
```

D. Core Logic Skeleton Code

Debugging Decorator for ReAct Cycle:

PYTHON

```python
# agents/react_agent.py (additional debugging code)

from functools import wraps

from typing import Callable, Any

from core.logging.structured_logger import StructuredLogger


def log_react_cycle(func: Callable) -> Callable:

    """Decorator to log each ReAct cycle with full context."""

    @wraps(func)

    async def wrapper(self, context: ReActLoopContext,

                      available_tools: List[Tool],

                      external_context: str) -> Tuple[AgentState, ReActLoopContext]:



        # Initialize logger if not present

        if not hasattr(self, '_debug_logger'):

            self._debug_logger = StructuredLogger(f"ReActAgent.{id(self)}")



        # Log before execution

        prompt = self._format_react_prompt(context, available_tools, external_context)

        self._debug_logger.log_cycle(

            cycle=context.iteration,

            state=context.state.value,

            prompt_preview=prompt[:500],

            llm_response="",  # Will be filled after LLM call

            parsed_action=None,

            tool_result=None,

            context={

                "subtask_description": context.subtask_description,

                "max_iterations": context.max_iterations,

                "previous_actions_count": len(context.previous_actions)

            }

        )



        try:

            # Execute the actual cycle

            start_time = time.time()

            new_state, new_context = await func(self, context, available_tools, external_context)

            elapsed = time.time() - start_time



            # TODO 1: Capture the LLM response (you'll need to modify _run_single_cycle to return it)

            # TODO 2: Extract the parsed action from the cycle execution

            # TODO 3: Capture the tool execution result if an action was taken
```

```
            # TODO 4: Log the complete cycle with all captured data


            # Example structure:
            # self._debug_logger.log_cycle_complete(
            #     cycle=context.iteration,
            #     llm_response=llm_response_text,
            #     parsed_action=parsed_action_dict,
            #     tool_result=tool_result_dict,
            #     elapsed_time=elapsed
            # )


            return new_state, new_context


        except Exception as e:
            self._debug_logger.log_error(
                error_type=type(e).__name__,
                message=str(e),
                component="ReActAgent._run_single_cycle",
                details={
                    "cycle": context.iteration,
                    "context_state": context.state.value
                }
            )
            raise


    return wrapper
```

Prompt Inspector Utility:

PYTHON

```python
# core/debugging/prompt_inspector.py

class PromptInspector:

    """Utility to save and analyze prompts for debugging."""



    def __init__(self, save_dir: str = "./debug_prompts"):

        self.save_dir = Path(save_dir)

        self.save_dir.mkdir(exist_ok=True)



    def save_prompt(self,

                    prompt: str,

                    context: Dict[str, Any],

                    prefix: str = "prompt") -> str:

        """Save a prompt to file with metadata."""

        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")

        filename = f"{prefix}_{timestamp}.txt"

        filepath = self.save_dir / filename



        # Create metadata header

        metadata = {

            "timestamp": datetime.now().isoformat(),

            "context": context,

            "prompt_length": len(prompt),

            "estimated_tokens": len(prompt) // 4  # Rough estimate

        }



        content = f"METADATA:\n{json.dumps(metadata, indent=2)}\n\n"

        content += f"PROMPT:\n{prompt}"



        filepath.write_text(content)

        return str(filepath)



    def analyze_prompt(self, prompt: str) -> Dict[str, Any]:

        """Analyze prompt structure and composition."""

        # TODO 1: Split prompt by sections based on markers (## System, ## Tools, etc.)

        # TODO 2: Count tokens per section (using tiktoken or approximate)

        # TODO 3: Identify if any section exceeds recommended token budget

        # TODO 4: Check for missing required sections

        # TODO 5: Return analysis dict with warnings and suggestions



        analysis = {
```

```
            "total_length": len(prompt),
            "estimated_tokens": len(prompt) // 4,
            "sections": {},
            "warnings": [],
            "suggestions": []
        }

        # Placeholder implementation
        if len(prompt) > 8000:  # Rough token estimate
            analysis["warnings"].append("Prompt may exceed context window")
            analysis["suggestions"].append("Consider truncating conversation history")

        return analysis
```

E. Language-Specific Hints

Python Async Debugging Tips:

1. **Use `asyncio.create_task()` with proper naming**: Name your tasks for better debugging: `task = asyncio.create_task(coroutine(), name="tool_execution_web_search")`
2. **Enable debug mode**: Run your agent with `PYTHONASYNCIODEBUG=1` to get warnings about unawaited coroutines.
3. **Monitor event loop delays**: Use `loop.slow_callback_duration = 0.1` to log callbacks taking >100ms.
4. **Use `asyncio.gather()` with `return_exceptions=True`**: This prevents one failing task from canceling all others, making debugging easier.

Structured Logging Tips:

1. **Add correlation IDs at task start**: Generate a unique ID when `Agent.run_task` begins and pass it through all components.
2. **Log in JSON format for machine processing**: Use `json.dumps()` with `default=str` to handle datetime objects.
3. **Include component hierarchy in log names**: Use `logging.getLogger("archon.agents.ReActAgent")` for fine-grained control.

F. Debugging Tips

Symptom	How to Diagnose	Fix
Agent consistently chooses wrong tool	1. Check tool descriptions in logs 2. Test the prompt in playground with different tool ordering 3. Count how often each tool is called incorrectly	1. Rewrite descriptions to highlight distinctions 2. Add negative examples in few-shot 3. Implement tool confidence scoring
Memory retrieval slows down agent	1. Profile `LongTermMemory.query` execution time 2. Check vector store connection latency 3. Count number of memories retrieved per query	1. Add caching layer for frequent queries 2. Use approximate nearest neighbors (ANN) 3. Limit retrieval to top 3 most relevant
Multi-agent message bus deadlock	1. Log message queue sizes over time 2. Check for agents not processing their queues 3. Look for circular message dependencies	1. Implement queue size limits with backpressure 2. Add message TTL (time-to-live) 3. Use dead letter queue for unprocessable messages

10. Future Extensions

> **Milestone(s):** None (these are potential enhancements beyond the current project scope)

The architecture of Project Archon has been intentionally designed as a modular, extensible foundation. While the core framework addresses the fundamental challenges of autonomous AI agents—tool use, planning, memory, and collaboration—the design leaves room for numerous advanced capabilities that could be built upon this foundation. This section explores potential future directions, illustrating how the existing architecture can accommodate these extensions without requiring fundamental redesign.

10.1 Extension Ideas

The following table catalogs potential enhancement areas, organized by the architectural layer they would extend. Each idea includes a description of the functionality, how it would integrate with existing components, and key implementation considerations.

Extension Category	Core Idea	Integration Point	Key Considerations
Human-in-the-Loop	Enable human oversight, intervention, and guidance during agent execution	`FlowController` intercept points, `ToolRegistry` for human interaction tools	Must balance autonomy with human control; design non-blocking async interfaces for human responses
Advanced Tool Chaining	Allow tools to call other tools, enabling composition of complex workflows	`ToolExecutionEngine` with recursive execution, new `CompositeTool` type	Need to prevent infinite recursion, manage execution context, handle error propagation between chained tools
Learning from Experience	Agents improve performance over time by learning from past task outcomes	`LongTermMemory` extension with success/failure metadata, reinforcement learning module	Requires reward function definition, careful handling of negative examples to avoid harmful learning
Web UI & Dashboard	Visual interface for monitoring agent activity, inspecting plans, and providing input	New `UIServer` component that subscribes to system events via `MessageBus`	Real-time updates, historical playback, security of agent control endpoints
Multi-Modal Tools	Extend tool system to handle images, audio, video, and other media types	Enhanced `Tool` interface with MIME type support, new embedding providers for vector store	Token budget management for large media, specialized LLMs for multi-modal understanding
Scripting & Automation	Allow users to define reusable agent workflows as scripts or templates	New `WorkflowRegistry` storing predefined `Task` templates with parameterization	Template variable substitution, dependency injection of context, versioning of workflow definitions
Advanced Orchestration Patterns	Extend multi-agent collaboration with market-based, swarm, or federated patterns	Alternative `OrchestrationPattern` implementations beyond hierarchical	Complex coordination logic, performance overhead from agent negotiation, emergent behavior risks
External Knowledge Integration	Connect agents to live databases, knowledge graphs, and real-time data streams	New `KnowledgeConnector` tool category with streaming capabilities	Rate limiting, data freshness guarantees, handling of structured vs unstructured knowledge
Cross-Agent Tool Sharing	Allow agents to dynamically share tools with each other during collaboration	Extension to `MessageBus` with tool advertisement/discovery protocol	Security implications of tool access, permission models, tool compatibility checking
Explainability & Auditing	Generate detailed explanations of agent decisions and maintain audit trails	Enhanced logging with causal reasoning chains, new `ExplanationGenerator` component	Storage overhead, query performance for audit trails, privacy considerations for sensitive tasks

10.1.1 Human-in-the-Loop Integration

> **Mental Model:** The **Flight Control Center** for Autonomous Operations. Think of the agent as an autonomous spacecraft and human operators as mission control. The spacecraft operates independently most of the time, but for critical decisions, unforeseen situations, or when explicit approval is needed, it establishes a communication channel with mission control, awaits guidance, then proceeds with the updated instructions.

The current framework operates in fully autonomous mode. A human-in-the-loop extension would introduce controlled interruption points where humans can review, approve, redirect, or provide additional information. This addresses the fundamental tension between agent autonomy and safety/control requirements.

Integration Design:

1. **Interception Points in the Execution Flow:** The `FlowController.execute_task()` method would gain configuration options to define interception triggers:
   - **Pre-execution approval:** Before starting a `Task` with specific characteristics (uses certain tools, involves sensitive operations)
   - **Mid-execution checkpoints:** After completing specific `SubTask` milestones or when confidence scores fall below threshold
   - **Exception handling:** When `RecoveryOrchestrator` encounters errors above a certain severity level
2. **Human Interaction as Special Tools:** Human input would be modeled as special `Tool` instances (e.g., `HumanApprovalTool`, `HumanGuidanceTool`) that:
   - Block execution until human response (with configurable timeout)
   - Format requests in human-readable form (e.g., Markdown, with context summary)
   - Parse and validate human responses back into structured data
3. **Async Notification System:** A new `NotificationService` component would:

- Push interruption requests to configured channels (web UI, email, Slack)
- Maintain request state with correlation to the paused `ExecutionFlow`
- Resume execution when human response arrives or timeout expires

Architecture Decision: Human Response Handling Pattern

> **Decision: Async-Await with Timeout Fallback for Human Responses**
>
> - **Context:** Human responses are inherently asynchronous and unpredictable. The agent must wait for input but cannot block indefinitely if the human is unavailable.
> - **Options Considered:**
>   1. **Synchronous blocking:** Agent thread blocks indefinitely waiting for human response (simple but risks deadlock)
>   2. **Async with timeout:** Agent pauses execution, stores state, continues when response arrives or after timeout (preserves resources but requires state management)
>   3. **Optimistic continuation with rollback:** Agent proceeds with best guess, human can later approve/correct (maintains flow but complex rollback logic)
> - **Decision:** Option 2 (async with timeout) balances responsiveness with robustness. The agent pauses the specific `ExecutionFlow`, allowing other tasks to continue, and implements a timeout fallback to predefined policies (abort, continue with default, escalate to another human).
> - **Rationale:** This pattern matches real-world operational practices (pilots awaiting ATC clearance), allows efficient resource use (single agent can manage multiple paused flows), and provides clear failure modes when humans are unresponsive.
> - **Consequences:** Requires persistent storage of paused flow states, adds complexity to `FlowController` state management, introduces latency for tasks requiring human input.

10.1.2 Advanced Tool Chaining & Composition

> **Mental Model: Function Composition in Mathematics.** Just as mathematical functions can be composed (f(g(x))) to create more complex operations, tools can be chained together where the output of one tool becomes the input to another. This enables building complex capabilities from simpler, well-tested primitives.

The current tool system treats each tool as an independent, atomic operation. Advanced tool chaining would allow tools to call other tools, enabling the creation of higher-level abstractions and reusable workflows.

Integration Design:

1. **CompositeTool Type:** A new `CompositeTool` class implementing the `Tool` interface that:
   - Contains a directed graph of tool calls with data flow between them
   - Defines input/output mappings between consecutive tools
   - Handles error propagation and partial execution rollback
2. **Tool Execution Context Extension:** The `ToolExecutionEngine` would be enhanced to:
   - Maintain a call stack for nested tool executions
   - Propagate execution context (user ID, session data) through the chain
   - Implement circuit breakers at both individual tool and composite tool levels
3. **Visual Workflow Builder:** A companion UI component allowing users to:
   - Drag-and-drop tools to create workflows
   - Define conditional logic and loops in the tool chain
   - Test workflows with sample data before deployment

Example Composite Tool: "Research and Summarize"

