ALMA MATER STUDIORUM – UNIVERSITY OF BOLOGNA
CESENA CAMPUS

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
Master Degree in Computer Science and Engineering

AN EVENT-DRIVEN SOFTWARE ARCHITECTURE FOR SITUATED LLM AGENTS

Master Thesis in
SOFTWARE ARCHITECTURE AND PLATFORMS

Supervisor
Prof. ALESSANDRO RICCI

Co-supervisor
Prof. ANDREI CIORTEA

Candidate
LUCA TONELLI

Academic Year 2024 – 2025

A loving thought goes to my grandfather,
who passed away during these years
but whose warmth has always been with me

Abstract

Realizing the potential of **Agentic AI** requires agents to maintain a continuous, structural coupling with dynamic environments. Mainstream paradigms, however, often reduce this complex interaction to synchronous, stateless function calls, a simplification that proves inadequate for persistent, asynchronous domains.

This thesis addresses a fundamental engineering challenge: ***establishing the architectural prerequisites that enable agents based on Large Language Models (LLMs) to effectively operate tools as persistent, observable, and asynchronous entities, rather than as synchronous, stateless functions.***

To resolve this challenge, we introduce the **Apprentice framework**. True situatedness requires a departure from the synchronous function-calling abstraction. Accordingly, the framework introduces a model for **Enhanced Tools** conceptually rooted in the *Agents & Artifacts* (A&A) theory, characterized by three essential properties: **active Observability** of state, **Asynchrony** via event emission, and **Self-Description** through semantic manuals.

Our analysis demonstrates a critical impedance mismatch when integrating these tools with standard reasoning loops like ReAct: synchronous implementations suffer from inherent *State Blindness* (the inability to perceive spontaneous environmental changes) and *Execution Blocking* (halting the reasoning loop during long-running operations), rendering them structurally insufficient for artifact-based interaction. In response, the framework proposes a novel **Event-Driven Agent Architecture**. Central to this design is the **S-ORA (Situate-Observe-Reason-Act)**, a decision cycle tailored to decouple *Context Alignment* from *Reasoning*. Unlike traditional linear models, where the agent is forced to pause all reasoning while waiting for a tool to return a result, this architecture allows the system to suspend specific **Activities** into a semantic "waiting" state without halting the global agent runtime, efficiently reconciling active state inspection with passive event handling.

We validate this architectural baseline through two simulated scenarios: managing critical infrastructure with time-dependent processes and achieving implicit multi-agent coordination. The results demonstrate that the S-ORA runtime successfully handles strict temporal safety constraints and prevents infinite polling loops, which are capabilities structurally precluded by traditional synchronous paradigms. The current implementation remains a deliberate proof-of-concept, limited by volatile in-memory storage, an all-or-nothing event subscription model, and reliance on upstream orchestrators for

goal decomposition. Nevertheless, this research confirms that the shift towards situated agency necessitates a **dual evolution**: the adoption of a rigorous **Artifact-based design for tools**, and the consequent redesign of the agent's **temporal control flow** and memory management systems.

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 From Static LLMs to Autonomous, Situated Agents

The recent trajectory of Artificial Intelligence is defined by a radical expansion in the operational scope of Large Language Models (LLMs). Once viewed merely as static repositories of parametric knowledge, these models are evolving into the cognitive cores of autonomous agents capable of interacting with dynamic environments. This transition marks the shift from *Generative AI*, focused on media synthesis, to **Agentic AI**, a paradigm centering on the execution of goal-directed workflows.

Standard LLMs function fundamentally as probabilistic engines, trained to minimize a negative log-likelihood loss over vast text corpora. Formally, given a sequence of tokens $x = (x_1, \ldots, x_t)$, the model learns to maximize the probability of each token given its preceding context:

$$P(x) = \prod_{i=1}^{n} P(x_i \mid x_1, \ldots, x_{i-1}; \theta) \tag{1.1}$$

While this mechanism enables impressive few-shot reasoning, the resulting system operates under a fundamental constraint: its knowledge is frozen at the training cut-off, and its interaction with the world is restricted to a single, stateless inference pass. Without direct access to external state, it cannot track evolving domain conditions beyond what is explicitly provided in its input context, often resulting in outdated or hallucinatory responses.

The concept of agency in AI has deep theoretical roots. Wooldridge and Jennings [22] provide one of the most influential foundational definitions: an agent is a computer system situated in some environment, capable of perceiving it and responding to changes in a timely fashion, while acting autonomously in

pursuit of its own goals. This establishes the fundamental **perception-action cycle** that characterizes intelligent behavior: agents continuously observe their environment, process these observations, and execute actions that modify the world state. Wooldridge and Jennings further distinguish between mere reactive systems and truly *rational agents*, those that do not simply respond to stimuli but proactively take initiative, selecting actions on the basis of their internal state and their knowledge of the world.

Traditional agent implementations typically required handcrafted rules or reinforcement learning, making generalization to novel environments challenging. The emergence of foundation models has reshaped this landscape, giving rise to a new class of AI systems known as **language agents**.[1] Language agents apply the latest advances in LLMs to the existing field of agent design, connecting them to internal memory and external environments to ground their reasoning in existing knowledge or live observations. By leveraging the commonsense priors embedded in LLMs, these systems can adapt to novel tasks without relying on extensive human annotation or trial-and-error learning. While the earliest agents used LLMs to directly select actions, more recent systems additionally use them to reason, plan, and manage long-term memory to improve decision-making. This has led to architectures that equip the LLM with explicit memory modules, a structured action space covering both external environment interactions and internal cognitive operations, and an iterative decision-making loop that interleaves reasoning and planning before committing to an execution step.

Reasoning frameworks emerged to drive this control loop, allowing agents to verify facts outside their parametric knowledge and effectively operationalizing the classical perception-action cycle for language models. Yet simply invoking an external action is only the first step. Achieving true **Situated Agentic AI** requires more than a language model equipped with tools; it demands *situatedness*. In classical multi-agent literature, being situated means maintaining a continuous temporal alignment with the world: perception must act as an ongoing, asynchronous stream of events, and the agent must be capable of reacting to these dynamics promptly. Without this temporal coherence, the agent inevitably falls out of sync, making decisions based on obsolete environmental states.

Current language agents fail to establish this persistent, structural coupling. Because their interaction is mediated almost entirely by synchronous function calls, perception is not treated as an independent background process, but merely as the immediate, static return value of a specific query. This

[1] In the remainder of this work, we use the term *language agent* to specifically denote autonomous systems that utilize an LLM as their core computational unit, distinguishing them from classical or theoretical agent formulations.

paradigm forces the agent into a rigid cycle where reasoning is blocked during execution, rendering it blind to spontaneous changes. Consequently, the agent cannot respond to asynchronous stimuli as they occur, remaining conceptually detached from the real-time flow of the environment.

1.2 The Concept of Tools: Language Agents vs. Agents

Although the architecture of the language agent's cognitive core (LLM, Planning, and Memory) is now well-established, the definition of its **Tool** interface remains unclear. As surveyed in recent literature [9], a tool in LLM-based systems is treated primarily as a synchronous API call, a deterministic mapping from an input vector to an output result.

From an architectural standpoint, this reduces the tool to a **stateless, ephemeral service**. Even if the underlying service persists externally (e.g., on a remote server), it exists within the language agent's cognitive space only during the execution interval ($t_{call} \rightarrow t_{return}$). Once the system injects the return value into the context window, the tool effectively ceases to exist from the agent's perspective, leaving no persistent state, observable channel, or event stream for the agent to monitor. This model forces the language agent to rely entirely on its internal memory to track the state of the world, treating the environment as a passive recipient of commands rather than an interactive space.

In classical multi-agent systems literature, the environment is not merely a passive message-passing medium but a first-class abstraction [21]. The Agents & Artifacts (A&A) meta-model [11] builds upon this perspective, elevating the environment to a space populated by **Artifacts**.

Unlike the stateless functions of current language agents, an Artifact is a computational object designed for agent exploitation. As Ricci et al. detail [12], Artifacts operate as "*resources and tools*" designed to support agent activities, characterized by *Observability*, *Controllability*, and *Persistence*.

Comparing these paradigms reveals a critical **Integration Gap**. While state-of-the-art language agents possess sophisticated internal reasoning capabilities, they interact with the world through the "keyhole" of stateless function calls.

This mismatch creates a structural inefficiency. Attempting to manage stateful environments using synchronous primitives forces complexity back into the prompt engineering layer, compelling the language agent to memorize what the tool fails to store. To bridge this gap, we must evolve the tool concept from a simple function to a **Tool Meta-Model** that incorporates the semantic

richness of the A&A Artifact. This evolution is the prerequisite for enabling a true structural coupling between the language agent and its environment.

1.3 Problem Statement

Failing to bridge the gap between the agent's cognitive needs and its operational interface results in severe functional limitations. Adhering to the stateless "Tool-as-Function" paradigm in persistent environments engenders three structural limitations: *State Blindness*, the *Blocking Execution* problem, and the consequent *Context Bottleneck*.

The most immediate manifestation is the decoupling of the agent from the environment's timeline. Since standard architectures perceive reality only through the narrow window of a function return, the agent remains effectively **blind** to spontaneous environmental changes, reasoning on outdated "snapshots" rather than live streams. Strictly coupled with this is the rigid, synchronous nature of function calls forces the agent to halt its reasoning during execution, preventing it from handling long-running processes or reacting to asynchronous interrupts.

These limitations force a structural inefficiency. To compensate for the tool's lack of persistence and observability, the architecture must offload the entire environmental state into the prompt. This creates a **Context Bottleneck**, where the agent's reasoning capacity is saturated not by the complexity of the task itself, but by the sheer volume of static context required to describe a dynamic world.

These three structural limitations are formalized in Section 2.4 as, respectively, the *Statelessness-Persistence conflict* (Section 2.4.1), the *Context Window Saturation* (Section 2.4.2), and the *Synchronous Execution Mismatch* (Section 2.4.3).

1.4 Thesis Proposal

This thesis proposes the **Apprentice** framework, a dual solution designed to bridge the integration gap: redesigning the tools, and redesigning the language agents that use them.

First, we formalize a **Tool Meta-Model** that transforms tools from stateless synchronous functions into persistent, observable entities. Drawing from A&A theory, this model equips tools with self-description (semantic manuals for discovery), observability (exposed state that can be monitored), and asynchrony (event emission independent of agent queries).

Second, we introduce the **S-ORA (Situate-Observe-Reason-Act) architecture**, an event-driven agent engine that can actually exploit these properties. Unlike traditional linear reasoning loops, S-ORA decouples context updates from deliberation, allowing the agent to suspend activities without freezing the entire system.

The result is a validated architecture baseline for situated agency: one where agents do not just call functions, but inhabit environments.

Organization

Chapter 2: Background & Gap Analysis. Surveys the state of the art in language agents, contrasting the limits of current synchronous paradigms with the theoretical foundations of the A&A model and emerging frameworks.

Chapter 3: The Apprentice Framework. Formalizes the conceptual conflict between stateless tools and situated requirements, defining the overarching theoretical specifications for the **Apprentice** framework and its Enhanced Tools.

Chapter 4: The Enhanced Tool Architecture. Defines the technical standard for the Enhanced Tool, specifying the protocols for discovery, subscription, and the asynchronous event stream. It introduces the Semantic Manual Schema as the formal cognitive interface between the environment and the agent.

Chapter 5: The S-ORA Agent Architecture. Details the engineering of the S-ORA Engine, describing the transition to an event-driven architecture, the Activity-based memory and concurrency model, and the formal steps of the S-ORA Decision Cycle.

Chapters 6 & 7: Implementations. Describes the concrete realization of the Enhanced Tool and the S-ORA agent.

Chapters 8 & 9: Validation & Conclusion. Evaluates the architectural baseline through comparative testing and summarizes the contributions towards a standardized design for situated agency.

Chapter 2

Background

This work is situated at the intersection of two distinct research lineages: the classical theory of Multi-Agent Systems (MAS) and the emerging paradigm of Large Language Model (LLM) based agents. While Generative AI has endowed autonomous systems with flexible reasoning capabilities, it has largely neglected the structural formalization of the *environment*. Current engineering practices tend to reduce the external world to a passive context window, ignoring the rich ontological definitions provided by classical MAS literature.

Therefore, to ground the architectural proposal presented in later chapters, this section critically examines the theoretical and technological foundations of situated agency.

2.1 Tool Use, Learning, and Connectivity

A language model operating purely on parametric knowledge cannot track live information, execute code, or affect external systems. The integration of external tools and reasoning skills into these models has given rise to the paradigm of Augmented Language Models [10], significantly expanding their capabilities beyond standard text generation. This section traces how the field has approached this evolution, from ad-hoc function schemas to self-supervised tool learning and, more recently, to standardized interoperability protocols.

2.1.1 Defining Tools in the Agentic Era

The term "tool" appears throughout the literature without much agreement on what it actually means. Wang et al. [19] offer one of the more systematic attempts at a definition, treating tools as external programs invoked by the LM to go beyond text generation. What distinguishes them from passive context is that they execute. Wang et al. group their functions into three categories:

perception, which allows the model to retrieve live or external information it lacks; *computation*, which offloads intensive mathematical or symbolic reasoning the model struggles with; and *action*, which issues commands to modify physical or digital systems outside the model's generative scope. In practice many tools span more than one category. A search engine, for instance, both retrieves web content (perception) and ranks it (computation).

2.1.2 The Current Paradigm: Stateless Function Calling

In the standard setup, the agent receives a JSON schema describing a tool's name, description, and parameters directly in the system prompt. When the model decides an action is needed, it halts generation and outputs a structured JSON object; the execution environment runs the corresponding function and appends the result to the conversation history.

This works well enough for isolated queries. The problems emerge at scale. A model given only a syntactic schema has no procedural knowledge: it does not know which edge cases to avoid, how to sequence calls across multiple tools, or when not to call a tool at all. Injecting dozens of schemas into a single prompt compounds the issue, degrading reasoning quality and driving up inference costs [19].