```
CompositeTool: research_and_summarize(topic: str, depth: str) -> str
  Steps:
  1. web_search(topic, num_results=5) → list[search_results]
  2. For each result in search_results:
      a. fetch_webpage(url) → webpage_content
      b. extract_key_points(webpage_content) → key_points
  3. synthesize_summary(all_key_points, depth) → final_summary
  4. Return final_summary
```
PLAINTEXT

Common Pitfalls in Tool Chaining:

- ⚠️ **Pitfall: Infinite Recursion Chains**
  Description: A composite tool inadvertently includes itself in its own chain, either directly or through intermediate tools, causing infinite recursion.

Why Wrong: Causes stack overflow, consumes resources indefinitely, halts agent execution.
Fix: Implement cycle detection in composite tool validation, maintain visited tool set during execution, enforce maximum recursion depth.

- ⚠️ **Pitfall: Context Bleed Between Chains**
  Description: Execution context from one tool chain unintentionally influences another chain running concurrently.
  Why Wrong: Leads to unpredictable behavior, security issues if sensitive context leaks.
  Fix: Implement isolated execution contexts for each composite tool invocation, use functional updates for context propagation.

10.1.3 Learning from Experience via Reinforcement

> **Mental Model: Professional Apprenticeship.** Just as an apprentice learns from both successes and mistakes under a master's guidance, the agent can improve its performance over time by analyzing which actions led to successful outcomes and which led to failures, adjusting its future behavior accordingly.

The current memory system stores experiences but doesn't actively learn from them to improve future performance. A reinforcement learning extension would enable the agent to optimize its planning and tool selection strategies based on historical outcomes.

Integration Design:

1. **Experience Replay Storage:** Extend `LongTermMemory` to store:
   - Complete execution traces (state, action, reward, next state)
   - Success/failure labels with human or automated feedback
   - Metadata about execution conditions (time, resource usage, complexity)
2. **Reward Function Service:** A new component that:
   - Computes reward signals from execution outcomes (speed, accuracy, cost)
   - Incorporates explicit human feedback scores
   - Balances multiple objectives (efficiency vs. thoroughness)
3. **Policy Optimization Module:** Integrates with the `Planner` and `ReActAgent` to:
   - Fine-tune prompt templates based on successful patterns
   - Adjust tool selection probabilities (prefer tools with high success rates)
   - Learn when to replan vs. retry failed subtasks

Training Data Flow:

```
Execution Complete →
  Store Trace in Experience Replay →
    Periodic Batch Training →
      Update Policy Model →
        Deploy Updated Model to Agents
```

Architecture Decision: Online vs. Offline Learning

> **Decision: Hybrid Online/Offline Learning with Human Oversight**
>
> - **Context:** Agents operate in production environments where mistakes can have real consequences. We need learning that improves performance without introducing unpredictable behavior during learning.
> - **Options Considered:**
>   1. **Pure offline learning:** Collect traces, train periodically in isolated environment, deploy updated models (safe but delayed improvement)
>   2. **Pure online learning:** Continuously update policy during execution (fast adaptation but risk of learning harmful behaviors)
>   3. **Hybrid approach:** Offline learning for major updates, online tuning of safe parameters (e.g., tool selection weights) with bounds
> - **Decision:** Option 3 (hybrid) with human approval gate for significant policy changes. Minor adjustments (tool preference weights) update online within safe bounds, while major architecture changes (new planning strategies) require offline training and human review.
> - **Rationale:** Balances safety with adaptability. Most real-world autonomous systems (self-driving cars, industrial robots) use similar hybrid approaches where core models update offline but tuning parameters adjust online within constraints.
> - **Consequences:** Requires versioning of agent policies, A/B testing infrastructure for new policies, clear rollback procedures if new policies degrade performance.

10.1.4 Web UI & Dashboard for Monitoring

> **Mental Model: Air Traffic Control Radar Display.** Just as air traffic controllers monitor multiple aircraft on a radar screen with overlays showing altitude, speed, and trajectory, the Web UI provides a real-time visualization of multiple agent activities, their current state, execution plans, and any issues requiring attention.

While the framework is designed as a backend system, a Web UI extension would dramatically improve operational visibility, debugging capabilities, and user interaction.

Integration Design:

1. **Event Streaming Layer:** Extend the `MessageBus` to:
   - Broadcast system events (agent state changes, tool executions, errors) to WebSocket clients
   - Support filtering and subscription by event type, agent ID, or task ID
   - Buffer events during client disconnection with configurable retention
2. **UI Server Component:** A new FastAPI/Flask service that:
   - Serves static frontend assets and API endpoints
   - Manages WebSocket connections for real-time updates
   - Provides REST API for historical querying and control actions
3. **Frontend Dashboard:** React/Vue application with:
   - Real-time visualization of agent activity (similar to process monitoring tools)
   - Interactive plan editor for modifying ongoing tasks
   - Chat interface for direct interaction with agents
   - Analytics views showing performance metrics over time

Key UI Views and Their Data Sources:

View	Purpose	Data Source
Agent Fleet Overview	Monitor all active agents	`AgentState` events via `MessageBus`
Task Execution Timeline	Visualize subtask dependencies and progress	`ExecutionFlow` state, `SubTask` status updates
Memory Explorer	Inspect and search agent memories	`LongTermMemory.query()` API
Tool Performance Dashboard	Monitor tool success rates and latency	`ToolExecutionEngine` metrics
Prompt Playground	Test and refine LLM prompts	Direct `MockLLM` interface with real prompt inspection

Implementation Consideration: Security Model The UI would need robust authentication and authorization since it provides control capabilities over agents. Recommended approach: role-based access control (RBAC) with:

- **Viewer role:** Read-only access to monitoring data
- **Operator role:** Can pause/resume agents, provide human input
- **Admin role:** Can modify tool registry, update agent configurations
- **Auditor role:** Access to full execution logs for compliance

10.1.5 Multi-Modal Tool Extension

> **Mental Model: Swiss Army Knife with Specialized Attachments.** The core tool system is like a basic Swiss Army knife with standard tools (knife, screwdriver). Multi-modal extensions add specialized attachments (magnifying glass, saw, file) that handle different material types (text, images, audio) but all attach to the same base handle (the tool interface).

Current tools primarily handle text inputs and outputs. Multi-modal extensions would enable tools that process images, audio, video, and other media types, significantly expanding the agent's capabilities in real-world applications.

Integration Design:

1. **Enhanced Tool Interface:** Extend the `Tool` base class with:
   - `input_mime_types: List[str]` - Supported input formats
   - `output_mime_types: List[str]` - Supported output formats
   - `max_input_size: int` - Maximum file size in bytes
   - Media-specific validation logic

2. **Media Processing Pipeline:** New components for:
    - **Media embedding:** Generate vector embeddings for images/audio to store in `LongTermMemory`
    - **Media chunking:** Split large files (videos, long audio) into processable segments
    - **Format conversion:** Transcode between formats for tool compatibility
3. **Multi-Modal LLM Integration:** Extend LLM interface to support:
    - Vision models that accept image URLs or base64-encoded images
    - Audio transcription models
    - Multi-modal prompt templates that mix text and media references

Example Multi-Modal Tools:

Tool Name	Input	Output	Use Case
`image_analyzer`	Image file (PNG, JPEG)	JSON description of image content	Visual question answering, image captioning
`document_ocr`	Scanned PDF/image	Extracted text with layout metadata	Processing scanned documents, receipts
`audio_transcriber`	Audio file (MP3, WAV)	Text transcript with speaker diarization	Meeting notes, podcast summarization
`video_summarizer`	Video file (MP4)	Key frames + audio transcript	Video content analysis, highlight generation

Challenges and Mitigations:

1. **Token Budget Management:** Media files consume significant context window space when encoded as text (base64) or described in detail. Mitigation: Implement intelligent compression strategies—store media in external storage, reference by URL in prompts, use embeddings for similarity search rather than raw content.
2. **Tool Compatibility:** Not all LLMs support multi-modal inputs. Mitigation: Implement fallback strategies—extract text descriptions from media first, then process with text-only LLM; use specialized models for specific modalities with results fed back to main agent.
3. **Performance Latency:** Media processing is computationally intensive. Mitigation: Implement async processing pipelines, support streaming partial results, use GPU acceleration where available.

10.1.6 Scripting & Automation Workflows

> **Mental Model: Macro Recorder for Complex Tasks.** Similar to how spreadsheet users record macros to automate repetitive multi-step operations, this extension allows users to capture successful agent executions as reusable templates that can be parameterized and invoked with different inputs.

While the `Planner` decomposes tasks dynamically, many real-world workflows follow predictable patterns that could be captured and reused. Scripting enables users to define parameterized templates for common task types.

Integration Design:

1. **Workflow Template Registry:** New component storing:
    - Parameterized `Task` templates with placeholder variables
    - Input/output schemas for template validation
    - Version history and dependency tracking
2. **Template Language:** A YAML/JSON-based DSL for defining workflows:

```yaml
workflow:
  name: "weekly_research_report"
  parameters:
    - name: "topic"
      type: "string"
      description: "Research topic"
  steps:
    - tool: "web_search"
      args:
        query: "{{topic}} latest developments 2024"
    - tool: "summarize_documents"
      args:
        documents: "{{step1.results}}"
    - tool: "format_report"
      args:
        content: "{{step2.summary}}"
        template: "weekly_research"
```

3. **Template Instantiation Engine:** Extends `FlowController` to:
   - Parse templates and substitute parameters with actual values
   - Validate that required tools are available in registry
   - Execute templated workflows with the same monitoring as dynamic tasks

Advanced Features:

- **Conditional Logic:** Templates can include if/else branches based on intermediate results
- **Loops:** Repeat steps for each item in a list (e.g., process each search result)
- **Error Handling Blocks:** Define fallback steps when specific tools fail
- **Human Approval Gates:** Insert human review points at defined steps

Use Case Example: Customer Support Ticket Triage

```yaml
workflow:
  name: "support_ticket_triage"
  parameters:
    - name: "ticket_text"
      type: "string"
  steps:
    - tool: "classify_issue"
      args:
        text: "{{ticket_text}}"
      save_as: "issue_type"

    - if: "{{issue_type}} == 'billing'"
      then:
        - tool: "fetch_customer_record"
          args:
            extracted_from: "{{ticket_text}}"
        - tool: "generate_billing_response"
      else:
        - tool: "escalate_to_human"
          args:
            reason: "Non-billing issue requires human review"
```

Integration with Dynamic Planning: Scripted workflows can serve as building blocks within larger dynamic plans. The `Planner` could recognize when a subtask matches a known template pattern and substitute the template execution rather than decomposing from scratch, improving efficiency and consistency.

10.1.7 Cross-Agent Tool Sharing & Dynamic Capability Discovery

> **Mental Model: Conference Room Whiteboard Session.** Imagine agents as specialists in a meeting. When one agent encounters a task outside their expertise, they can ask the room, "Does anyone know how to do X?" Another agent raises their hand, "I have a tool for that—let me show you how it works." They share not just the tool but also usage examples and limitations.

In the current multi-agent system, each agent has a fixed set of tools defined by its role. Cross-agent tool sharing would enable dynamic capability discovery and utilization, allowing agents to temporarily "borrow" tools from peers when needed.

Integration Design:

1. **Tool Advertisement Protocol:** Extend the `MessageBus` to support:
   - Tool description broadcasts when agents join the system
   - Capability queries ("Who can perform operation X?")
   - Usage agreement negotiation (temporary access grants)
2. **Dynamic Tool Proxy:** New `RemoteTool` class that:
   - Implements the `Tool` interface but forwards execution to another agent
   - Handles authentication and permission checking for tool access
   - Manages lifecycle of temporary tool grants (lease expiration)
3. **Capability Registry:** Distributed registry tracking:
   - Which agents have which tools available
   - Tool performance metrics (success rate, latency)
   - Access control policies for sensitive tools

Message Flow for Tool Sharing:

```
Agent A needs tool T →
  Broadcasts "capability_query(T)" →
    Agent B responds "tool_offer(T, terms)" →
      Agent A sends "tool_request(T, context)" →
        Agent B validates, creates lease →
          Agent B sends "tool_lease(T, proxy_endpoint)" →
            Agent A registers RemoteTool, uses it
```

Security Considerations:

- **Least Privilege:** Tools should be shared with minimal necessary permissions
- **Audit Trail:** All cross-agent tool usage must be logged with full context
- **Revocation:** Tool owners must be able to immediately revoke access
- **Sandboxing:** Remote tools should execute in the providing agent's environment, not the requesting agent's, to maintain security boundaries

Performance Optimization: Frequently shared tools could be cached as code (with owner approval) rather than executed remotely after establishing trust relationships, reducing latency. The system could implement a reputation system where agents with high success rates have their tools cached more aggressively.

10.1.8 Explainability & Audit Trail Generation

> **Mental Model: Black Box Flight Recorder + Investigative Journalist.** The framework currently operates like a black box—we see inputs and outputs but limited insight into internal reasoning. This extension adds both a flight recorder (comprehensive data capture) and an investigative journalist (post-hoc analysis that reconstructs and explains the "why" behind decisions).

As AI agents take on more consequential tasks, the ability to explain their reasoning and maintain audit trails becomes critical for debugging, compliance, and trust.

Integration Design:

1. **Comprehensive Execution Tracing:** Enhance all components to emit structured trace events:
   - `Planner`: Decision points, alternative plans considered, confidence scores
   - `ReActAgent`: Full thought chains, tool selection rationale, uncertainty expressions
   - `ToolExecutionEngine`: Input/output snapshots, performance metrics, error details
2. **Causal Graph Construction:** Post-execution analysis that:
   - Reconstructs the decision pathway from final answer back through reasoning steps
   - Identifies critical junctures where different choices would have led to different outcomes
   - Calculates confidence intervals for conclusions based on source reliability
3. **Natural Language Explanation Generator:** Component that:
   - Translates execution traces into human-readable narratives
   - Highlights key decisions and supporting evidence
   - Identifies potential biases or limitations in the reasoning process

Explanation Formats:

Audience	Format	Detail Level	Example
End User	Simple summary paragraph	High-level	"The agent determined X because it found evidence Y and Z through web search."
Developer	Structured JSON with citations	Medium	`{"conclusion": "X", "evidence": [{"source": "tool T", "result": "Y"}, ...]}`
Auditor/Compliance	Complete trace with timestamps	Verbose	Timeline view showing every thought, tool call, and observation with exact timing

Implementation Strategy: Implement as a layered system where basic tracing is always on (low overhead), detailed causal analysis runs post-execution for important tasks, and natural language explanations generate on-demand.

Architecture Decision: Trace Storage vs. Regeneration

Decision: Hybrid Store-and-Regenerate with Configurable Detail Levels

- **Context:** Complete execution traces can be large (megabytes per complex task). Storing everything for all tasks is expensive, but regenerating explanations from minimal data may lose fidelity.
- **Options Considered:**
  1. **Store everything:** Keep complete traces for all executions (maximal fidelity, high storage cost)
  2. **Store minimal, regenerate on demand:** Keep only final results, regenerate traces by re-running (low storage, but may not reproduce exact reasoning)
  3. **Hybrid with detail levels:** Store structured summaries always, complete traces for important tasks, regenerate intermediate details when needed
- **Decision:** Option 3 with configurable retention policies. All tasks store structured summaries (key decisions, outcomes). Important tasks (by user marking, error occurrence, or sensitive domain) store complete traces. System can regenerate intermediate details by replaying from summaries when needed.
- **Rationale:** Balances storage costs with explanation fidelity. Critical tasks (medical, financial, legal) require complete audit trails, while routine tasks need only summaries. The regeneration capability provides flexibility when unexpected questions arise.
- **Consequences:** Requires versioning of agent components to ensure regenerated traces match original execution, adds complexity to trace serialization/deserialization.

10.2 Implementation Guidance for Future Extensions

While the full implementation of these extensions is beyond the current project scope, this section provides starter patterns and integration points to guide future development.

A. Technology Recommendations for Extensions

Extension	Core Technology	Integration Approach
Human-in-the-Loop	WebSocket + FastAPI for real-time UI, Redis for state persistence	Extend `FlowController` with pause/resume methods, add `HumanInteractionTool` base class
Tool Chaining	NetworkX for DAG representation, RxPY for reactive data flows	Implement `CompositeTool` using existing `Tool` interface, extend `ToolExecutionEngine` with stack tracking
Learning from Experience	PyTorch for RL, MLflow for experiment tracking, Weights & Biases for visualization	Create `ExperienceReplayBuffer` class, integrate reward computation into `ToolResult` metadata
Web UI Dashboard	React + TypeScript frontend, FastAPI backend, Socket.IO for real-time updates	Add `EventBroadcaster` component that publishes `MessageBus` events to WebSocket
Multi-Modal Tools	CLIP for image embeddings, Whisper for audio, OpenCV for video, Pillow for images	Extend `Tool` with media handling methods, create `MediaProcessor` utility class
Scripting Workflows	PyYAML for template parsing, Jinja2 for variable substitution	Implement `WorkflowTemplate` class, extend `Planner` to recognize template patterns
Tool Sharing	OAuth2 for authentication, Protocol Buffers for RPC, Redis for distributed locks	Create `ToolLeaseManager`, implement `RemoteToolProxy` class
Explainability	spaCy for NLP, NetworkX for causal graphs, Matplotlib/Plotly for visualization	Add `ExecutionTracer` decorator to key methods, create `ExplanationGenerator` service

B. Recommended Module Structure for Extensions

```
project-archon/
├── core/                    # Existing core framework
│   ├── agent/
│   ├── tools/
│   ├── planning/
│   ├── memory/
│   └── orchestration/
├── extensions/              # New extensions directory
│   ├── human_in_the_loop/
│   │   ├── interceptors.py          # Flow interception points
│   │   ├── human_tools.py           # HumanInteractionTool base class
│   │   └── notification_service.py  # Async notification system
│   ├── tool_chaining/
│   │   ├── composite_tool.py        # CompositeTool implementation
│   │   ├── workflow_dsl.py          # YAML/JSON DSL parser
│   │   └── visual_builder.py        # UI for building tool chains
│   ├── learning/
│   │   ├── experience_replay.py     # Store execution traces
│   │   ├── reward_calculator.py     # Compute reward signals
│   │   └── policy_optimizer.py      # Update agent behavior
│   ├── ui_dashboard/
│   │   ├── backend/
│   │   │   ├── server.py            # FastAPI server
│   │   │   ├── websocket_handler.py # Real-time events
│   │   │   └── api/                 # REST endpoints
│   │   └── frontend/                # React application
│   │       ├── src/
│   │       └── public/
│   ├── multimodal/
│   │   ├── media_tool.py            # Base class for media tools
│   │   ├── embedding_generators.py  # CLIP, Whisper integrations
│   │   └── media_processor.py       # Chunking, transcoding utilities
│   ├── scripting/
│   │   ├── workflow_registry.py     # Store and retrieve templates
│   │   ├── template_engine.py       # Parse and instantiate templates
│   │   └── template_library/        # Predefined workflow templates
│   ├── tool_sharing/
│   │   ├── capability_protocol.py   # Advertisement/discovery messages
│   │   ├── remote_tool.py           # Proxy for remote tool execution
│   │   └── lease_manager.py         # Track and expire tool leases
│   └── explainability/
│       ├── execution_tracer.py      # Decorate methods for tracing
│       ├── causal_analyzer.py       # Reconstruct decision pathways
│       └── explanation_generator.py # Create human-readable narratives
└── tests/
    └── extensions/           # Tests for each extension
```

C. Starter Code for Human-in-the-Loop Extension

Human Interaction Tool Base Class:

PYTHON

```python
# extensions/human_in_the_loop/human_tools.py

from typing import Any, Dict, Optional, Callable

from dataclasses import dataclass

from datetime import datetime

import asyncio

from enum import Enum


from core.tools import Tool, ToolResult


class HumanResponseStatus(Enum):

    PENDING = "pending"

    APPROVED = "approved"

    REJECTED = "rejected"

    TIMEOUT = "timeout"

    CANCELLED = "cancelled"


@dataclass

class HumanRequest:

    """Represents a request for human input."""

    request_id: str

    task_id: str

    subtask_id: Optional[str]

    question: str

    context: Dict[str, Any]

    options: Optional[Dict[str, Any]] = None

    created_at: datetime = None

    timeout_seconds: float = 300.0  # 5 minute default



    def __post_init__(self):

        if self.created_at is None:

            self.created_at = datetime.now()


class HumanInteractionTool(Tool):

    """Base class for tools that require human interaction."""



    def __init__(self, notification_callback: Callable[[HumanRequest], None]):

        self.notification_callback = notification_callback

        self.pending_requests: Dict[str, asyncio.Future] = {}



    async def execute(self, **kwargs) -> ToolResult:

        """

        Base implementation for human interaction tools.
```

```
        Subclasses should call await self._await_human_response()
        """
        raise NotImplementedError("Subclasses must implement execute")

    async def _await_human_response(
        self,
        request: HumanRequest
    ) -> Optional[Dict[str, Any]]:
        """
        Send request to human and await response with timeout.
        Returns parsed response dict or None if timeout/cancelled.
        """
        # Create future for this request
        loop = asyncio.get_event_loop()
        future = loop.create_future()
        self.pending_requests[request.request_id] = future

        try:
            # Notify human via callback (e.g., to Web UI, email, Slack)
            self.notification_callback(request)

            # Wait for response with timeout
            response = await asyncio.wait_for(
                future,
                timeout=request.timeout_seconds
            )
            return response

        except asyncio.TimeoutError:
            # Handle timeout - return default or raise
            return {"status": HumanResponseStatus.TIMEOUT, "response": None}
        finally:
            # Clean up pending request
            self.pending_requests.pop(request.request_id, None)

    def provide_response(
        self,
        request_id: str,
        response_data: Dict[str, Any]
    ) -> bool:
```

```
        """
        Called by external system (e.g., Web UI) to provide human response.
        Returns True if request was found and response delivered.
        """
        if request_id in self.pending_requests:
            future = self.pending_requests[request_id]
            if not future.done():
                future.set_result(response_data)
                return True
        return False

class HumanApprovalTool(HumanInteractionTool):
    """Tool that requests human approval before proceeding."""

    def __init__(self, notification_callback):
        super().__init__(notification_callback)
        self.name = "human_approval"
        self.description = "Request human approval before proceeding with sensitive operation"
        self.parameters_schema = {
            "type": "object",
            "properties": {
                "operation_description": {
                    "type": "string",
                    "description": "Description of what needs approval"
                },
                "sensitive_data_summary": {
                    "type": "string",
                    "description": "Summary of sensitive data involved"
                },
                "timeout_minutes": {
                    "type": "number",
                    "description": "Timeout in minutes before automatic rejection",
                    "default": 5
                }
            },
            "required": ["operation_description"]
        }

    async def execute(self, **kwargs) -> ToolResult:
        # Create human request
        request = HumanRequest(
```

```
            request_id=f"approval_{datetime.now().timestamp()}",
            task_id=kwargs.get("task_id", "unknown"),
            subtask_id=kwargs.get("subtask_id"),
            question=f"Approve operation: {kwargs['operation_description']}",
            context={
                "sensitive_data_summary": kwargs.get("sensitive_data_summary"),
                "full_operation": kwargs
            },
            options={
                "approve": "Proceed with operation",
                "reject": "Cancel operation",
                "modify": "Request modifications"
            },
            timeout_seconds=kwargs.get("timeout_minutes", 5) * 60
        )

        # Await human response
        response = await self._await_human_response(request)

        if response and response.get("status") == HumanResponseStatus.APPROVED:
            return ToolResult.success(
                content="Human approval granted",
                metadata={"human_response": response}
            )
        else:
            return ToolResult.error(
                error_msg="Human approval denied or timed out",
                metadata={"human_response": response}
            )
```

Flow Controller Interceptor Integration:

PYTHON

```python
# extensions/human_in_the_loop/interceptors.py

from typing import Dict, Any, Optional, Callable

from enum import Enum

from dataclasses import dataclass


from core.flow_controller import FlowController, ExecutionFlow


class InterceptionTrigger(Enum):

    """Events that can trigger human interception."""

    BEFORE_TOOL_EXECUTION = "before_tool_execution"

    AFTER_SUBTASK_COMPLETE = "after_subtask_complete"

    ON_ERROR = "on_error"

    ON_SENSITIVE_OPERATION = "on_sensitive_operation"

    ON_PLAN_CHANGE = "on_plan_change"


@dataclass

class InterceptionRule:

    """Rule defining when to intercept execution for human input."""

    trigger: InterceptionTrigger

    condition: Callable[[Dict[str, Any]], bool]  # Returns True if should intercept

    human_tool_name: str  # Which human interaction tool to use

    tool_arguments: Dict[str, Any]  # Arguments to pass to human tool

    priority: int = 0  # Higher priority rules execute first


class HumanInTheLoopFlowController(FlowController):

    """Extended FlowController with human interception support."""



    def __init__(self, *args, **kwargs):

        super().__init__(*args, **kwargs)

        self.interception_rules: Dict[InterceptionTrigger, list[InterceptionRule]] = {}

        self.human_tools: Dict[str, Any] = {}  # Registered human interaction tools



    def add_interception_rule(self, rule: InterceptionRule):

        """Add a rule for human interception."""

        if rule.trigger not in self.interception_rules:

            self.interception_rules[rule.trigger] = []

        self.interception_rules[rule.trigger].append(rule)

        # Sort by priority (highest first)

        self.interception_rules[rule.trigger].sort(

            key=lambda r: r.priority,

            reverse=True

        )
```

```python
def register_human_tool(self, tool_name: str, tool: Any):
    """Register a human interaction tool."""
    self.human_tools[tool_name] = tool


async def _check_and_intercept(
    self,
    trigger: InterceptionTrigger,
    context: Dict[str, Any]
) -> bool:
    """
    Check rules for given trigger and intercept if any match.
    Returns True if execution was intercepted (paused for human input).
    """
    if trigger not in self.interception_rules:
        return False

    for rule in self.interception_rules[trigger]:
        if rule.condition(context):
            # Execute human interaction tool
            if rule.human_tool_name in self.human_tools:
                tool = self.human_tools[rule.human_tool_name]
                result = await tool.execute(
                    **rule.tool_arguments,
                    **context
                )

                # If human denied or error, stop execution
                if result.status != ToolResultStatus.SUCCESS:
                    # Update flow state to reflect human intervention
                    flow_id = context.get("flow_id")
                    if flow_id in self.active_flows:
                        flow = self.active_flows[flow_id]
                        flow.state = AgentState.ERROR
                        flow.context["human_intervention"] = {
                            "result": result,
                            "trigger": trigger.value
                        }
                    return True  # Execution was intercepted and stopped
```

```
                # Human approved, continue execution
                return False

    return False  # No rules matched, no interception

async def execute_task(self, user_query: str) -> str:
    """Override to add interception checks at key points."""
    # Original flow setup
    flow = await self._create_execution_flow(user_query)

    # Check for pre-execution interception
    if await self._check_and_intercept(
        InterceptionTrigger.BEFORE_TOOL_EXECUTION,
        {
            "flow_id": flow.task_id,
            "user_query": user_query,
            "agent": self.agent
        }
    ):
        return "Task paused for human approval before starting"

    # Continue with original execution (simplified)
    # In full implementation, would intercept at multiple points
    return await super().execute_task(user_query)
```

D. Core Logic Skeleton for Composite Tool Implementation

```python
# extensions/tool_chaining/composite_tool.py

from typing import Dict, Any, List, Optional, Tuple

from dataclasses import dataclass, field

import networkx as nx

import json


from core.tools import Tool, ToolResult, ToolResultStatus

from core.tool_execution_engine import ToolExecutionEngine


@dataclass

class ToolNode:

    """Represents a tool invocation in a composite tool."""

    node_id: str

    tool_name: str

    tool_arguments: Dict[str, Any]

    # Mapping of output fields to input fields of downstream nodes

    output_mapping: Dict[str, str] = field(default_factory=dict)


@dataclass

class CompositeTool(Tool):

    """

    A tool composed of multiple tool executions with data flow between them.

    Executes tools in order defined by a directed acyclic graph (DAG).

    """


    def __init__(self, name: str, description: str):

        self._name = name

        self._description = description

        self.graph = nx.DiGraph()

        self.nodes: Dict[str, ToolNode] = {}

        self.parameters_schema = self._build_parameters_schema()


    @property

    def name(self) -> str:

        return self._name


    @property

    def description(self) -> str:

        return self._description
```

```python
    @property
    def parameters_schema(self) -> Dict[str, Any]:
        return self._parameters_schema

    def add_node(self, node: ToolNode) -> None:
        """
        Add a tool node to the composite tool graph.

        TODO 1: Validate that tool_name exists in the tool registry
        TODO 2: Generate unique node_id if not provided
        TODO 3: Add node to internal graph and nodes dictionary
        TODO 4: Update parameters_schema to include any new input parameters
        """
        pass

    def add_edge(self, from_node_id: str, to_node_id: str) -> None:
        """
        Add dependency edge between two tool nodes.
        from_node must complete before to_node starts.

        TODO 1: Check that both nodes exist in the graph
        TODO 2: Check that adding edge doesn't create a cycle
        TODO 3: Add edge to the NetworkX graph
        TODO 4: Validate that output_mapping is compatible between nodes
        """
        pass

    def validate(self, tool_registry) -> List[str]:
        """
        Validate the composite tool structure.
        Returns list of error messages, empty list if valid.

        TODO 1: Check graph is acyclic (no circular dependencies)
        TODO 2: Verify all referenced tools exist in registry
        TODO 3: Validate parameter schemas match between connected nodes
        TODO 4: Check that all required inputs are provided (either from parameters or upstream nodes)
        TODO 5: Ensure output mappings reference valid fields
        """
        errors = []
        # Implementation
```

```
        return errors


    async def execute(self, **kwargs) -> ToolResult:
        """
        Execute the composite tool by running nodes in topological order.

        TODO 1: Validate input parameters against parameters_schema
        TODO 2: Get topological order of nodes from graph
        TODO 3: For each node in order:
            a. Build arguments by combining:
               - User-provided parameters (for input nodes)
               - Outputs from upstream nodes (based on output_mapping)
               - Constant values from tool_arguments
            b. Execute tool via ToolExecutionEngine
            c. If tool fails, decide whether to continue, retry, or abort
            d. Store results for downstream nodes
        TODO 4: Collect final outputs from output nodes
        TODO 5: Format combined result with execution trace
        TODO 6: Handle errors and partial execution gracefully
        """
        try:
            # Initialize execution context
            context = kwargs.copy()
            results = {}

            # Get execution order
            execution_order = list(nx.topological_sort(self.graph))

            for node_id in execution_order:
                node = self.nodes[node_id]

                # Build arguments for this node
                node_args = self._build_node_arguments(node, context, results)

                # TODO: Execute tool with timeout and error handling
                # tool_result = await tool_engine.execute_tool_call(...)

                # Store result for downstream nodes
                results[node_id] = {"args": node_args, "result": None}  # placeholder
```

```
            # Format final result
            final_output = self._format_final_output(results, execution_order)
            return ToolResult.success(
                content=final_output,
                metadata={"execution_trace": results}
            )

        except Exception as e:
            return ToolResult.error(
                error_msg=f"Composite tool execution failed: {str(e)}",
                metadata={"partial_results": results}
            )

    def _build_parameters_schema(self) -> Dict[str, Any]:
        """
        Build combined JSON schema for composite tool parameters.