2.1.3 Tool Learning and the Emergence of Agent Skills

One response to the brittleness of zero-shot function calling is to have models learn tool use rather than merely follow schemas. *Toolformer* [14] is the clearest example: it trains a model to decide autonomously when and how to call external APIs, without any human-labeled examples of correct usage.

The core mechanism relies on a self-supervised filtering step. Assuming an API call is introduced at index i within a text sequence, we can define $L_i(z)$ as the weighted cross-entropy loss calculated for the subsequent tokens (x_i through x_n), conditioned on the context prefix z:

$$L_i(z) = -\sum_{j=i}^{n} w_{j-i} \cdot \log p_M(x_j \mid z,\, x_{1:j-1}) \tag{2.1}$$

A call c_i is retained only if providing the model with both the call and its result r_i reduces the loss more than providing neither, formally $L_i^- - L_i^+ \geq \tau_f$. This way the model retains only those invocations that genuinely help it predict what comes next, with no task-specific supervision required [14].

The paradigm has since extended to *tool making*, where agents generate new scripts to handle tasks for which no tool yet exists [19]. Fine-tuning

for every possible tool remains computationally intractable, however, and the industry has converged on a lighter-weight alternative: **agent skills** [1]. A skill is a self-contained directory built around a `SKILL.md` file that combines YAML metadata, describing what the skill does and when it applies, with Markdown instructions for how to execute it. Auxiliary scripts, reference documents, and templates live in subdirectories alongside it.

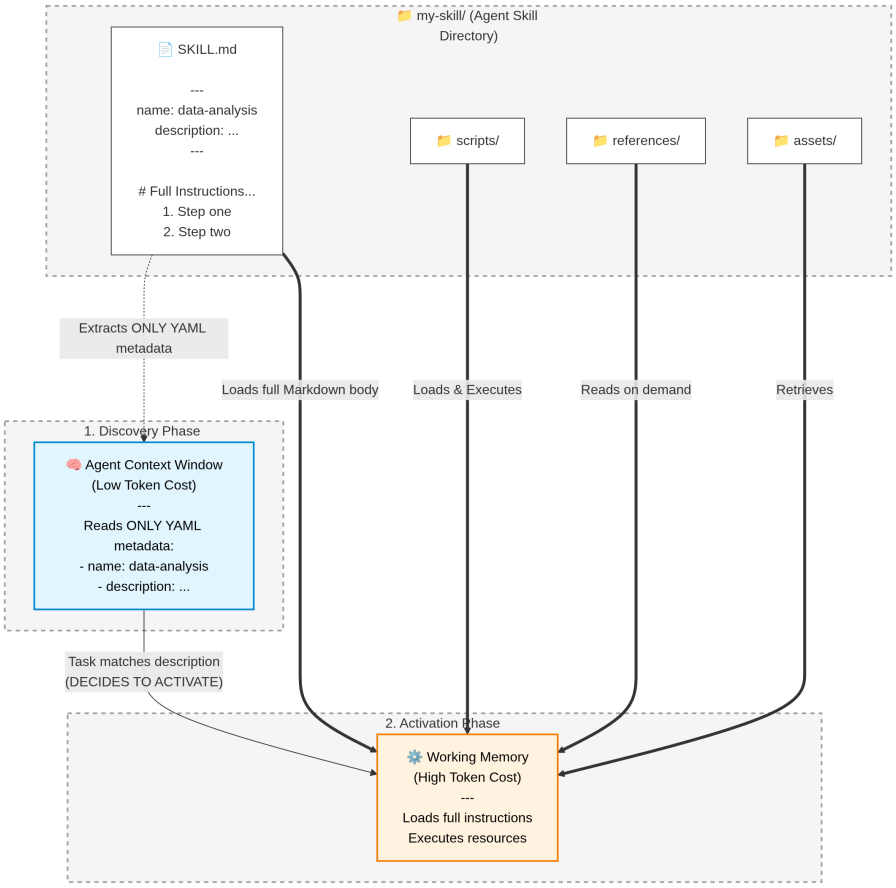

Figure 2.1: The modular structure of an agent skill and its progressive disclosure mechanism. Metadata is loaded during discovery; full procedural instructions are retrieved only on activation.

The practical advantage is selective loading, implemented through a strategy known as *progressive disclosure*. During an initial discovery pass, the system injects only the concise YAML metadata into the prompt, keeping the model aware of available capabilities without exhausting the context window. When a request matches a skill's description, the orchestrator retrieves the full instructional content and loads it into working memory. The model then

executes the prescribed steps, calling external scripts or accessing auxiliary documents as needed. Filesystem-native agents navigate these directories via shell commands; tool-constrained agents rely on predefined interfaces to unpack the bundled assets [1].

A recent analysis of over 40,000 publicly listed skills [7] found the ecosystem heavily concentrated around software engineering tasks. An evaluation of skill sizes reveals an extremely skewed distribution: while the vast majority of these modules are compact enough to respect standard context limits (displaying a median size of 1,414 tokens), significant outliers exist. The top 1% surpass the 9,253-token mark, peaking at 116,239. This inflation is primarily caused by developers embedding comprehensive documentation, extensive code snippets, and templating structures directly within individual files. This makes dynamic loading not an optimization but a requirement. The same study raised a safety concern worth noting: almost 40% of the available modules have the potential to execute critical system modifications or alter environmental states [7], which means sandboxing and human-in-the-loop confirmation cannot be treated as optional.

2.1.4 Interoperability Protocols: The Model Context Protocol (MCP)

As the number of tools and providers grows, maintaining a custom integration for each one becomes unworkable. The **Model Context Protocol (MCP)** [5, 2] addresses this by defining a shared communication standard between reasoning engines and external infrastructure, replacing one-off adapters with a single negotiated protocol.

The architecture is three-tiered, establishing a clear boundary between the agent's cognitive core and the external environment. A user-facing *Host* application (which embeds the language agent and its LLM) manages one or more *Clients*. These clients connect to dedicated *Servers* (which host the actual tools, data sources, and APIs) through a standardized transport. Communication happens locally via STDIO or remotely via Streamable HTTP, which replaced the earlier HTTP+SSE approach in the March 2025 specification revision [5, 2].

Recent work has mapped the operational lifecycle of an MCP server into four phases (creation, deployment, operation, and maintenance), each introducing distinct security exposure points [5]. A threat class receiving particular attention is *parasitic toolchain attacks*, where malicious content returned by an external tool is passed silently into the model's context and used to redirect its behavior toward unauthorized actions such as credential exfiltration.

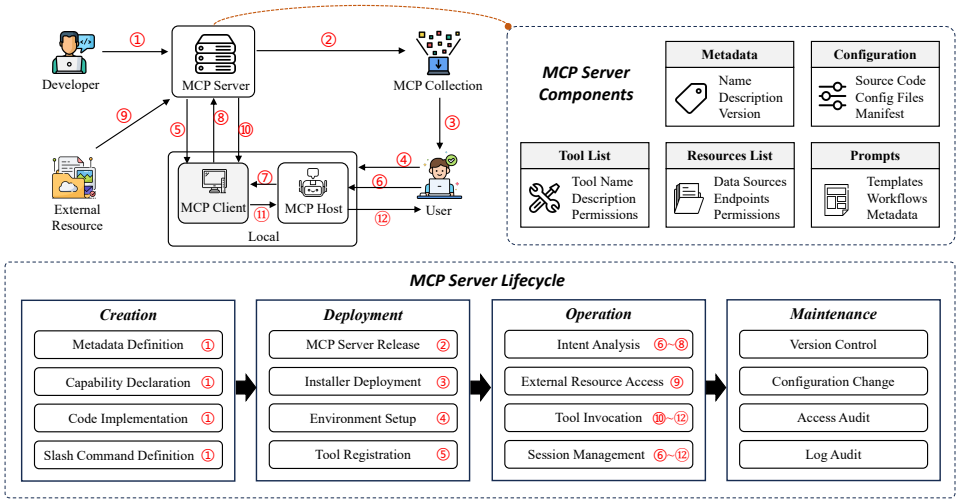

Figure 2.2: MCP server components and operational lifecycle. The upper part shows the interaction flow among developer, server, host, client, and external resources; the lower part summarizes the four lifecycle phases (creation, deployment, operation, maintenance) with their key activities. Source: [5], Fig. 3.

The protocol defines two sets of primitives. On the server side, *tools* are model-controlled functions the agent can invoke to change external state; *resources* are application-controlled passive stores (file URIs and similar) that provide verifiable context and help anchor the model against hallucinations; and *prompts* are user-controlled templates for orchestrating multi-step workflows. On the client side, *sampling* lets an external server query the host model for intermediate inferences, shifting AI access credentials entirely onto the client; *elicitation* allows a server to pause its processing and request missing parameters or explicit confirmations directly from the user; and *roots* lets the client broadcast filesystem boundaries the server should not cross.

2.2 Cognitive Architectures for Language Agents

The emergence of capable LLMs has not automatically solved the problem of building *autonomous agents*. A language model is, by design, stateless: it retains no information across independent inference calls and cannot, on its own, sustain the multi-step reasoning and persistent memory required for long-horizon tasks. **Cognitive Architectures for Language Agents (CoALA)** [16], published in TMLR (2024), addresses this gap by importing the structural principles of classical cognitive architectures, most notably Soar [6], into the design of LLM-based agents. The central argument is that just as production systems required an explicit control architecture to become capable agents,

so do LLMs. CoALA characterises a language agent along three orthogonal dimensions: its *memory organisation*, its *action space*, and its *decision-making procedure*. The LLM functions as the core computational unit, while the surrounding architecture compensates for its inherent limitations: bounded context, lack of persistence, and myopic autoregressive generation.

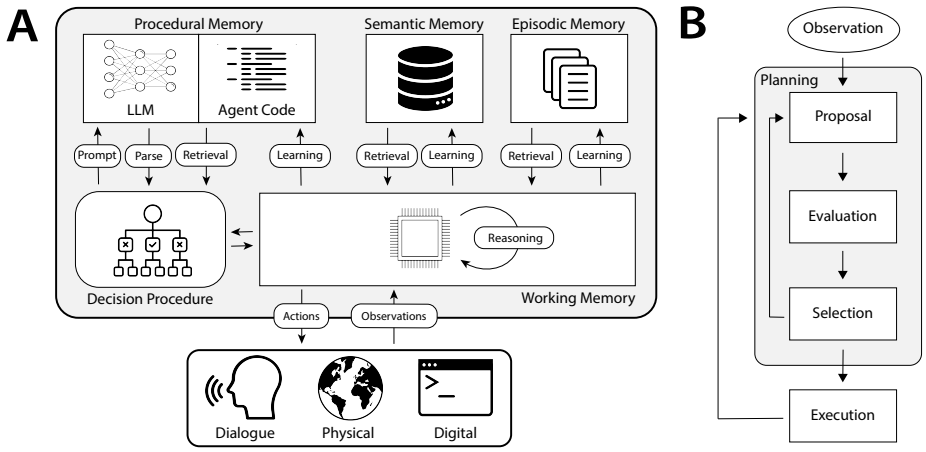

Figure 2.3: The CoALA framework. **(A)** Modular components: the decision procedure orchestrates interactions between the LLM, the three long-term memory partitions, and the external environment. **(B)** The iterative decision cycle: at each step, the agent uses retrieval and reasoning to plan (propose → evaluate → select), executes the winning action, observes the outcome, and cycles again. Source: [16], Fig. 4.

2.2.1 Memory Organisation

CoALA partitions an agent's information storage into two levels. **Working Memory (WM)** corresponds to the active LLM context window: it holds current observations, intermediate reasoning traces, and active goals. Its capacity is strictly bounded by the architectural limits of the underlying language model, making selective population of the context a critical design concern.

Long-Term Memory (LTM) is an externalised, persistent store subdivided into three psychologically-grounded partitions: *Episodic Memory* ($\mathcal{E}$), a chronological record of past agent trajectories queryable by recency, importance, and semantic similarity; *Semantic Memory* ($\mathcal{S}$), a repository of factual world knowledge (such as documents, tool schemas, and environment models) that the agent retrieves on demand rather than keeping resident in context; and *Procedural Memory* ($\mathcal{P}$), knowledge of *how* to act, existing in two sub-forms: *implicit* (the LLM's parametric weights) and *explicit* (agent source code, verified tool-call sequences, compiled skill libraries). Writes to procedural memory

carry the highest risk, as an erroneous update can silently corrupt the agent's behaviour across all future decision cycles [16].

2.2.2 Action Space

The full action repertoire is formalised as:

$$\mathcal{A} = \mathcal{A}_{\text{int}} \cup \mathcal{A}_{\text{ext}} \tag{2.2}$$

External actions ($\mathcal{A}_{\text{ext}}$), collectively termed *grounding*, direct the agent's outputs towards the environment, including API calls, web navigation, robotic actuation, or dialogue. They alter world state but yield no cognitive update until the resulting observation is fed back into WM.

Internal actions ($\mathcal{A}_{\text{int}}$) target the agent's own memory structures and represent the key departure from passive Retrieval-Augmented Generation (RAG). CoALA distinguishes three classes: *reasoning* (read/write within WM via LLM inference), *retrieval* (read from LTM into WM on demand), and *learning* (write from WM into LTM). The architectural breakthrough here is conceptualising *learning* as an explicit operational choice rather than a passive background routine. This grants the agent the autonomy to actively filter information and decide the optimal moments for knowledge consolidation. This mirrors biological memory consolidation and stands in sharp contrast to standard RL agents that apply a fixed update rule at every step.

2.2.3 Decision-Making Procedure

CoALA structures the agent's control flow as a repeating **decision cycle** comprising a *planning* stage followed by an *execution* stage, as depicted in Figure 2.3.

During **planning**, the agent interleaves reasoning and retrieval across three sub-stages. In the *proposal* phase, the LLM, conditioned on the current WM state s_t, generates a candidate set $A_c \subset \mathcal{A}$. During *evaluation*, the system assesses the viability of each option. This scoring mechanism can rely on hard-coded rules, parametric reward models, or forward-looking projections generated by the LLM itself, which temporarily serves as a transition model to anticipate future states. In the *selection* phase, the highest-scoring candidate a^* is chosen; if no candidate meets a quality threshold the loop returns to proposal, enabling iterative refinement.

During **execution**, a^* is dispatched: if $a^* \in \mathcal{A}_{\text{int}}$, the relevant memory partition is updated; if $a^* \in \mathcal{A}_{\text{ext}}$, the action is issued to the environment, the resulting observation is appended to WM, and a new cycle begins. Taken together, the CoALA framework provides a unifying language for comparing

existing agent designs and a normative blueprint for developing new ones. By explicitly separating memory management, action formulation, and decision-making into independently configurable modules, the framework formalises the theoretical transition from simple language models to robust, stateful cognitive agents.

2.2.4 Reasoning and Acting Paradigms

The evolution of Large Language Models (LLMs) as autonomous agents has historically proceeded along two orthogonal trajectories: internal cognitive processing and external environmental execution. Early paradigms treated these capabilities as mutually exclusive, leading to structural limitations in tasks requiring both complex logic and real-world grounding.

Pure reasoning approaches, most notably Chain-of-Thought (CoT) prompting [20], significantly enhance complex problem-solving by exposing intermediate inferential steps. However, because they are constrained entirely to the model's static parametric memory, these closed-loop systems are inherently vulnerable to factual hallucinations and cascading deductive failures. The model cannot verify its internal assumptions against external ground truth. Conversely, execution-oriented paradigms (Act-only) allow agents to query APIs and databases, yet they fundamentally lack the explicit working memory necessary to maintain long-horizon goals or dynamically re-evaluate strategies upon encountering execution errors.

The ReAct Framework

To reconcile this dichotomy, Yao et al. [25] introduced the **ReAct** (Reasoning and Acting) framework, a standardized paradigm that embeds semantic reasoning traces directly into the action execution loop.

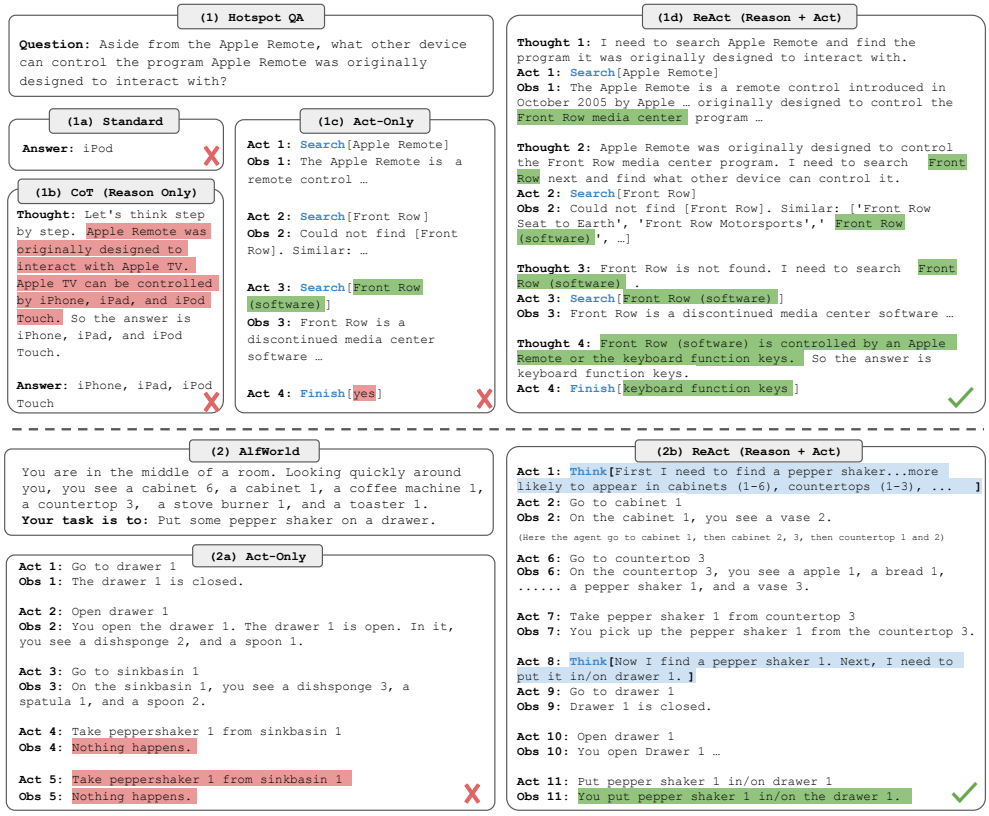

Figure 2.4: Comparison of prompting paradigms for language agents. While Chain-of-Thought (CoT) suffers from hallucination due to a lack of external grounding, and Act-only approaches struggle with myopic planning, ReAct creates a synergistic loop by interleaving internal reasoning (*Thought*) with external execution (*Action*) and environmental feedback (*Observation*). Source: [25], Fig. 1

In standard formalisations, an interactive agent perceives its surroundings at a given time step by registering an observation $o_t \in \mathcal{O}$, which prompts the execution of an environmental action $a_t \in \mathcal{A}$. This decision is strictly conditioned on the accumulated historical sequence $c_t = (o_1, a_1, \ldots, o_{t-1}, a_{t-1}, o_t)$. The ReAct architecture breaks this structural limitation by expanding the permissible operational space to include a latent linguistic dimension $\mathcal{L}$. During

any discrete time step t, the agent's policy π may output either an external, state-altering action $a_t \in \mathcal{A}$ or an internal, state-evaluating thought $t_t \in \mathcal{L}$.

Following [25], this expands the standard context into an augmented, interleaved sequence:

$$\hat{\tau}_t = (o_1, t_1, a_1, \ldots, o_{t-1}, t_{t-1}, a_{t-1}, o_t) \tag{2.3}$$

This structural interleaving serves a critical dual purpose in agent architecture:

- **Cognitive State Tracking (Thoughts):** Internal reasoning traces (t_t) act as a dynamic state tracker. They allow the model to synthesize past observations, formulate immediate sub-goals, and perform error recovery. For instance, by generating a thought such as "The previous search returned no results, I need to try a broader query," the agent explicitly modifies its execution plan before committing to a new action.
- **Factual Grounding (Actions):** External actions (a_t) serve to ground the agent's internal logic against verifiable realities. By extracting factual claims from a Wikipedia endpoint or an external database, the agent breaks the auto-regressive hallucination loop that plagues standard LLMs.

Extensive empirical evaluations validate this architectural synergy [25]. In knowledge-intensive benchmarks such as HotpotQA [23] and FEVER [17], ReAct effectively leverages external search capabilities to bypass the factual blind spots of standard CoT, significantly reducing hallucination rates. Similarly, in multi-step decision-making environments like ALFWorld [15] and WebShop [24], the capacity to explicitly reason about task progress allows ReAct agents to recover from localized failures and adapt to unexpected environment responses, drastically outperforming myopic Act-only baselines.

Ultimately, ReAct establishes that treating reasoning and acting as mutually reinforcing processes, rather than orthogonal capabilities, is an architectural prerequisite for developing robust, grounded language agents.

2.3 Environments in Multi-Agent Systems: The Agents & Artifacts (A&A) Model

For much of its early history, research in Distributed Artificial Intelligence treated the environment as little more than a communication medium, acting as a transparent channel through which agents exchanged messages. This *communication vacuum* perspective assumes that all meaningful computation

resides within agents themselves; consequently, it reduces the environment to a passive container.

Recent work in *Environment Programming* [13] challenges this view, demonstrating that deliberately structuring the environment can significantly enhance system modularity and agent coordination. Rather than forcing agents to manage all coordination logic internally, we can offload much of this complexity into a well-designed environment layer.

The **Agents & Artifacts (A&A)** meta-model [11] formalizes this idea by treating the environment as a first-class design concern. In this framework, a MAS comprises not just agents, but two complementary types of entities: *Agents* and *Artifacts*.

Drawing from Activity Theory [11], the A&A model establishes a clean separation between the active and passive elements of a system:

- **Agents** embody the system's autonomy. They pursue goals (Goals $\rightarrow$ Actions) and make decisions about *what* to do and *why*.
- **Artifacts** are non-autonomous resources designed to be used. They encapsulate functionality (Input $\rightarrow$ Output/StateChange) and exist purely to support agents' work, effectively representing the *how*.

This separation yields a practical benefit known as **Cognitive Offloading**. When we embed coordination protocols, shared state, and operational mechanisms directly into artifacts, we free agents to focus on high-level decision-making. The agent decides; the artifact executes.

2.3.1 Artifact Properties

To move beyond vague notions of "tools", Ricci et al. [13] provide a formal definition of artifacts as tuples:

$$\mathcal{A} = \langle \mathcal{I}_{use}, \Sigma, \mathcal{P}_{obs}, \mathcal{M} \rangle \tag{2.4}$$

where $\mathcal{I}_{use}$ denotes the usage interface, Σ the internal state, $\mathcal{P}_{obs}$ the observable properties, and $\mathcal{M}$ the manual.

Usage Interface

The usage interface $\mathcal{I}_{use}$ defines operations that agents can invoke. Unlike the stateless function calls typical of current LLM tool-use patterns, operations in the A&A model induce state transitions:

$$\delta : \Sigma \times \mathrm{Op} \rightarrow \Sigma'$$

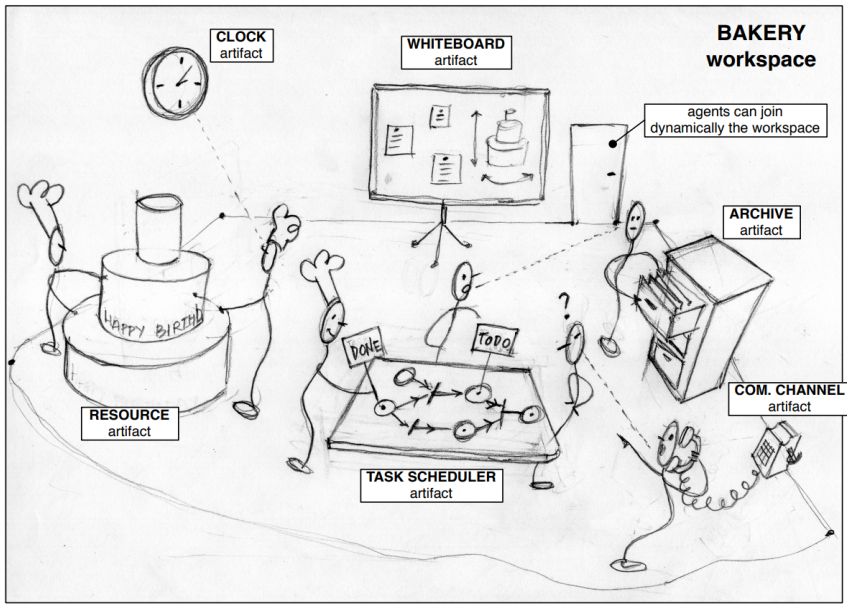

Figure 2.5: **Cognitive Offloading in the A&A Model.** An intuitive representation of a Multi-Agent System where agents (e.g., cooks) coordinate not only through direct communication but by exploiting the observable properties and signals of shared environmental artifacts (e.g., task schedulers, ovens). Source: [3], Fig. 5.1

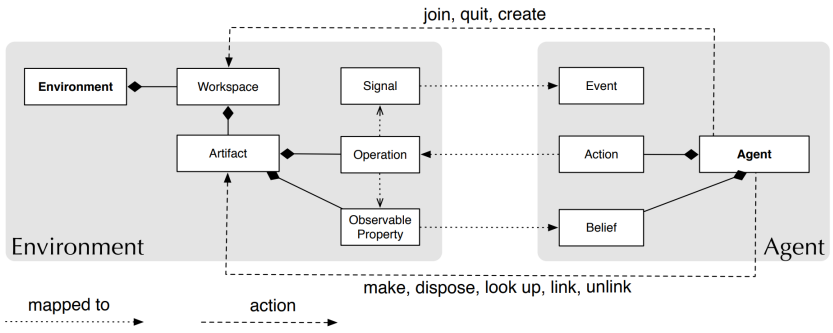

Figure 2.6: **Agent-Artifact Interaction Dynamics.** The diagram formalizes the interaction loop: agents act upon the environment by triggering artifact *operations* (via the usage interface I_{use}), while the artifact updates the agents' beliefs through *observable properties* (P_{obs}) and active *signals*. Source: [3], Fig. 5.3

This architectural choice enables the modeling of realistic, long-running processes. The execution model [13] decouples operation *invocation* from *completion*. This implies that an operation is not merely a request for a return value but a process that may persist in time, allowing the agent to proceed with other cognitive tasks while the artifact processes the request in the background.

Observable Properties

Unlike traditional object-oriented design that emphasizes encapsulation, artifacts prioritize *coordination transparency*. The set $\mathcal{P}_{obs} \subset \Sigma$ represents state variables explicitly exposed to the environment, eliminating the need for constant polling.

This supports two interaction modes:

- **Passive Inspection**: Agents can query $\mathcal{P}_{obs}$ to gather context without altering the system state.
- **Active Notification**: When a property $p \in \mathcal{P}_{obs}$ changes, the artifact triggers an *observable event*. In architectures such as JaCaMo [4], agents must explicitly *focus* on an artifact to subscribe to these updates. This mechanism filters irrelevant information, ensuring agents are notified only of pertinent state changes.

Additionally, artifacts can emit **signals**. Distinct from property updates, which reflect persistent state changes, signals convey volatile information (e.g., temporary alerts or synchronization ticks) without permanently modifying the artifact's state variables.

The Manual

The A&A model acknowledges that in open systems, agents cannot know every tool in advance. Therefore, artifacts must provide a machine-readable **Manual** $\mathcal{M}$ (containing the *Operating Instructions* [11]). This component includes:

1. A **Function Description**, defining the artifact's purpose.
2. **Operating Instructions**, detailing usage protocols and constraints (e.g., `init()` before `start()`).

In the context of LLM-based agents, we can reinterpret this requirement as a precursor to *in-context learning*: the manual acts as a runtime prompt that configures agent behavior, connecting the tool's implementation to the agent's dynamic reasoning.