        TODO 1: Identify all input nodes (nodes with no incoming edges)
        TODO 2: For each input node, extract required parameters from tool's schema
        TODO 3: Merge parameter schemas, handling conflicts
        TODO 4: Return combined JSON schema
        """
        return {
            "type": "object",
            "properties": {},
            "required": []
        }

    def _build_node_arguments(
        self,
        node: ToolNode,
        user_inputs: Dict[str, Any],
        previous_results: Dict[str, Any]
    ) -> Dict[str, Any]:
        """
        Build arguments for a node by combining sources.

        TODO 1: Start with constant arguments from node.tool_arguments
        TODO 2: For each parameter, check if mapped from upstream node output
        TODO 3: If mapped, extract value from previous_results
```

```
        TODO 4: Otherwise, check if in user_inputs (for input nodes)
        TODO 5: Validate final arguments against tool's parameter schema
        TODO 6: Return combined arguments dictionary
        """
        args = {}
        # Implementation
        return args

    def _format_final_output(
        self,
        results: Dict[str, Any],
        execution_order: List[str]
    ) -> str:
        """
        Format the final output from all node results.

        TODO 1: Identify output nodes (nodes with no outgoing edges)
        TODO 2: Extract relevant content from each output node's result
        TODO 3: Combine into coherent final output
        TODO 4: Include execution summary if verbose output requested
        """
        return "Composite tool execution completed"
```

E. Language-Specific Hints for Extensions

Python Async Patterns for Human-in-the-Loop:

PYTHON

```python
# Use asyncio.Queue for decoupling human responses from request generation
human_response_queue = asyncio.Queue()

# Pattern for timeout with default fallback
try:
    response = await asyncio.wait_for(
        human_response_queue.get(),
        timeout=300.0  # 5 minutes
    )
except asyncio.TimeoutError:
    response = {"status": "timeout", "action": "use_default"}

# Use asyncio.Event for signaling human response received
response_event = asyncio.Event()
# UI calls response_event.set() when human responds
# Agent awaits response_event.wait() with timeout
```

Performance Optimization for Multi-Modal Tools:

```python
# Use lazy loading for expensive media processing libraries
def get_image_processor():
    # Import only when needed
    import torch
    import clip
    from PIL import Image
    # Return processor function

# Implement streaming for large media files
async def process_video_in_chunks(video_path, chunk_size=1024*1024):
    with open(video_path, 'rb') as f:
        while chunk := f.read(chunk_size):
            yield await process_chunk(chunk)

# Use LRU cache for embedding models
from functools import lru_cache

@lru_cache(maxsize=10)
def get_embedding_model(model_name="clip-vit-base-patch32"):
    import clip
    model, preprocess = clip.load(model_name)
    return model, preprocess
```

Security Considerations for Tool Sharing:

```python
# Implement permission checking for cross-agent tool access

def check_tool_permission(tool_name, requesting_agent, owner_agent):

    # Check if tool is sharable

    if not tool_registry.get_tool(tool_name).sharable:

        return False



    # Check if requesting agent has required role

    if not has_required_role(requesting_agent, tool_name):

        return False



    # Check rate limits

    if exceeded_rate_limit(requesting_agent, tool_name):

        return False



    return True


# Use JWT tokens for temporary tool access

def create_tool_access_token(tool_name, requesting_agent, duration_minutes=10):

    import jwt

    import datetime



    payload = {

        "tool": tool_name,

        "requester": requesting_agent.id,

        "exp": datetime.datetime.utcnow() + datetime.timedelta(minutes=duration_minutes),

        "permissions": ["execute"]  # No modify permissions

    }



    return jwt.encode(payload, SECRET_KEY, algorithm="HS256")
```

PYTHON

F. Milestone Checkpoint for Extensions

Testing Human-in-the-Loop Extension:

```bash
# Run the human interaction tool tests
python -m pytest extensions/human_in_the_loop/tests/ -v

# Expected output should show:
# √ HumanApprovalTool sends notification
# √ HumanApprovalTool times out after configured period
# √ FlowController correctly pauses on interception rule
# √ Human response correctly resumes execution

# Manual test: Start agent with human interception enabled
python examples/human_interception_example.py

# Expected behavior:
# 1. Agent starts task execution
# 2. Pauses when reaching sensitive operation
# 3. Sends notification to configured channel (console, in test)
# 4. Waits for human input
# 5. When human responds "approve", continues execution
# 6. When human responds "reject", stops with error message
```

Verifying Composite Tool Execution:

```bash
# Test composite tool validation and execution
python -m pytest extensions/tool_chaining/tests/test_composite_tool.py

# Expected test cases:
# √ CompositeTool validates acyclic graph
# √ CompositeTool executes nodes in topological order
# √ CompositeTool handles node failure gracefully
# √ CompositeTool correctly maps outputs between nodes