2.4 The Modeling Gap: When Language Models Meet Dynamic Environments

The preceding sections outline a fundamental paradox in modern Agentic AI. On one side, cognitive frameworks like ReAct [25] and CoALA [16] endow language models with sophisticated internal reasoning and memory management. On the other side, the Agents & Artifacts (A&A) model [11] provides a robust ontology for structuring stateful, observable environments. However, the current engineering consensus attempts to bridge these two paradigms using the primitive mechanism of stateless function calling.

Reducing persistent environmental artifacts to ephemeral API endpoints introduces three structural limitations that jointly preclude situated agency: the absence of persistent state awareness, progressive saturation of the context window, and a fundamental mismatch between synchronous reasoning and asynchronous execution.

It is worth noting that even MCP [5, 2], the most sophisticated interoperability standard currently available, does not resolve these limitations. While MCP standardises the *communication channel* and introduces basic observability for passive data (Resources), it does not alter the fundamental nature of the active endpoints: an execution *Tool* in MCP remains a stateless, synchronous function invocation. The server returns a value and the execution thread is effectively closed. At the tool level, there is no native concept of persistent observable state, no asynchronous event stream for long-running operations, and no semantic manual encoding procedural constraints. MCP therefore solves the *connectivity* problem while leaving the *semantic and temporal mismatch* between agent execution and environmental dynamics entirely unaddressed.

2.4.1 Statelessness versus Persistence

The primary limitation of standard function calling lies in its treatment of time and state. As defined in Section 2.1.2, a standard LLM tool invocation is strictly ephemeral. It exists in the agent's cognitive trajectory only during the execution interval between invocation and return ($t_{call} \rightarrow t_{return}$). Once the result is appended to the context window, the environment is effectively frozen in the agent's memory.

This stateless consumption sharply contradicts the definition of an A&A Artifact, which is defined by an internal, continuous state Σ subject to autonomous transitions $\delta : \Sigma \times \text{Op} \rightarrow \Sigma'$. Because the LLM only perceives the environment through discrete, agent-initiated function returns, it suffers from a fundamental lack of state awareness. If an artifact's state changes sponta-

neously (due to external processes, other agents, or temporal decay), the agent remains entirely unaware. It continues to reason over an outdated snapshot of the world permanently etched into its context history. Consequently, the agent is not structurally coupled with the environment; it merely interacts with a static representation of the environment's past.

2.4.2 The Context Window Saturation

To mitigate the lack of intrinsic environmental observability, current architectures attempt to compensate by pre-loading extensive tool descriptions into the agent's prompt. In the terminology of A&A, this equates to injecting the entire Manual $\mathcal{M}$ (operating instructions, API schemas, and constraints) into the agent's Working Memory prior to execution.

This creates a severe bottleneck regarding the context window, leading to a direct trade-off between the number of available tools and the agent's reasoning capacity. In complex environments populated by dozens of artifacts (e.g., the sprawling ecosystems managed by the Model Context Protocol [5]), the agent requires highly detailed schemas to avoid syntax errors and functional hallucinations. However, injecting these schemas consumes a massive portion of the LLM's context window.

The result is a structural trade-off with no good solution under the current paradigm: loading fewer tool schemas preserves reasoning capacity but limits what the agent can do; loading more schemas extends its reach but degrades the very reasoning it depends on.

2.4.3 The Synchronous Execution Mismatch

Perhaps the most structurally crippling limitation of current paradigms is their *temporal rigidity*. As formalised in Section 2.2.4, the standard ReAct trajectory imposes a strictly alternating, synchronous sequence of cognitive and executive steps. The constitutive incompatibility of this formulation with asynchronous environments becomes apparent when we examine its implications for long-running processes: a standard tool invocation in the ReAct loop occupies the agent's execution thread for the full duration $[t_{\text{call}}, t_{\text{return}}]$, preventing any concurrent observation, planning, or context update.

$$\hat{\tau}_t = (o_1, t_1, a_1, \ldots, o_t, t_t, a_t) \tag{2.5}$$

While elegant for rapid information retrieval (e.g., querying Wikipedia), this synchronous loop catastrophically fails when confronted with the asynchronous reality of A&A artifacts. Real-world operations (such as compiling a

codebase, waiting for human approval via Elicitation [2], or training a model) are long-running processes.

In a standard ReAct loop, invoking such a tool blocks the execution thread. The agent cannot issue concurrent actions, monitor other artifacts, or update its reasoning until the blocking function returns. This blocking execution model violates the core requirement of autonomous agency: the ability to maintain continuous environmental awareness.

Without a mechanism to decouple invocation from completion, the agent cannot react to environmental changes while waiting for a tool to return. It simply blocks the entire reasoning loop.

The architecture required to overcome these limitations cannot be built by patching the existing paradigm. It requires rethinking the tool abstraction from the ground up, which is the starting point of Chapter 3.

Part I

The Apprentice Framework: From Design to Architecture

Chapter 3

The Apprentice Framework

The modeling gap identified in Chapter 2 (Section 2.4) calls for a concrete response. This chapter provides it, laying out the conceptual foundation of the **Apprentice framework**.

The framework advances through two interconnected conceptual contributions. First, it redefines what a tool *is*, formalizing the **Enhanced Tool** meta-model as the theoretical answer to the three structural limitations identified in Section 3.1. Second, it redefines how a tool is *used*, specifying both the cognitive requirements for the agent and the procedural lifecycle governing interaction with these reactive entities (Section 3.2).

The chapter then establishes the orchestration assumptions and introduces the S-ORA reasoning cycle as a conceptual baseline for operating within this new paradigm (Section 3.3). The architectural engineering that gives concrete form to these theoretical concepts is presented in Chapter 4 (The Enhanced Tool Architecture) and Chapter 5 (The S-ORA Agent Architecture).

3.1 The Enhanced Tool Meta-Model

We propose the **Enhanced Tool**, grounded in the Agents & Artifacts (A&A) meta-model (Section 2.3). Rather than a stateless function invoked on demand, this reactive entity maintains a persistent, structural coupling with the language agent, directly improving its situatedness.

3.1.1 Formal Specification of the Enhanced Tool

To transition from a stateless function to a situated entity, we model the Enhanced Tool as a domain object with its own control flow and internal state, whose usage interface is inherently asynchronous. We formally define this entity as a tuple $\mathcal{ET} = \langle \mathcal{P}, \mathcal{S}, \mathcal{O}, \mathcal{M} \rangle$, representing Observable Properties,

Signals, Operations, and the Tool Manual. Together, these four components solve the structural limitations of current standard tools:

1. Observable Properties ($\mathcal{P}$)

This property directly addresses what Section 1.3 termed *State Blindness* and what Section 2.4.1 formalized as the *Statelessness-Persistence Conflict*: the agent's inability to perceive spontaneous environmental changes.

Unlike the basic tool use paradigm, Enhanced Tools expose a persistent, observable state. **Observable Properties** represent the persistent ground truth from the environment. They allow language agents to monitor the tool's state continuously and asynchronously. This design improves agent efficiency by eliminating the need to repeatedly poll the environment and mitigates hallucinations by grounding reasoning in the actual environmental state rather than relying solely on conversation history. To illustrate this, consider a running example of an asynchronous document converter. In a standard setup, an agent is blind to the server's load. By exposing an Observable Property such as `converter.status` (with values like `IDLE`, `PROCESSING`, or `OVERLOADED`), the agent can verify the tool's actual state before interacting with it, avoiding sending requests to an overloaded server.

2. Signals ($\mathcal{S}$)

This property, alongside the refactored operations, resolves the *Synchronous Execution Mismatch* identified in Section 2.4.3, where blocking tool calls prevent concurrent observation and planning.

While properties represent persistent state, **Signals** represent transient events that occur within tools and carry information relevant to the agent's situational awareness. Operation execution can generate these signals. Signals allow the language agent to remain responsive and maintain liveness. Continuing with the document converter scenario, once a conversion begins, the agent should not repeatedly poll the `converter.status` property to check for completion. Instead, the tool emits a discrete signal, such as `JOB_COMPLETED`. This informs the agent that the process has concluded, allowing it to suspend the waiting activity and advance other tasks in parallel.

3. Operations ($\mathcal{O}$)

While functional endpoints exist in standard LLM tools, they inherently exacerbate the *Synchronous Execution Mismatch* by freezing the agent's runtime until a task is fully completed. In the Enhanced Tool model, **Operations** ($\mathcal{O}$) must be structurally refactored.

Because the usage interface is inherently asynchronous, invoking an operation does not block the agent's control flow. Instead, operations act as non-blocking triggers: they return an **immediate acknowledgment** confirming that the command was accepted, but not necessarily the final result. The actual execution modifies the tool's internal state in the background, eventually updating the observable properties ($\mathcal{P}$) and emitting signals ($\mathcal{S}$) upon completion. In the context of the document converter, the operations represent the executable commands, such as `start_job(file)` and `download_result()`. Unlike standard synchronous tools, invoking `start_job` does not freeze the agent while the file converts; it returns an immediate acknowledgment indicating that the process has successfully started, effectively releasing the agent's thread.

4. The Tool Manual ($\mathcal{M}$)

This final pillar primarily addresses the *Semantic Gap* inherent in standard tool definitions. Furthermore, as a secondary effect, its structured format enables the Just-In-Time loading mechanism implemented in S-ORA (Chapter 5), which contributes to mitigating the *Context Window Saturation* described in Section 2.4.2.

Current tool calling protocols typically provide only minimal API schemas that describe syntax but fail to capture procedural semantics. The Enhanced Tool addresses this gap through the **Tool Manual**, which provides the procedural knowledge required to interact with the object effectively and safely. It formally describes the tool's metadata, its functional purpose, and the strict definitions of its properties, signals, and operations. Crucially, it encodes *Usage Protocols & Safety* constraints, explicitly instructing the agent on when and how to use operations, what to expect, and under what conditions an activity must be suspended to wait for specific environmental feedback. Concluding the document converter example, a standard API schema might simply list the two operations without temporal context. An agent lacking procedural rules might hallucinate a synchronous workflow, calling `download_result()` immediately after `start_job()`, leading to a race condition or an empty file. The Tool Manual explicitly unites the previous three pillars: it instructs the agent to check the `converter.status` ($\mathcal{P}$), invoke `start_job` ($\mathcal{O}$), explicitly suspend execution to wait for the `JOB_COMPLETED` signal ($\mathcal{S}$), and only then call `download_result` ($\mathcal{O}$).

The synergy of these four components ($\mathcal{P}$, $\mathcal{S}$, $\mathcal{O}$, $\mathcal{M}$) creates a **reactive, stateful entity** capable of independent execution, rather than a passive function waiting to be called.

3.1.2 Structuring the Manual

A core principle of the A&A meta-model is that agents in open systems must learn to interact with their environment dynamically rather than relying on pre-programmed knowledge. Translating this principle into a working LLM-based architecture, however, requires a concrete data structure.

We draw architectural inspiration from the Agent Skills framework discussed in Section 2.1.3. Ontologically, a "skill" represents an internal agent capability, a conceptual mismatch for an external tool. Yet the *implementation pattern* of skills, packaging domain expertise into self-contained directories that combine lightweight metadata with rich markdown instructions, offers a natural blueprint for structuring the Enhanced Tool's Manual ($\mathcal{M}$).

Three considerations motivate this choice. First, **procedural encapsulation and safety**: standard API schemas lack the expressive power to enforce safety protocols, whereas a manual bundled with detailed operational instructions acts as a guardrail, teaching the agent *how* and *when* to invoke operations safely. Second, **progressive disclosure**: real-world tool documentation is notoriously verbose, and empirical evidence confirms that token bloat is a structural problem rather than a minor inconvenience (Section 2.1.3). The two-tier discovery mechanism inherited from the skills format lets agents scan lightweight metadata first and retrieve full documentation only when they actually commit to using a tool, directly relieving Context Window Saturation. Third, **intrinsic context**: by embedding instructions directly in the tool's architecture, the Manual travels with the tool itself, ensuring its operational context remains independent of whatever the agent was pre-trained on.

It bears emphasizing that borrowing the Agent Skills pattern is strictly an architectural choice for the semantic interface. Ontologically, the Enhanced Tool remains a distinct external entity, and the Manual is simply its readable descriptor.

3.2 The Enhanced Tool Usage

Redefining the tool inherently redefines the agent. Because execution is now decoupled from the reasoning loop, the agent shifts from being a sequential *caller* of blocking endpoints to an active *participant* in a dynamic world. This shift is not merely operational; it imposes specific cognitive requirements.

3.2.1 Cognitive Requirements for the Situated Agent

To interact with these reactive objects effectively, a situated agent must possess capabilities that extend beyond the standard ReAct loop:

1. **Dynamic Context Management:** Since Enhanced Tools come with extensive Manuals ($\mathcal{M}$), it is impossible to fit all documentation into the context window at once. The agent must treat its context as a limited, dynamic resource, loading the specific procedural knowledge needed for the current goal and discarding it when no longer relevant.

2. **Selective Attention:** Unlike standard agents that only process the direct return value of a function, a situated agent is exposed to continuous environmental telemetry ($\mathcal{P}$). It must explicitly decide *what* to observe. This requires the ability to filter the environment: the agent must be able to "focus" on specific tools to monitor relevant properties and signals ($\mathcal{S}$), ignoring noise from other parts of the system.

3. **Asynchronous Flow Control:** In standard architectures, "waiting" blocks the entire system. In our model, waiting is a deliberate cognitive decision. The agent must be able to interpret the Manual's safety protocols and decide to **suspend** a specific goal. Crucially, while that execution is suspended, the agent remains active and responsive, ready to process incoming signals to resume the workflow.

3.2.2 The Interaction Process Lifecycle

To operationalize these cognitive requirements, we define a lifecycle composed of five distinct phases. This process replaces the linear *Request* $\rightarrow$ *Execution* $\rightarrow$ *Response* cycle with a more flexible, stateful approach.

Discovery

The process begins with discovery. The agent scans the environment using lightweight metadata (e.g., tool names and short descriptions) to identify which tools are relevant to its current goal. This step avoids overloading the agent's memory with full documentation for tools that are not currently needed.

Learning

Once a tool is selected, the agent retrieves its **Manual** ($\mathcal{M}$) and loads it into the context. In this phase, the agent reads the functional specifications and usage protocols. This is where the agent learns not just *how* to call a function, but *when* to do so and *what* to expect in return.

Focus

Before executing any command, the agent must ”tune in” to the tool. Guided by the manual, the agent **subscribes** to the tool's data stream. This enables *active perception*: the agent begins monitoring the tool's persistent state and listening for real-time events, grounding its reasoning in the live status of the environment.

Operation

The agent invokes an action defined in the tool's interface. This invocation is **asynchronous**. The tool returns an immediate acknowledgment to confirm the command was received (e.g., ”Starting sequence initiated”), but not the final result. This allows the agent to continue thinking or monitoring the environment while the tool performs the work in the background.

Suspension and Resumption

This phase handles long-running processes and safety protocols.

- **Suspension:** If the manual specifies that an operation requires time, the agent does not block its execution thread. Instead, it interprets the protocol and decides to **suspend** the current execution, entering a semantic ”waiting state.”
- **Resumption:** The execution remains suspended until the specific event described in the manual is received. This event acts as a trigger, waking up the agent's reasoning process to evaluate the new state and proceed to the next step of the workflow.

This separation between initiating an action and finalizing it allows the agent to respect complex safety constraints without hallucinating results or freezing the system.

3.3 Agent Design Rationale and the S-ORA Cycle

Two design decisions need to be made explicit before moving to the architectural implementation: what the agent is expected to manage on its own, and the reasoning cycle that governs its behavior. Both shape the architectural choices detailed in the chapters that follow.

3.3.1 Goal Orchestration Assumptions

The Apprentice framework operates under a specific assumption about task decomposition: a complex user request is broken into atomic sub-goals by an upstream orchestrator that lies outside the scope of this contribution.

How those sub-goals are executed depends on their mutual dependencies. When the orchestrator submits independent goals, the situated agent interleaves their execution natively. When goals carry strict logical dependencies (where g_2 requires the output of g_1), sequencing is delegated back to the orchestrator. The agent functions as a pure execution engine without an internal dependency graph; the orchestrator achieves sequentiality simply by withholding g_2 until it observes g_1 completing.

3.3.2 The S-ORA Cycle

The standard ReAct loop is structurally insufficient for asynchronous interaction. Its synchronous rhythm cannot accommodate an environment where tools push state updates independently of the agent's generation cycle.

This work proposes **S-ORA (Situate, Observe, Reason, Act)** as a foundational reasoning cycle suited to this environment. S-ORA is not presented as the only possible solution, but as a concrete and well-motivated baseline: an agent operating within this cycle first *Situates* itself by aligning its internal context (including the retrieval of relevant manuals and the state of active tool focuses), *Observes* the environment by ingesting properties and signals from the artifacts it is currently *focusing* on, *Reasons* to select the next valid step, and *Acts* by invoking operations or deliberately suspending execution.

The following chapters detail how this cycle is concretely engineered: Chapter 4 presents the Enhanced Tool environment architecture, and Chapter 5 details the S-ORA Agent runtime.

Chapter 4

The Enhanced Tool Architecture

This chapter translates the formal specification of Section 3.1 into a concrete architecture.

To support the **asynchronous tool use paradigm**, the interaction between the agent and the Enhanced Tool is grounded in the **Publisher-Subscriber (Pub/Sub)** model: the tool functions as an autonomous *Publisher* of state changes, and the agent acts as a selective *Subscriber*. This decoupling is what allows the tool to emit asynchronous updates without blocking the agent's cognitive process.

4.1 The Enhanced Tool Structure

The architectural core of the Enhanced Tool marks a shift from stateless function library to *stateful mediator*. Unlike standard LLM tools, which act as transparent proxies for synchronous API calls, the Enhanced Tool is a long-lived entity with its own internal lifecycle whose state evolves independently of the agent's interactions.

This design creates a fundamental tension: the agent operates on a synchronous, discrete generation loop, while the tool's internal processes are asynchronous and continuous. The architecture resolves this tension by defining a structured protocol built around three capabilities: *Discovery & Learning*, *Subscription*, and *Asynchronous Execution*.

4.1.1 Discovery and The Learning

Agents in open-ended environments cannot be pre-trained on every tool they may encounter, so the architecture supports a two-stage process: identi-

fication followed by acquisition.

The first stage, Discovery, is handled by a lightweight metadata layer. The interface exposes an identification profile consisting of high-level metadata such as the tool name and a concise functional description. This profile acts as a semantic index, allowing the agent to scan the environment and identify candidates relevant to its current goal without the need to process full documentation. By exposing this minimal surface, the architecture ensures that the agent memory is not overloaded with unnecessary details for tools that are not currently required.

Identification alone, however, is not sufficient for safe operation. Once the agent selects a relevant tool, a dedicated learning endpoint serves the full Manual ($\mathcal{M}$) on request. Architecturally, this requires the tool to treat its own documentation as a retrievable resource, strictly decoupled from its execution logic, a clean separation of concerns that also enables the Just-In-Time loading strategy described in Chapter 5.

4.1.2 Tool continuous observation

Standard tool architectures are stateless: they have no concept of a connected user beyond the lifespan of a single request. Supporting the *Focus* phase of our interaction protocol (Section 3.2.2) therefore requires something fundamentally different, a persistent **Subscription Method**.

When an agent invokes the subscribe action, two operations take place. First, a **logical notification channel** is established between the tool and the agent. This architectural abstraction defines a dedicated path for the delivery of *Observable Properties* and *Signals*, ensuring that the agent can receive environmental updates as they occur. Second, the agent is registered in an internal Subscriber List, enabling *selective broadcasting*: a change in an observed property is not sent to the entire network but targeted specifically to agents that have explicitly subscribed to that tool. This transforms the tool from a passive server into an active publisher.

Refining Observability: Properties vs. Signals

To propagate the evolving state without flooding the agent context, the architecture explicitly decouples continuous state monitoring from control flow management. If every variable update triggered a reasoning cycle, the computational cost would be prohibitive. To resolve this, the architecture distinguishes between two types of updates based on their semantic weight:

- **Continuous Telemetry (Passive Observations):** The tool exposes high-frequency updates of its internal status via Observable Properties.

These are intended for *passive ingestion*: they synchronize the environment ground truth in the background, allowing the agent to access up-to-date context whenever it is already engaged in reasoning, without forcing unnecessary cognitive interrupts.

- **Meaningful Semantic Signals (Cognitive Interrupts):** To manage the control flow, we introduce **Signals**. These are discrete, qualitative events emitted when specific logic predicates are satisfied. While any state change can carry semantic weight, Signals are explicitly designed as *active interrupts*: they notify the agent that a milestone or a critical threshold has been reached, requiring immediate evaluation and potentially triggering a transition from a dormant to an active reasoning state. It is worth noting that this semantic weight is assigned at design time by the tool designer; a more flexible architecture in which the agent layer itself determines what constitutes a meaningful event is left as future work (Section 8.2.4).

4.1.3 The Operations

The final component handles the domain-specific functions. In standard architectures these are blocking calls: the agent sends a request and waits until the task completes. In the Enhanced Tool architecture, execution methods function as **Triggers**.

When an agent invokes an operation, the interface follows a non-blocking pattern:

1. **Validation:** The request is checked against the tool's current internal state.
2. **Execution Initiation:** The background **asynchronous task** is initiated or signaled.
3. **Immediate Acknowledgement (ACK):** The interface returns a structural confirmation immediately, releasing the agent's **control flow**.

This shifts the complexity of state management from the agent's prompt to the tool's backend, allowing the agent to remain responsive while the tool handles long-running processes in parallel.

4.2 The Semantic Manual Schema

This schema acts as the cognitive interface for the tool, transforming raw, unstructured documentation into a rigorous format that the agent can parse,

index, and actively use for reasoning.

To bridge the gap between static code and dynamic execution, the schema is organized into three distinct layers, each serving a specific cognitive function: the **Functional Description**, the **Usage Interface**, and the **Protocol & Safety** definitions.

4.2.1 Functional Description

The first section provides a high-level natural language summary of what the tool does, serving as the discovery layer. When an agent initializes a task, its embedding model analyzes this description to determine semantic relevance, going beyond keyword matching to consider the tool's category, version, and operational role. Distinguishing whether a tool functions as an *active actuator* (performing tasks autonomously) or a *passive enabler* (unlocking capabilities for other tools) sets correct expectations regarding autonomy before any interaction begins.

4.2.2 Usage Interface Description

The core of the manual maps the tool's internal logic directly onto the agent's cognitive cycle, creating a bidirectional, state-aware contract. Standard APIs specify what to call; this interface additionally specifies when, why, and what to expect.

Observable Properties

This subsection defines the telemetry stream. It lists the state variables exposed by the tool, which the agent can monitor via the subscription method. To prevent the agent from hallucinating values or misinterpreting units, the schema enforces a strict tabular definition for each property:

- **Property Name:** The exact key used in the telemetry.
- **Data Type:** The primitive type, such as String, Integer, or Boolean.
- **Range/Enum:** The specific set of valid values or numerical limits, defining the boundaries of normal operation.
- **Description:** A semantic explanation of what the value represents in the physical world, grounding the data in reality.

By strictly defining these properties, the architecture allows the agent to validate its internal perceptual buffer against the formal definition of the environment.

Operations

This subsection defines the functional entry points, representing the actions the agent can dispatch. The Semantic Manual requires declarations of behavior and effects to support strategic planning. Each operation includes:

1. **Description:** A concise summary of the action's purpose.

2. **Behavior:** Describes the specific operational logic and temporal dynamics of the function. It explicitly details what the method does immediately upon invocation and how the system state is expected to evolve as a consequence. This semantic description allows the agent to distinguish between operations that produce instantaneous state changes and those that initiate complex, long-running processes, enabling it to autonomously determine the appropriate monitoring strategy.

3. **Preconditions:** Defines the required state of the system *before* the action can be safely invoked.

4. **Effects:** Describes the projected causal impact on the system state. To enable predictive planning, this field explicitly maps the operation to its observable consequences, detailing the **Telemetry Mutations**.

5. **Payload:** The exact structure required by the runtime engine to execute the command.

Signals

To handle asynchronous feedback without forcing polling, the manual defines **Signals** as discrete, semantically significant events distinct from Observable Properties. For each signal, the schema specifies the trigger condition and payload structure. This definition enables the architecture's event listener to filter the incoming stream precisely, waking suspended activities only when a signal relevant to the current goal arrives.

4.2.3 Protocol & Safety

The final section constitutes the constraint layer. In complex or critical systems, simply knowing the available operations is insufficient to prevent failure. The agent must understand the procedural "grammar" that governs their interaction.

This section encodes the rules of engagement:

- **Critical Constraints:** Forbidden state-action pairs that could lead to irreversible failure.
- **Sequential Dependencies:** The mandatory order of operations.
- **External Prerequisites:** External dependencies, such as authentication levels that must be acquired via auxiliary tools.

By explicitly documenting these protocols, the Semantic Manual shifts the burden of safety from hard-coded logic in the runtime engine to dynamic, learnable knowledge within the agent's context.

Chapter 5

The S-ORA Agent Architecture

5.1 Architecture Overview

The Interaction Process Lifecycle (Section 3.2.2) requires a runtime built around the same asynchronous information flows that define the Enhanced Tool architecture (Chapter 4). Since Enhanced Tools push state updates and signals independently rather than waiting to be polled, a linear execution script is structurally incompatible with this model. The agent's runtime must mirror the reactivity of its environment.

The solution is to architect the agent as a non-blocking engine, representing a concrete instantiation of the **CoALA 2.2** conceptual framework [16]. By adopting the CoALA blueprint, the S-ORA architecture organizes the agent's internal processes into a structured modular system where the LLM serves as the core reasoning unit. Specifically, the system is centered around a coordinator that manages the interaction between the agent's memory modules and the environment, maintaining persistent subscriptions to tools, ingesting environmental observations, and dispatching execution to isolated semantic containers called *Activities*.

5.1.1 The Decision Procedure Module

To ensure responsiveness without blocking on tool outputs, the runtime abandons the standard "call-and-wait" cycle typical of traditional ReAct implementations. The concurrent execution of activities is managed by the **Decision Procedure module**.

Rather than relying on a rigid, synchronous script, this central module implements a continuous event processing pipeline relying on three conceptual components:

- **The Perceptual Listener:** This component manages subscriptions to the Enhanced Tools' asynchronous channels. It receives raw streams (observable properties and signals) and reflects them passively into the agent's working memory.
- **The Activity Scheduler:** This mechanism acts as the synchronization point between the asynchronous environment and the cognitive core. By managing queues of ready and suspended Activities, it prevents bursts of external events from overwhelming the reasoning engine.
- **Event Loop:** Operating as the main execution loop, it continuously selects an activity from the scheduling queue and advances its reasoning process by executing one discrete action at a time according to the S-ORA Cycle.