# Visual verification of tool chain DAG
python tools/visualize_composite_tool.py --tool research_and_summarize

# Should generate a PNG showing:
# - Nodes as boxes with tool names
# - Edges showing data flow between nodes
# - Highlighted execution path for sample input
```

G. Debugging Tips for Extension Development

Symptom	Likely Cause	How to Diagnose	Fix
Human response never arrives	Notification callback not configured, WebSocket disconnected	Check `notification_callback` is called, verify WebSocket connection status	Implement fallback notification channels, add connection health checks
Composite tool runs in wrong order	Incorrect topological sort, cyclic dependencies	Visualize graph with `nx.draw()`, check for cycles with `nx.find_cycle()`	Validate graph is acyclic during construction, manually specify execution order if needed
Multi-modal tool consumes all memory	Large files loaded entirely into memory, no streaming	Monitor memory usage during execution, check if file is read all at once	Implement chunked processing, use disk-backed temporary storage
Tool sharing causes permission errors	JWT tokens expired, missing permission claims	Decode and inspect JWT token contents, check expiration time	Implement token refresh mechanism, add detailed permission claims
Learning module overfits to recent tasks	Experience replay buffer too small, no diversity in training	Analyze training data distribution, check buffer sampling strategy	Increase replay buffer size, implement prioritized experience replay
UI dashboard slows down with many agents	Too many real-time events, inefficient React re-renders	Check WebSocket message volume, profile React component renders	Implement client-side throttling, virtualize lists, batch updates

---

Summary

The extensibility of Project Archon's architecture is one of its core strengths. By adhering to clean interfaces, modular components, and well-defined data flows, the framework can accommodate a wide range of advanced capabilities beyond the initial five milestones. The extension ideas presented here—from human oversight to experiential learning—demonstrate how the foundation can evolve to address increasingly complex real-world applications of autonomous AI agents.

Each extension builds upon existing patterns: human interaction extends the `Tool` interface, composite tools leverage the `ToolExecutionEngine`, learning modules enhance the `MemorySystem`, and so on. This evolutionary approach ensures that advanced capabilities integrate seamlessly rather than requiring disruptive rewrites.

As with any extensible system, the key to successful expansion lies in maintaining the core architectural principles: clear separation of concerns, well-defined interfaces, and comprehensive observability. Future developers building upon this framework should prioritize these principles while exploring the exciting possibilities of autonomous AI systems.

11. Glossary

> **Milestone(s):** 1, 2, 3, 4, 5

The AI agent framework introduces specialized terminology from multiple domains: autonomous systems, large language models, software architecture, and cognitive science. This glossary provides clear definitions for the key technical terms, acronyms, and domain-specific vocabulary used throughout this design document. Each entry explains the concept in simple terms and references where it appears in the architecture.

Terms and Definitions

The following table lists all major terms in alphabetical order with their definitions and architectural context:

Term	Definition	Architectural Context
Action Parsing	The process of extracting a structured tool call (tool name and JSON arguments) from the LLM's free-form text output. Critical for connecting the agent's reasoning to executable actions.	Used in the ReAct Loop Engine (Milestone 2) to transform LLM thoughts into tool invocations.
ADR (Architecture Decision Record)	A structured document that captures an important architectural decision made along with its context, alternatives considered, rationale, and consequences. Provides transparency into the design process.	Used throughout the design document to explain key choices like JSON Schema validation, loop termination strategies, and context window management.
Adjacency List	A graph representation where each node is associated with a list of its neighboring nodes (dependencies). Efficient for representing sparse dependency graphs in task planning.	Used in the Planning component (Milestone 3) to represent task dependencies within the DAG structure.
Agent Core	The central coordinating component that orchestrates the planning, execution, and memory retrieval processes for a single agent. Maintains the agent's state and manages the flow between other components.	The main entry point in the High-Level Architecture; coordinates Tool System, Planner, Memory, and Orchestrator.
Agent-to-Agent Communication Medium	The underlying infrastructure that enables message passing between agents. Options include shared message buses, direct method calls, and blackboard architectures.	A key design decision in Multi-Agent Collaboration (Milestone 5) affecting how agents coordinate.
AI Executive	The primary mental model for the framework: an autonomous agent that plans, delegates to specialized tools, manages progress, and learns from experience—much like a human executive running a company.	The overarching analogy used throughout the design document to explain the framework's purpose and behavior.
Bag of Properties	A pattern using a dictionary (`Dict[str, Any]`) for extensible metadata storage, allowing components to attach arbitrary context without modifying core data structures.	Used in many data structures like `ToolResult.metadata` , `MemoryEntry.metadata` , and `context` fields throughout the framework.
Blackboard Architecture	An architecture pattern where multiple agents read and write to a common workspace (blackboard) containing shared problem data, enabling indirect coordination without direct message passing.	Considered as an alternative communication medium in Multi-Agent Collaboration (Milestone 5).
Broadcast Pattern	An orchestration pattern where messages are sent to all agents in the system (or a subset), enabling discovery and peer-to-peer coordination without a central coordinator.	One of the common orchestration patterns discussed in Multi-Agent Collaboration (Milestone 5).
Circuit Breaker Pattern	A design pattern that prevents repeated calls to a failing service by tracking failure rates and temporarily "opening the circuit" (blocking requests) to allow the service time to recover.	Used in the Tool Execution Engine to prevent cascading failures when tools repeatedly timeout or return errors.
Conflict Resolution	The process of reconciling contradictory results from multiple agents using defined merge strategies such as voting, confidence-weighted averaging, or hierarchical arbitration.	A key capability in Multi-Agent Collaboration (Milestone 5) for handling divergent outputs from specialized agents.
Context Window Management	The strategy for managing the limited token budget of LLMs by selectively including, truncating, or summarizing conversation history and retrieved memories.	Central to Memory & Context Management (Milestone 4); strategies include sliding window, recursive summarization, and importance scoring.
Conversation Memory	The linear, chronological record of dialogue turns (user messages, agent thoughts, tool results) for the current session. Analogous to an open notebook.	Part of the episodic memory layer in Memory & Context Management (Milestone 4); implemented as a buffer with token-aware truncation.
Correlation ID	A unique identifier passed through the execution flow (e.g., across tool calls, agent messages, and log entries) to enable tracing and debugging of a single request across distributed components.	Used throughout the framework for observability; appears in `Message.correlation_id` and structured logging.
Coverage Reports	Analysis showing which lines of code are executed during test runs, helping identify untested paths and ensure comprehensive testing of the framework.	Part of the Testing Strategy (Section 8) for maintaining code quality and preventing regressions.
DAG (Directed Acyclic Graph)	A graph structure with directed edges and no cycles, used to model task dependencies where each subtask depends on zero or more predecessors.	The core representation for task decomposition in Planning & Task Decomposition (Milestone 3).

Term	Definition	Architectural Context
DelegatedTask	A data structure representing a subtask assigned to a specific agent in a multi-agent system, tracking assignment, status, result, and dependencies.	Used by the `OrchestratorAgent` in Multi-Agent Collaboration (Milestone 5) to manage distributed work.
Delegation Loop	An infinite cycle where agents keep delegating the same task back and forth (e.g., Agent A delegates to B, who delegates back to A). A critical failure mode in multi-agent systems.	A common pitfall discussed in Multi-Agent Collaboration (Milestone 5); prevented by tracking delegation chains and timeouts.
Deterministic Tests	Tests that produce identical results on every run, without external dependencies or randomness, making them reliable for continuous integration.	Emphasized in the Testing Strategy (Section 8) for unit and integration tests of core components.
E2E Tests (End-to-End Tests)	Tests that run the complete system with minimal mocks to validate real task completion, often using lightweight tools and a mock LLM.	The top layer of the Testing Pyramid for Agent Systems; validates that all components work together correctly.
Episodic Memory	The linear sequence of dialogue turns (user inputs, agent responses, tool results) stored in chronological order, like an open notebook for the current session.	One of the three memory layers in Memory & Context Management (Milestone 4); implemented as a conversation buffer.
Error Context	A structured data container capturing details about an error: component, error type, severity, timestamp, details, recovery attempts, and correlation ID. Used by the recovery orchestrator.	Defined in the data model for systematic error handling; used by the `RecoveryOrchestrator` in Section 7.
Error Propagation Chain	The mechanism by which errors move through components (e.g., from tool execution to ReAct loop to planner) and trigger coordinated recovery actions at appropriate levels.	Described in Error Handling and Edge Cases (Section 7) as part of the graded response strategy.
Event Loop Debugging	Diagnosing issues in async Python code by examining the event loop, pending tasks, and asynchronous execution flow. Particularly important for multi-agent systems with concurrent messaging.	A debugging technique mentioned in the Debugging Guide (Section 9) for async-heavy components like the MessageBus.
Exponential Backoff with Jitter	A retry strategy where wait time doubles each attempt (exponential backoff) with random variation (jitter) to prevent synchronized retries from multiple clients.	Used in the Tool Execution Engine for retrying transient tool failures (Milestone 1).
Flight Control Center	A mental model for human-in-the-loop systems: humans act as mission control overseeing autonomous agents, intervening at critical decision points, much like NASA controllers monitoring a spacecraft.	Introduced in Future Extensions (Section 10) as an analogy for human oversight of autonomous agents.
Flaky Tests	Tests that sometimes pass and sometimes fail due to timing, randomness, or external dependencies, undermining confidence in the test suite.	A risk discussed in the Testing Strategy (Section 8); mitigated by using mock LLMs and deterministic tool simulations.
Flow Controller	A component that manages the end-to-end execution of a task, coordinating the agent, planner, memory, and tool registry while tracking the execution flow state.	Introduced in the data model as a high-level orchestrator for single-agent and multi-agent tasks.
Function Composition	A mental model for tool chaining: viewing tools as mathematical functions that can be composed into pipelines, where the output of one tool becomes the input to another.	Introduced in Future Extensions (Section 10) as an analogy for advanced tool scripting capabilities.
Functional Update	Creating a new instance of a data structure with modified fields instead of mutating the existing object, enabling immutable state management and easier debugging.	A core pattern used throughout the framework for updating `SubTask`, `Task`, and other stateful entities.
Graded Response Strategy	An error recovery approach that escalates gradually: first attempt automatic retry, then component-level adjustments, then system-level intervention, and finally human escalation.	The overarching error handling philosophy described in Section 7.
Hierarchical Pattern	An orchestration pattern with a single manager (orchestrator) and multiple workers in a tree structure, where the orchestrator decomposes tasks and delegates to specialized workers.	The primary pattern implemented in Multi-Agent Collaboration (Milestone 5) for its simplicity and clarity.
Human-in-the-Loop (HITL)	A system design where human oversight is integrated into autonomous processes, allowing intervention, approval, or guidance at critical decision points.	A future extension discussed in Section 10, enabled by interception rules and human interaction tools.

Term	Definition	Architectural Context