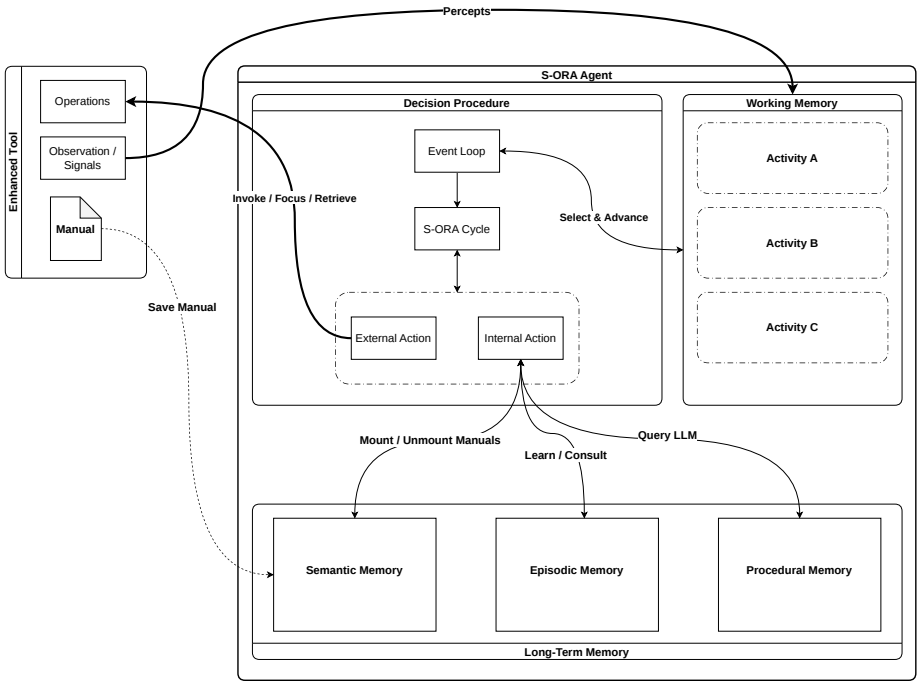

Figure 5.1: **Overview of the S-ORA cognitive architecture.** The diagram illustrates the operationalization of the CoALA memory modules. The Working Memory is structured as a federation of isolated activities to support concurrency. The Decision Procedure orchestrates execution utilizing internal actions for memory management and external actions to interact asynchronously with the environment.

5.1.2 Bimodal Perception and Dispatching

The architecture implements the **Bimodal Perception** to reconcile the agent's internal context with Enhanced Tool outputs. An Event Dispatcher routes information to the appropriate cognitive pathway, implementing the consumption model from Section 4.1.2:

- **Observable Properties:** This pathway manages the *Passive Observations* defined in the tool architecture. The Dispatcher automatically updates persistent variables in the agent's working memory without interrupting the execution flow.
- **Signals:** Adhering to the definition in Section 4.1.2, these signals are treated as **Cognitive Interrupts**. Unlike the silent nature of observable properties, signals explicitly trigger observability: the Dispatcher routes them to the correct section of Working Memory, prompting the agent to actively evaluate the new environmental state.

This dual structure ensures the agent distinguishes between maintaining the *state* of the world and reacting to the *flow* of the process. This approach prevents the reasoning engine from being overwhelmed by a continuous stream of low-level environmental updates, which would otherwise saturate the context window and trigger redundant, costly reasoning cycles. By insulating the reasoning loop from high-frequency noise, the architecture ensures that the agent's attention and computational resources are reserved exclusively for semantically significant events.

5.2 The Memory System

The S-ORA architecture instantiates the abstract memory modules proposed by CoALA (Section 2.2) into concrete runtime components tailored for the Situated Interaction Model. The contribution here is not simply mapping theory to practice: it lies in how these memory tiers are engineered to support asynchronous concurrency and strictly typed tool interactions.

5.2.1 Working Memory

The Working Memory is not implemented as a monolithic storage buffer, but rather as a **structured federation of execution contexts**. Following the CoALA model's definition of working memory, we designed this component to serve as the aggregate container for all active tasks.

The operational core of this module is the principle of **Activity-Based Partitioning**. In a complex, event-driven environment, mixing observations and signals from different asynchronous processes into a single global state would inevitably lead to reasoning hallucinations and state corruption. To mitigate this risk, our architecture segments the memory into discrete, isolated units known as **Activities**.

Each Activity functions as a dedicated "memory slice" for a specific goal. Within this partition, the system enforces boundaries to ensure data integrity. Incoming signals are rigorously dispatched, ensuring that an Activity processes only the environmental feedback relevant to its own subscription, while observed variables remain strictly encapsulated to prevent cross-contamination between different reasoning processes.

Furthermore, this structure addresses the bottleneck of **Context Window Saturation**. Instead of forcing the agent to load the entire library of available tool manuals at once, the Working Memory allows each Activity to transiently "mount" only the specific knowledge bases required for the task at hand, keeping the context sparse and highly relevant.

5.2.2 Semantic Memory

Semantic Memory serves as the central library for the agent, storing declarative knowledge about the environment. Beyond storing **complete tool manuals**, it also maintains a high-level **catalog of available tools**, including their unique identifiers and functional descriptions. In our architecture, this knowledge is not generic text but structured documentation that adheres to the Semantic Manual Schema defined in Section 4.2.

Not all manuals need to be in context at once. Rather than injecting everything upfront, we implement a **Parent Document Retrieval** strategy that organizes this knowledge into two functional tiers available to all concurrent activities:

1. **The Lightweight Catalog (Discovery Tier):** When a manual is downloaded, the system extracts its high-level metadata, such as the tool's identifier and a concise semantic description. This information is stored in a lightweight index. This allows any activity to perform rapid "Tool Discovery" across the entire workspace without the overhead of loading the full content. It effectively filters out irrelevant tools before any significant cognitive cost is incurred.

2. **The Full Manual Repository (Execution Tier):** Alongside the catalog entry, the system stores the complete and unprocessed manual in a persistent repository. It is retrieved and "mounted" into an Activity's

Working Memory only when the agent explicitly selects that specific tool from the catalog and determines that accessing its full interface is necessary to safely execute a task.

This structure creates a clean separation of concerns. The Semantic Memory persists the knowledge globally while the Working Memory loads it sparsely and only on demand.

5.2.3 Episodic Memory

Working Memory is ephemeral by design; Episodic Memory compensates by persisting experience across tasks. The mechanism relies on **Procedural Summaries**: upon completion of an Activity, the system processes the raw interaction log to extract either a refined action sequence for successful outcomes or a summary of the factors that led to failure. These summaries are archived and queried when the agent encounters semantically similar goals in the future, injecting both proven procedures and negative constraints into the new Activity's context before planning begins. This dual perspective ensures that the agent can replicate efficient workflows while systematically avoiding the repetition of previously encountered mistakes.

5.2.4 Procedural Memory

The S-ORA architecture relies on the **Implicit Parametric Knowledge** of the Large Language Model to fulfill the role of Procedural Memory. While CoALA also supports explicit procedural knowledge in the form of plan libraries, this work focuses on the LLM's pre-trained weights as the reasoning engine: the agent queries the model to interpret the current state and the loaded tool manuals, dynamically deriving the next logical step without relying on hand-crafted plans.

5.3 The Activity Construct

The **Activity** is the central architectural primitive of the S-ORA system. It simultaneously serves as the atomic unit of memory isolation and the atomic unit of concurrency, making it the bridge between the static memory architecture and the dynamic runtime. This subsection defines what an Activity *is*: its internal structure and its role as a scheduling unit. How the Agent operates on it over time is detailed in the Reasoning Runtime (Section 5.4).

Internal Structure

Every Activity is initialized by the specific **Goal** that generated it and assigned a unique identifier that serves as a correlation key for routing incoming signals and observations. Conceptually, it is analogous to an **Intention** in the classical BDI model: a goal the agent is currently committed to, persisting until either achieved or deemed impossible. Internally, each Activity holds three components within its Working Memory partition. The **Interaction History** is a chronological log of all reasoning steps, actions, and results, the narrative of what has happened so far. The **Observed Variables** and **Signals** representing the current environmental condition as perceived by this specific activity. The **Context Configuration** specifies which tool manuals are currently loaded, defining what the agent knows how to do at this moment. Together, these three components let the Agent "freeze" a task in full fidelity and resume it later, which is the precondition for the concurrency model that follows.

The Activity-Based Concurrency Model

The self-contained nature of the Activity directly enables the Decision Procedure to manage multiple Activities simultaneously. Because every Activity carries its own isolated context, the engine can perform efficient context switching between different goals without risk of state contamination. This architecture enables true non-blocking multitasking, allowing the agent to advance the reasoning of one task while another is waiting for an asynchronous process to complete.

To orchestrate this flow, the engine utilizes two primary scheduling structures:

- *The Ready Queue:* This buffer stores all activities currently in a ready state. It serializes both new goals and previously suspended activities that have been promoted back to execution after a context switch. By processing reasoning steps one at a time across different activities, the engine maintains a high degree of responsiveness without state contamination.

- *The Wait Queue:* This structure manages activities that are currently suspended. These tasks remain idle, yielding computational resources to other activities while waiting for external triggers. An activity remains here until the required signal is received.

5.4 The S-ORA Decision Cycle

With the memory architecture and the Activity primitive established, we can now describe the dynamic runtime that orchestrates the system. A critical distinction underlies everything that follows: the **Decision Procedure** is the active execution engine, while the **Activity** is the passive container for state and context. The Decision Procedure acts on Activities; Activities do not act on themselves.

The runtime operates on two distinct layers. The **Lifecycle Orchestration** governs how the Decision Procedure instantiates, schedules, and suspends concurrent Activities. The **Internal Cognitive Cycle (S-ORA)** is the reasoning loop the Decision Procedure executes on a single Activity while it holds the focus of attention.

5.4.1 The Activity Lifecycle

The Activity lifecycle is managed entirely by the **Decision Procedure**. Mirroring process management in operating systems, an Activity is transitioned through four distinct states:

- **Ready:** The Activity resides in the ready queue, initialized and waiting to be allocated computation.
- **Running:** The Decision Procedure has selected the Activity as its current Focus of Attention and is actively modifying its memory, updating its history, and advancing its execution logic.
- **Blocked:** The Activity is suspended after the Decision Procedure dispatches an asynchronous operation. It remains dormant, consuming no reasoning resources, until a correlated Signal arrives from the environment.
- **Terminated:** The Activity is archived upon goal satisfaction or abandonment, triggering episodic memory consolidation.

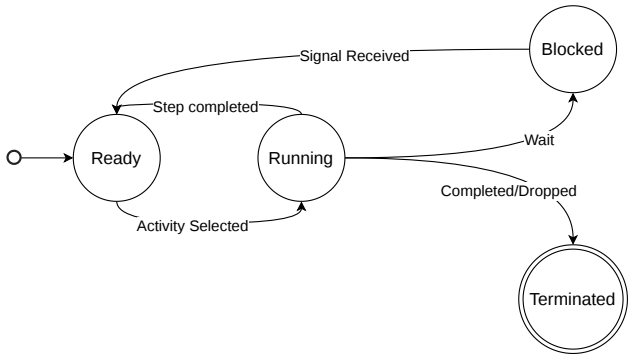

Figure 5.2: **The Activity Lifecycle.** Operational states of an activity managed by the S-ORA decision procedure. Activities yields control (transitioning to Ready or Blocked) to allow the handling of concurrent goals.

5.4.2 From Goal to Activity: The Initialization Pipeline

The lifecycle begins with an **Instantiation Phase**. The Agent does not simply "start" a task; it follows a strict initialization pipeline to transform a raw user intent into a managed software process:

1. **Goal Acquisition:** The process initiates when the Agent receives a high-level natural language Goal from the user or an external system.

2. **Activity Instantiation:** The Agent creates a new **Activity Instance**, assigning it an isolated working memory partition. At this stage, the context is empty and unconfigured.

3. **Scheduling:** The Agent submits the new Activity to the **Ready Queue**.

5.4.3 The S-ORA Cognitive Cycle

The S-ORA decision cycle is the central control loop of the agent, structured around four conceptual phases (Situate-Observe-Reason-Act) as shown in Figure 5.3a. In each iteration, the cycle selects one activity to progress and executes at most one external action. As shown in Figure 5.3b, activity management (including the selection of a ready activity and the resumption of suspended ones) is an integral part of the cycle itself (Step 2), not a precondition to it. To support true concurrency, the cycle is not executed as a monolithic block: the Agent advances the selected activity by a single logical step and then immediately yields control. Unless the activity enters a blocked state, it is returned to the *Ready* queue at the end of the step, allowing the

Agent to interleave multiple parallel goals and preventing any single complex task from starving the system.

The execution of this cycle is driven by the Agent's **Decision Procedure**, which in every iteration selects a single, discrete operation from a structured action space. The S-ORA Agent operates within two distinct functional domains:

- **Internal Actions (Self-Regulation):** These operations allow the Agent to modify its own memory or determine the execution flow without affecting the outside world. This category includes:
    1. **Manual Storage:** The ability to store retrieved tool manuals into semantic memory, making them available for future activities without requiring retrieval again.
    2. **Context Configuration (Mounting):** The ability to dynamically load or unload tool manuals from semantic memory into working memory. This defines *what* the agent knows how to do at any given moment.
    3. **Context Filtering:** The Agent actively filters the Activity's context before each phase, ensuring the LLM processes only the data strictly necessary for the current step rather than the full activity state.
    4. **Plan Inference:** The Agent interacts with its procedural memory by querying an LLM to infer or revise a plan for the current activity and select the next action to advance it.
    5. **Suspension:** The Agent can invoke the *suspend* internal action to suspend the current activity while waiting for external events, transitioning it to the Blocked state without occupying the execution queue.
    6. **Experience Retrieval:** The ability to consult Episodic Memory to retrieve successful completion summaries from previous similar activities, guiding planning for newly created activities toward proven strategies.
    7. **Experience Consolidation:** The explicit decision to finalize the Activity when the goal is satisfied (or deemed impossible), triggering the *learn* internal action to summarize the activity's history into an experience stored in Episodic Memory.
- **External Actions (Environment Interaction):** These operations are directed outward to manipulate available tools or acquire information from the environment:

1. **Tool Discovery:** The active scanning of the environment to identify available tools and their metadata before selecting them.
2. **Knowledge Acquisition:** The active retrieval of tool documentation, enabling the Agent to learn how to use a tool before interacting with it. This distinguishes S-ORA from paradigms where agents operate tools without prior knowledge of their interfaces.
3. **Tool Manipulation:** The invocation of tool operations to perform tasks in the environment.
4. **Subscription Management (Focus/Unfocus):** The explicit act of subscribing or unsubscribing to a tool's observable properties and signals, enabling selective monitoring of only the parts of the environment relevant to the current activity.

The cycle proceeds through six steps described below.

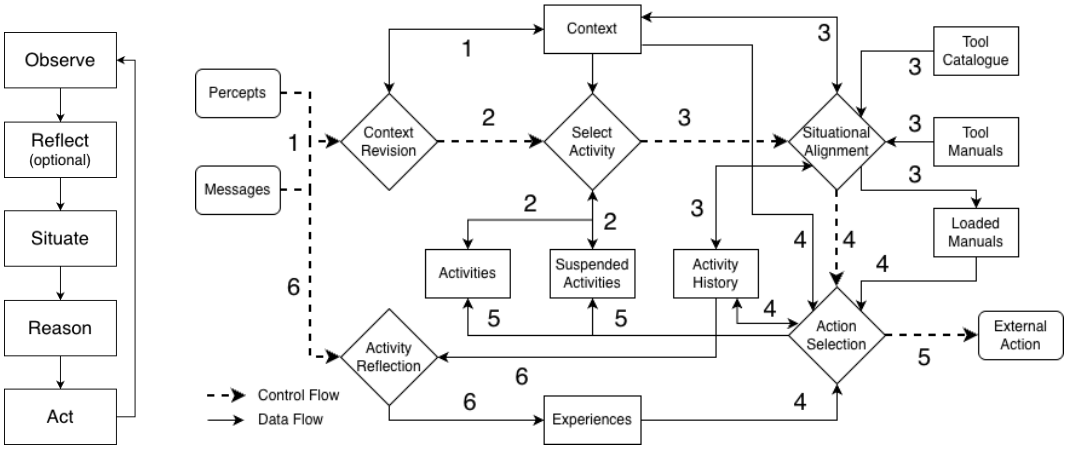

Figure 5.3: **The S-ORA decision cycle:** (a) the four main conceptual phases of the decision loop, and (b) the six execution steps of the decision procedure.

Step 1 – Context Revision (Observe)

The decision cycle begins by retrieving the current set of percepts, defined by observable properties and signals in the environment, and reflecting them in the agent's context. Observable property updates are written passively into the agent's working memory, while signals are dispatched to their target activity. If new signals are detected, the agent is directed to the Observe phase before any reasoning occurs, ensuring that environmental reality always takes precedence over an outdated plan. The Observe phase is also where the agent

interprets what has changed, updates its plan accordingly, and determines whether the goal has been achieved. If the activity has completed successfully, it transitions to the optional Reflect phase.

Step 2 – Activity Selection

The Select Activity function selects the current activity based on the agent's revised context, as well as its ongoing and suspended activities. An activity is selected from the ready queue if it has pending work to perform, or a suspended activity is resumed if the external event it was waiting for has arrived. This function can also create a new activity, for example when a new goal is assigned to the agent. Upon creation, the Episodic Memory is queried using the activity's goal to retrieve successful procedural patterns from previous similar tasks, priming the first reasoning cycle with proven strategies before any action is taken.

Step 3 – Situational Alignment (Situate)

The Situational Alignment function adjusts the agent's working memory for the selected activity by selecting tools, loading and unloading manuals, and filtering observable properties and signals relevant to the current task. The Agent operates as a Just-In-Time memory controller: it mounts only the manuals it needs for the immediate sub-goal, unmounts those that are no longer relevant, and filters the perceptual input to fit the needs of the current activity, keeping the context window sparse and focused.

Step 4 – Action Selection (Reason)

Once the context is configured, the Agent enters the core decision-making phase. It infers or revises a plan for the current activity, taking as input the Activity History, which is a log of all reasoning steps, actions, and results to date, alongside the current context and loaded manuals. For newly created activities, it also consults Experiences from episodic memory to guide planning toward previously successful strategies.

This phase enforces a strict knowledge constraint: before deciding to execute any action, the agent verifies that it holds a manual for every tool it intends to use. The Situate phase may suggest prerequisite external actions for situated reasoning, such as retrieving manuals from an external repository or focusing on and unfocusing from tools. These prerequisite actions take priority unless a more urgent action is needed, for example to respond to a critical signal. Otherwise, the agent selects the next external action that advances the plan.

Step 5 – Action Execution (Act)

The decision cycle completes by executing the external action selected in the previous step. The agent invokes at most one tool operation per cycle. If the tool's manual specifies waiting for a signal or an observable property change before proceeding, the agent invokes the *suspend* internal action. The activity then transitions to the Blocked state and is removed from the ready queue, remaining dormant until the expected external event arrives and reactivates it. When the activity is eventually resumed, the cycle restarts at Context Revision.

Step 6 – Activity Reflection

When the agent determines that an activity has completed successfully, for example by observing signals and observable properties or because all planned actions have been completed, it transitions the activity to the terminated state. Rather than discarding the execution log, the agent executes the *learn* internal action to summarize the activity's history into a consolidated experience. This summarization extracts the optimal sequence of actions that led to the successful outcome and generalizes specific values into reusable patterns. The resulting experience is stored in episodic memory, building a repository of proven strategies that guide future semantically similar activities and closing the cognitive loop.

Part II

Framework development and Validation Tests

Chapter 6

Implementing the Enhanced Tools

This chapter translates the architectural specifications defined in Chapter 4 into concrete software components. Following the theoretical definition of the stateful, asynchronous, publishing entity established in the previous chapter, we focus here on the technical implementation details required to operationalize the *Enhanced Tool*.

The implementation relies on three critical layers: the construction of a polyglot runtime environment using the *Model Context Protocol* (MCP), the formalization of the interface via JSON Schemas, and the internal logic required to manage state and asynchrony.

6.1 MCP as the Connectivity Bus

Within the S-ORA framework, the MCP functions as more than a simple API definition library; it serves as the structural **Environment Interface**. Conceptually, MCP acts as the abstraction bus connecting the cognitive agent to external tools. It harmonizes the heterogeneous capabilities of the tools, whether implemented in Node.js or Python, into a unified JSON-RPC surface.

6.1.1 Polyglot Runtime Strategy

To empirically validate the language-agnostic nature of the architecture, we developed reference implementations across two distinct technology stacks. This approach ensures that the proposed interaction model is not bound to a specific runtime environment but constitutes a generalized protocol.

The Node.js Implementation

The first reference stack is built upon the **Node.js** runtime. As defined in the project configuration ('package.json'), we integrated three core libraries:

- `Express`[1]: To manage the HTTP transport layer.
- `@modelcontextprotocol/sdk`[2]: To handle JSON-RPC schema validation and message framing conformant to the standard.
- `zod`[3]: To define the strict typing of the tool interfaces and programmatically generate the corresponding JSON Schemas via `zod-to-json-schema`.

This environment exploits the native non-blocking Event Loop of the V8 engine. This makes it naturally suited for handling the asynchronous I/O required by the Server-Sent Events (SSE) broadcasting channel without requiring explicit threading, as implemented in tools like the *Water Hammer Simulator*[4].

The Python Implementation

The second reference stack adopts the **Python** ecosystem. The implementation relies on `FastAPI`[5] for high-performance ASGI (Asynchronous Server Gateway Interface) request handling and `sse-starlette`[6] to manage the server-sent event streams.

This implementation was developed specifically to handle hardware-centric scenarios, such as the *Robotic Arm Manipulation* test case [7]. To accommodate the blocking nature of standard Python hardware control libraries, this stack explicitly utilizes the standard `threading` library. By encapsulating the state evolution logic within a background *daemon thread*, we ensure that the API responsiveness remains decoupled from the internal computational latency.

Note on Code Demonstrations: While the architecture has been validated on both stacks, the subsequent sections of this chapter will primarily utilize code snippets from the **Node.js implementation**. This choice is driven by

[1] `https://expressjs.com/`

[2] `https://www.npmjs.com/package/@modelcontextprotocol/sdk`

[3] `https://zod.dev/`

[4] Source code available at: `https://github.com/TonelliLuca/Artisan/blob/main/src/mcp/node/water-hammer.js`

[5] `https://fastapi.tiangolo.com/`

[6] `https://github.com/sysid/sse-starlette`

[7] Source code available at: `https://github.com/TonelliLuca/Artisan/blob/main/src/mcp/mcp_py/mcp_virtual_bot_cube.py`

consistency, as the primary experimental tool used for the complexity analysis was developed within the Node.js environment. However, the architectural patterns discussed are structurally isomorphic in the Python implementation.

6.2 The Interface Definition Strategy: JSON Schema vs. Manuals

A critical implementation detail concerns how the tool presents itself to the agent. While the MCP standard mandates that the interface be formally defined using **JSON Schema**, our architecture enforces a strict separation of concerns between "Discovery" and "Learning."

6.2.1 Schema-Based Discovery

In our implementation, we utilize the ‘zod‘ library to programmatically generate strict JSON Schemas. These schemas serve a specific function: **Discovery**.

The ‘description‘ fields within these schemas are intentionally concise. They provide just enough semantic information for the agent to determine relevance, such as "Use this tool to control hydraulic valves, ..." but intentionally omit safety protocols, edge cases, and complex state dependencies. This lightweight definition ensures that the agent's context window is not saturated with operational details until the tool is actually selected.

6.2.2 Manual-Based Learning

Deep procedural knowledge is decoupled from the JSON schema and offloaded to the **Semantic Manual**. As discussed in the architectural design, the agent must retrieve this manual to understand how to use the tool safely. This split implementation ensures that the JSON Schema drives the **Routing Decision**, while the Manual drives the **Execution Strategy**, preventing the agent from hallucinating usage protocols based on sparse schema descriptions.

6.3 Enhanced Tool Design Patterns

The development of tools such as ‘water-hammer.js‘ and ‘shared-counter.js‘[8] relies on three critical implementation choices: Explicit Subscription, Func-

[8]Source code available at:`https://github.com/TonelliLuca/Artisan/blob/main/src/mcp/node/shared-counter.js`

tional Aggregation, and Embedded Self-Documentation.

6.3.1 The Subscription and Focus Mechanism

Standard MCP interactions are stateless by default, isolating each request. However, the Pub/Sub architecture necessitates that the tool identify the specific consumer to route asynchronous **Signals** correctly.

To ensure correct routing, the architecture imposes a mandatory **Session Binding** logic. This is implemented through a distinct "Login" or "Subscribe" action that registers the specific **Activity's UUID** in the tool's internal 'focusedAgents' set. Listing 6.1 demonstrates how every subsequent action signature includes a mandatory 'uuid' parameter, which the tool validates against its registry before executing any logic. This ensures that asynchronous signals are routed precisely back to the isolated concurrent activity that requested them, rather than broadcasting broadly to the agent's global state.

```
// Registers focused agents to enable targeted SSE broadcasting
const focusedAgents = new Set();

server.tool("counterTool",
    {
        uuid: z.string().uuid(), // Mandatory Identity Parameter
        action: z.enum(["focus", "inc"])
    },
    async ({ uuid, action }) => {
        if (action === "focus") {
            const isNew = !focusedAgents.has(uuid);
            focusedAgents.add(uuid);
            // Returns confirmation and triggers initial SSE sync
                logic
            return { content: [{ type: 'text', text: isNew ? "Focus
                established." : "Already focused." }] };
        }

        // Guard Clause: Ensures the agent is registered before
            performing logic
        if (!focusedAgents.has(uuid)) {
            return {
                isError: true,
                content: [{ type: 'text', text: "Error: You must
                    FOCUS first." }]
            };
        }
```

```
        // ... (execution logic continues for "inc" action)
    }
);
```

Listing 6.1: Session Binding and Access Control Implementation (Simplified from `shared-counter.js`)

6.3.2 Functional Aggregation and Interface Design

Context Window saturation represents a significant bottleneck in LLM-based systems: exposing every atomic operation as a distinct tool inflates the system prompt unnecessarily. To mitigate this, we adopted a **Functional Aggregation Pattern**, grouping related capabilities into semantic "Macro-Tools" (e.g., `hydraulic_control`) and using an `action` parameter to discriminate the specific operation.

In standard LLM architectures, hiding specific endpoints behind a generic dispatcher is often considered an anti-pattern. Without a clear procedural context in the JSON schema, such sparse interfaces typically induce the model to hallucinate invalid action parameters. Within our framework, however, this is a deliberate design choice. By exposing only a generic interface during the *Discovery* stage, the architecture structurally encourages the agent to retrieve and process the **Semantic Manual**. It is through the manual that the agent discovers the valid operations and their underlying causal rules.

Listing 6.2 demonstrates how this dispatcher aggregates operations. If an agent attempts to blind-guess the sequence without consulting the manual, the hard-coded safety interlocks within the tool logic intercept the temporal violation and prevent execution.

```
// Aggregates related operations into a single semantic 'Macro-Tool'
server.tool("hydraulic_control",
    {
        action: z.enum(["power_on_pump", "open_valve"]),
        uuid: z.string().uuid()
    },
    async ({ action, uuid }) => {
        if (action === "power_on_pump") {
            sys.pump_status = "RAMPING"; // Trigger asynchronous
                state evolution
            return { content: [{ type: 'text', text: "Pump started.
                WAIT for NOMINAL status." }] };
        }
```

```
        if (action === "open_valve") {
            // CRITICAL SAFETY INTERLOCK: Detects timing violations
            if (sys.pump_status === "RAMPING") {
                sys.system_lockout = true; // Permanent failure state
                return {
                    isError: true,
                    content: [{ type: 'text', text: "CRITICAL FAILURE:
                        Water Hammer triggered." }]
                };
            }
            sys.valve_status = "OPEN";
            return { content: [{ type: 'text', text: "Valve OPEN.
                Path clear." }] };
        }
    }
);
```

Listing 6.2: Functional Aggregation and Safety Interlocks (Simplified from `water-hammer.js`)

6.4 Implementing Asynchrony

The *Enhanced Tool* is defined by its temporal autonomy; specifically, the capacity to evolve over time and notify the agent via Signals. This reactivity relies on two coupled loops.

6.4.1 The Independent State Evolution Loop

The *Enhanced Tool* is architecturally defined by its internal control flow. Unlike standard passive APIs, it implements an autonomous loop that updates the state Σ according to its internal logic—whether driven by internal process transitions or by concurrent interactions. This mechanism ensures that the environment is a dynamic entity with its own temporal progression, shifting the tool from a reactive function to a stateful, time-aware component.