Hybrid Sliding Window with Summarization	A context management strategy that combines recent verbatim conversation turns with summarized older segments to maximize information retention within token limits.	An ADR option in Memory & Context Management (Milestone 4) that balances detail preservation with token efficiency.
Iteration Limit	The maximum number of ReAct cycles allowed before forced termination, preventing infinite loops when the agent cannot reach a final answer.	A critical safety mechanism in the ReAct Loop Engine (Milestone 2); configurable per task.
JSON-in-Text Parsing	The technique of extracting JSON objects embedded within free-form LLM output, typically using regex or robust JSON parsers that handle malformed text.	One of the action parsing strategies discussed in the ReAct Loop Engine (Milestone 2).
JSON Schema	A declarative language for annotating and validating JSON documents, used to define the expected parameters for tools with type constraints, descriptions, and required fields.	Chosen in an ADR for tool parameter validation (Milestone 1) due to its expressiveness and LLM familiarity.
LLM (Large Language Model)	A neural network trained on massive text corpora capable of generating human-like text, answering questions, and following instructions. The "brain" of the AI agent.	The foundational technology powering the framework's reasoning, planning, and decomposition capabilities.
Long-Term Memory	Semantic memory with vector search capabilities, storing past interactions and facts in an indexed format for retrieval by similarity, like an indexed filing cabinet.	One of the three memory layers in Memory & Context Management (Milestone 4); typically backed by a vector store.
Macro Recorder	A mental model for scripting: recording successful tool execution sequences as reusable templates that can be replayed with different parameters.	Introduced in Future Extensions (Section 10) as an analogy for learning and automating repetitive workflows.
Market-Based Pattern	An orchestration pattern using auction mechanisms where agents bid on tasks based on capability and cost, with the highest bidder winning the task.	One of the advanced orchestration patterns discussed in Multi-Agent Collaboration (Milestone 5).
Memory Retrieval	The process of fetching relevant past context (from episodic, working, and long-term memory) and injecting it into the LLM prompt to inform current reasoning.	A core function of the Memory & Context Management system (Milestone 4), using vector similarity and recency.
Message Bus	A publish-subscribe system for asynchronous communication between agents, where agents publish messages to topics and subscribe to receive messages from topics of interest.	The chosen communication medium in Multi-Agent Collaboration (Milestone 5); implemented as a topic-based routing system.
Mock LLM	A simulated LLM that returns predetermined responses for testing agent reasoning loops, enabling deterministic tests of ReAct cycles and planning.	A critical testing tool described in the Testing Strategy (Section 8); implemented as the `MockLLM` class.
OODA Loop (Observe-Orient-Decide-Act)	A military strategy cycle for rapid, adaptive decision-making: observe the situation, orient to context, decide on action, act, then repeat. Used as a mental model for the ReAct loop.	Introduced in the ReAct Loop Engine (Milestone 2) to explain the agent's continuous decision cycle.
Orchestrator Agent	A specialized coordinator agent that manages multiple worker agents, decomposes tasks, matches subtasks to agent roles, routes messages, and synthesizes results.	The central entity in hierarchical multi-agent systems (Milestone 5); extends `BaseAgent` with delegation logic.
ParsingError	A custom exception raised when the action parser fails to extract a valid tool call from the LLM's output, triggering a corrective observation in the ReAct loop.	Defined in the data model for structured error handling in the ReAct Loop Engine (Milestone 2).
Plan	A data structure representing the decomposed task as a DAG of subtasks, including the execution order derived from topological sorting of dependencies.	The output of the Planner component (Milestone 3) and input to the execution engine.
Professional Apprenticeship	A mental model for learning from experience: the agent learns from successes and mistakes over time, improving its planning and tool selection through reinforcement.	Introduced in Future Extensions (Section 10) as an analogy for adaptive improvement mechanisms.
Prompt Inspection	The debugging technique of analyzing the exact prompt sent to the LLM (including context, instructions, and history) to diagnose reasoning or tool selection issues.	A key debugging method in the Debugging Guide (Section 9); enabled by the `PromptInspector` utility.
ReAct Loop (Reasoning + Acting)	The core agent cycle: **Reason** (think about next step), **Act** (execute a tool or produce final answer), **Observe** (receive tool	The foundational execution engine implemented in Milestone 2; based on the ReAct paper.

Term	Definition	Architectural Context
	result), then repeat until task completion.	
Replanning Trigger Strategy	The policy determining when to regenerate a task plan—options include on every failure, when confidence drops below a threshold, or only on explicit user trigger.	An ADR topic in Planning & Task Decomposition (Milestone 3); affects resilience versus efficiency trade-offs.
Request-Scoped Context	Context that is associated with a specific request/flow and follows through async execution, enabling coherent state management across distributed components.	Used in the `FlowController` and `ExecutionFlow` to maintain task-specific state across async operations.
Role-Based Delegation	Task assignment strategy where subtasks are routed to agents based on their advertised capabilities and assigned roles (e.g., "Researcher", "Writer", "Coder").	The primary delegation mechanism in Multi-Agent Collaboration (Milestone 5); uses role matching for task routing.
Semantic Gap	The disconnect between the LLM's natural language reasoning and the structured world of software tools—bridged by the framework through parsing, schemas, and validation.	A core problem statement introduced in Section 1; the framework exists to bridge this gap systematically.
Service Locator	A design pattern used to encapsulate the processes involved in obtaining a service, such as the `ToolRegistry` providing tool lookup by name.	Used in the Tool System (Milestone 1) for dynamic tool discovery without tight coupling.
Shared Context	A common workspace (often a dictionary or blackboard) where agents working on the same task can access relevant information, intermediate outputs, and partial results.	A key enabler in Multi-Agent Collaboration (Milestone 5) to prevent agents from working in isolation.
Structured Logging	Logging in machine-readable format (typically JSON) with consistent schema, enabling automated analysis, correlation, and debugging of complex agent behaviors.	Recommended in the Debugging Guide (Section 9); implemented via `StructuredLogger` and `JSONFormatter`.
Subscription	A data structure representing an agent's interest in a message topic, containing a callback function and agent ID, used by the MessageBus for routing.	Part of the `MessageBus` implementation in Multi-Agent Collaboration (Milestone 5) for publish-subscribe messaging.
Swiss Army Knife with Specialized Attachments	A mental model for multi-modal tools: the base system provides core capabilities, while interchangeable attachments (tools) handle specific media types (images, audio, etc.).	Introduced in Future Extensions (Section 10) as an analogy for extensible tool systems.
Task Decomposition	The process of breaking a complex, high-level goal into a sequence or graph of concrete, actionable subtasks, each potentially requiring different tools or expertise.	The core function of the Planner component (Milestone 3), typically performed by an LLM with specific prompting.
Testable Seams	Interfaces where real components can be substituted with mocks during testing, such as the LLM interface, tool registry, or message bus.	A key principle in the Testing Strategy (Section 8) for isolating components and creating deterministic tests.
Testing Pyramid for Agent Systems	A three-layer testing strategy: **unit tests** for individual components (tools, parsers), **integration tests** for component interactions (ReAct loop with mock LLM), and **end-to-end tests** for full system validation.	The overarching testing philosophy described in Section 8, adapted for the unique challenges of AI agents.
Token	The basic unit of text processing for LLMs, roughly equivalent to a word or subword. Token limits constrain the amount of context (prompt + history) that can be sent to the LLM.	A fundamental constraint affecting memory management, context window design, and prompt engineering.
Token Budget Management	Enforcing context length limits through strategies like truncation, summarization, or selective retrieval to ensure the prompt stays within the LLM's token limit.	A core responsibility of the Memory & Context Management system (Milestone 4).
Tool	A well-defined interface that extends the agent's capabilities, consisting of a name, description, JSON schema for parameters, and an execute method that performs the actual operation.	The foundational abstraction in Milestone 1; all agent capabilities are exposed through tools.
Tool Chaining	The ability to connect multiple tools in a sequence where the output of one becomes the input to another, enabling complex workflows without requiring the LLM to orchestrate every step.	A future extension discussed in Section 10; implemented via `CompositeTool` and workflow graphs.

Term	Definition	Architectural Context
Tool Execution Engine	The component responsible for safely executing tool calls: validating parameters against schema, applying timeouts, handling errors, and formatting results for the LLM.	A critical part of the Tool System (Milestone 1); ensures reliability and safety of tool invocations.
Tool Registry	A central catalog that allows dynamic registration, lookup, and discovery of available tools, using the Service Locator pattern to decouple tool definition from usage.	The core of the Tool System (Milestone 1); enables runtime extensibility without agent restart.
Topic-Based Routing	Message routing where agents subscribe to topics (e.g., "research_requests", "writing_tasks") and receive messages published to those topics, enabling flexible communication patterns.	The routing strategy used by the `MessageBus` in Multi-Agent Collaboration (Milestone 5).
Topological Sort	A linear ordering of a DAG's vertices such that for every directed edge u → v, u comes before v in the ordering. Used to determine execution order for dependent subtasks.	The algorithm used by the Plan execution engine (Milestone 3) to schedule subtasks respecting dependencies.
Validation Result	The outcome of parameter validation against a JSON schema, including whether the input is valid and any error messages describing violations.	Used internally by the `ToolExecutionEngine._validate_arguments` method in Milestone 1.
Vector Embedding	A dense numerical representation of text (or other data) in a high-dimensional space, where semantically similar items have similar vectors, enabling similarity search.	The foundation of long-term memory retrieval in Memory & Context Management (Milestone 4).
Vector Store	A database optimized for storing vector embeddings and performing nearest-neighbor searches, used for semantic retrieval in long-term memory. Options include Chroma, FAISS, and Pinecone.	The storage backend for long-term memory in Milestone 4; the design supports pluggable implementations.
Verification Checkpoints	Concrete test commands and expected outputs that serve as objective completion criteria for each milestone, enabling learners to validate their implementation step-by-step.	Defined for each milestone in the Testing Strategy (Section 8) to ensure progressive validation.
Whole-Part Relationship	A composition pattern where the whole (e.g., `Task`) owns its parts (e.g., `SubTask`s), with lifecycle management and encapsulation of internal structure.	Used throughout the data model: `Task` contains `SubTask`s, `Agent` contains `Memory`, etc.
Work Breakdown Structure (WBS)	A hierarchical decomposition of a project into smaller components (tasks, subtasks), commonly used in project management. Used as an analogy for task decomposition.	The mental model for the Planner component (Milestone 3); helps explain hierarchical task decomposition.
Worker Agent	A specialized agent with a specific role and toolset that executes delegated subtasks, reporting results back to the orchestrator. Examples: Researcher, Writer, Coder.	The specialized agents in Multi-Agent Collaboration (Milestone 5); extend `BaseAgent` with role-specific tools.
Working Memory	Task-scoped key-value state that holds intermediate results, partial outputs, and contextual variables for the current task, like a whiteboard for active work.	One of the three memory layers in Memory & Context Management (Milestone 4); implemented as a mutable dictionary.

This glossary serves as a quick reference for understanding the specialized vocabulary of autonomous AI agent systems. When encountering an unfamiliar term while reading the design document, refer back to this table for clarification on its meaning and architectural role.