Listing 6.3 illustrates this principle through the *Water Hammer* simulator. In this implementation, a background process manages latent variables (such as pressure levels and temperature status) and emits discrete signals when critical state thresholds are reached. This design demonstrates how the state evolves according to the tool's internal clock, independently of external requests, effectively operationalizing the concept of environmental persistence.

```
// Updates the internal state independently of Agent requests
setInterval(() => {
    // 1. Simulate Latency: Incremental Pressure buildup
    if (sys.pump_status === "RAMPING") {
        sys.hydraulic_pressure += (Math.random() * 300 + 100);
        if (sys.hydraulic_pressure >= TARGET_PRESSURE) {
            sys.pump_status = "NOMINAL";
            // Fires the discrete Signal
            sendEvent(uuid, "event", "pump.pressure_nominal", { psi:
                2500 });
        }
    }

    // 2. Simulate Decay: Core Temperature reduction
    if (sys.core_status === "FLUSHING") {
        sys.core_temp -= (sys.core_temp - 200) * 0.2;
        if (sys.core_temp <= SAFE_TEMP) {
            sys.core_status = "STABLE";
            // Fires completion Signal
            sendEvent(uuid, "event", "core.stabilized", { temp: 440
                });
        }
    }
    // Continuous broadcast of current state snapshot
    broadcastTelemetry(uuid);
}, 1000);
```

Listing 6.3: The Physics Loop simulating latency and decay (Simplified from `water-hammer.js`)

6.4.2 Transport Layer: Server-Sent Events (SSE)

To enable the Server-to-Agent communication channel, we implemented a custom **Server-Sent Events (SSE)** transport layer. While the current MCP specification defines *Streamable HTTP* as the recommended transport standard [2], this choice was dictated by a concrete implementation constraint: the `langchain4j`[9] MCP client library, used in the S-ORA agent runtime, did not yet provide stable support for Streamable HTTP at the time of development.

SSE was therefore selected as the most compatible alternative to opera-

[9] `https://docs.langchain4j.dev/`

tionalize the **Publisher-Subscriber (Pub/Sub)** architecture. Unlike WebSockets, which require bidirectional framing and a dedicated handshake protocol, SSE provides a lightweight, unidirectional text stream over standard HTTP. This characteristic aligns naturally with our requirement for **selective broadcasting**: while the agent dispatches **Operations** (Tool Invocations) via the standard MCP command channel, it requires a separate, persistent listening channel to receive asynchronous *Telemetry* and *Signals* from the environment.

In our implementation, the tool exposes a dedicated `/sse` endpoint. Upon connection, the HTTP response is kept open with `Content-Type: text/event-stream`, and data packets are pushed as they are generated by the internal evolution loop. This transport layer remains decoupled from the interaction model.

6.4.3 The Communication Protocol: Telemetry vs. Signals

To operationalize the different levels of observability defined in the architecture, we implemented a discriminator within the JSON payload of each SSE message using the `mcpType` attribute. This allows the runtime to process incoming data streams according to their semantic weight.

1. **Continuous Telemetry (`mcpType: "variable"`):** These packets represent the continuous evolution of the state (e.g., a pressure reading changing from 100 to 101). The Agent's runtime interprets the 'variable' type as a passive update.

2. **Discrete Signals (`mcpType: "event"`):** These packets represent significant state transitions (e.g., `pump.pressure_nominal`). The runtime interprets the 'event' type as an active interrupt.

Listing 6.4 illustrates how the tool logic constructs these asynchronous JSON payloads to route information according to the Pub/Sub model.

```
// Implementation of the Bimodal Perception protocol using mcpType

// CHANNEL A: Continuous Telemetry (Passive Memory Update)
function broadcastTelemetry(uuid) {
    const payload = {
        params: {
            uuid: uuid,
            mcpType: "variable", // Discriminator for silent memory
                update
```

```
            name: "reactor_telemetry",
            value: sys // Full state snapshot
        }
    };
    sendSse(JSON.stringify(payload));
}

// CHANNEL B: Discrete Signals (Active Reasoning Interrupt)
function sendEvent(uuid, eventName, eventData) {
    const payload = {
        params: {
            uuid: uuid,
            mcpType: "event", // Discriminator to trigger context
                revision
            event: { name: eventName, payload: eventData }
        }
    };
    sendSse(JSON.stringify(payload));
}
```

Listing 6.4: Implementation of the Semantic Protocol distinguishing variables from events (Simplified from `water-hammer.js`)

6.5 Reference Implementation of a Manual

This section presents a concrete implementation of a Manual. To ensure the Language Model can parse the information efficiently, the manual is structured using standard **Markdown** syntax. This design choice leverages the agent's pre-trained familiarity with structured text (such as headers, lists, and tables), allowing it to interpret the schema defined in Section 4.2 without the overhead of complex JSON parsers.

The example below focuses on the **Fluid Control Unit** (`hydraulic_control`). This tool was selected for the reference implementation because it combines asynchronous temporal dynamics with strict safety constraints.

(For the complete unabridged text of the manual, please refer to Appendix A).

6.5.1 Functional Description and Discovery

The manual begins by establishing the identity and operational scope of the tool. As defined in the schema, this layer acts as the primary index for the

semantic search of the agent.

```
## 1. Functional Description
This tool manages the generation of hydraulic pressure...
It acts as a passive enabler for downstream active systems.
```

Listing 6.5: Extract from the Functional Description defining the tool's role.

By explicitly categorizing the tool as a "passive enabler," the manual informs the agent that this component does not autonomously solve the cooling problem. Instead, it creates the necessary conditions (hydraulic pressure) for other tools, such as the Reactor Core, to function. This distinction prevents the agent from erroneously waiting for the pump to cool the reactor directly.

6.5.2 Usage Interface

This section demonstrates how the tool's internal logic is exposed to the agent.

Observable Properties

The primary function of this section is to establish a rigorous **Data Contract** for the telemetry stream. Since the architecture relies on asynchronous subscriptions, the agent must know exactly which variables will be present in the incoming JSON payloads and what values constitute a valid state.

```
### 2.1 Observable Properties
| Property | Type | Range | Description |
| :--- | :--- | :--- | :--- |
| `pump_status` | String | `OFF`, `RAMPING`, `NOMINAL` | Operational
    state... |
| `hydraulic_pressure` | Integer | 0 - 3000 PSI | System pressure.
    **Operational Target: > 2500 PSI**. |
```

Listing 6.6: Schema definition for the telemetry stream.

By strictly defining the variable names (e.g., `pump_status`) and their admissible ranges (e.g., `OFF`, `RAMPING`, `NOMINAL`), the manual prevents the agent from hallucinating non-existent states or misinterpreting the data types.

Operations

This section of the manual acts as the definitive instruction set for the agent.

For each operation, the manual explicitly defines three critical components: the **Payload** (the syntax), the **Internal Process** (the temporal behavior), the **Causal Consequence** (the observable effect) and the **Preconditions**. The definition of `power_on_pump` below serves as a prime example of how to document a non-blocking, time-dependent process:

```
### 2.2 Operations

* **Operation:** `power_on_pump`
  * *Description:* Energizes the high-pressure pumps.
  * *Behavior:* **Latent.** The transition from `OFF` to `NOMINAL`
    is not immediate. The system enters a temporary `RAMPING` state
    while pressure builds. Completion is indicated by the
    `pump.pressure_nominal` signal.
  * *Effects:* Transitions `pump_status` to `RAMPING`. Initiates the
    physics simulation for incremental pressure buildup.
  * *Preconditions:* `pump_status` is `OFF`. Requires `ADMIN`
    authentication level (via `security_terminal`).
  * *Payload:*
    ```json
    { "action": "power_on_pump", "uuid": "<ACTIVITY_UUID>" }
    ```
```

Listing 6.7: Operational definition of the pump.

In this example, the manual prevents a common failure mode in autonomous agents: the expectation of instant results. By clarifying that the action triggers a `RAMPING` state and defining the behavior as **Latent**, the manual explicitly tells the agent that the immediate return of the function signifies the *initiation* of a process, not its conclusion.

Signals

To close the loop on such long-running processes, the manual defines the specific events that the tool emits. This section informs the agent exactly what ”handshake” to listen for after dispatching the previous command.

```
### 2.3 Signals

* **Signal:** `pump.pressure_nominal`
  * *Trigger:* Emitted automatically when pressure stabilizes at the
    target level (> 2500 PSI).
  * *Payload:* `{ "psi": Integer, "msg": String }`
```

Listing 6.8: Definition of the completion signal for the pump.

By formally defining the `pump.pressure_nominal` signal, the manual establishes a deterministic causal link. The agent understands that the `power_on_pump` operation is only truly ”complete” when this specific broadcast is received.

6.5.3 Protocol and Safety Constraints

Finally, the implementation codifies physical risks into explicit procedural rules. Rather than relying on the model to implicitly infer the ”Water Hammer” risk, the manual defines it as a hard constraint:

```
## 3. Protocol & Safety
**WARNING: WATER HAMMER RISK**
2. **Critical Constraint:** Opening the valve while pressure is
   building (State: ‘RAMPING‘) triggers a "Water Hammer" effect.
   This results in immediate and permanent System Lockout.
```

Listing 6.9: Safety constraint defining forbidden state-action pairs.

This section acts as a ”Negative Constraint” in the search space of the agent. It effectively prunes any plan that attempts to sequence `open_valve` immediately after `power_on_pump` without an intervening stabilization phase. Furthermore, the **Integration Note** in the full text links this tool to the `reactor_core`, establishing a dependency that directs the high-level workflow.

Chapter 7

Implementing the S-ORA Agents

Following the development of the Enhanced Tools in Chapter 6, this chapter focuses on the implementation of the agent. We detail the translation of the S-ORA architectural model and the Activity concurrency paradigm into a robust event-driven runtime.

7.1 Technology Stack

To operationalize the requirements of non-blocking multitasking and asynchronous tool interaction, the agent's runtime is implemented on a technology stack specifically selected for its robust support of concurrent operations and event-driven I/O. This choice ensures that the agent's decision procedure remains decoupled from the latency of individual tool executions, maintaining the architectural integrity of the S-ORA cycle across multiple concurrent activities.

7.1.1 The Host Runtime

We selected Java 21 as the foundational runtime for the agent host. While Python and Node.js were utilized for the tool layer, Java provides advanced native concurrency primitives along with strict static typing. This combination is critical for an architecture where the cognitive loop and the environmental perception operate concurrently. The strict object-oriented paradigm ensures that memory partitions remain rigorously encapsulated and thread-safe during asynchronous state mutations.

7.1.2 The Cognitive Interface

To orchestrate the Large Language Model, we integrated the LangChain4j framework. A key departure from standard agent implementations is our reliance on structured prompt definitions. Instead of using unstructured string manipulation to build prompts, we utilize LangChain4j's declarative `@Agent` and `@UserMessage` annotations. By mapping the cognitive phases to strict Java interfaces, we transform the prompt into a formal software contract. This approach ensures input and output consistency, structurally preventing the model from hallucinating phase transitions.

7.1.3 The Communication Bus

Aligning with the tool implementation described in Section 6.1, the agent acts as an MCP Client. It utilizes the Model Context Protocol standards to standardize tool discovery and execution. For the perception channel, the agent implements a dedicated Server-Sent Events (SSE) client. This allows the runtime to passively ingest the continuous telemetry and discrete signals broadcasted by the tools without blocking the main cognitive thread.

7.2 The Decoupled Orchestrator Design

A key implementation decision concerns the relationship between LangChain4j and our custom runtime. Rather than extending the framework's built-in stateful agent classes, `AsyncAgent` acts as a high-level runtime environment that dynamically manages isolated sub-components, treating the language model as a stateless inference engine.

Concretely, this means LangChain4j never holds conversational state between calls. The `ReactBrain` interface maps a typed input context to a structured JSON output and nothing more. Every piece of context the model needs, including the chronological history, the dynamically updated environment variables, and the specific tool manuals currently mounted, is explicitly extracted from the current `Activity` and injected into the typed parameters of `ReactBrain` at call time.

The only static binding occurs during tool registration. LangChain4j's MCP integration is used to connect the Enhanced Tools (implemented as MCP servers as described in Chapter 6) to the cognitive core at initialization. This is a one-time wiring step that leverages the standard MCP lifecycle: it provides the model with baseline knowledge of the available function signatures (the JSON Schemas) for routing purposes. However, the deep procedural knowledge

(the Manuals) and the runtime state are never statically bound; they remain entirely decoupled and under `AsyncAgent`'s dynamic control.

Because the language model is treated as a stateless inference engine, the `AsyncAgent` owns all memory management. At each step of the **Decision Procedure**, the orchestrator decides what the model sees, dynamically filtering the context based on both the active `Activity`'s isolated memory partition and the current S-ORA phase. This ensures the model always operates on a minimal, relevant slice of working memory rather than a monolithic, cross-task context dump.

The orchestrator also centrally manages all asynchronous I/O. SSE and MCP connections are initialized once at startup and run on a background daemon thread, independent of any ongoing activity. When a signal arrives, `AsyncAgent` extracts the UUID from the payload, locates the matching `Activity`, and routes the event without interrupting the cognitive core. If the targeted activity was suspended in the **Blocked** state, the orchestrator automatically promotes it back to the **Ready** queue.

The result is a clean separation: the language model handles intelligence, the Java runtime handles orchestration. By retaining full programmatic control over state and execution flow, the custom runtime can suspend, resume, and switch contexts between multiple concurrent goals.

7.3 Observable Properties and Signal Handling

Integrating asynchronous signals and continuous observations into the synchronous execution of a language model requires precise state synchronization. To manage this, the `AsyncAgent` implements a two-tiered handling mechanism that separates passive data ingestion from active state validation. This ensures the system remains reactive to external stimuli without executing logic based on outdated environmental information.

7.3.1 Passive Signal Ingestion and Semantic Decoding

The primary method for ingesting environmental data operates independently of the main **Decision Procedure**. The orchestrator initializes a dedicated background thread that maintains a continuous connection to the Server-Sent Events (SSE) stream. This listener acts as a daemon, receiving raw JSON payloads from the environment as they are emitted.

Upon receiving a message, the `handleMcpEvent` method parses the JSON to extract two critical pieces of routing information: the correlation key (`uuid`) and the semantic discriminator (`mcpType`). The `uuid` ensures the data is routed

to the correct isolated *Activity*. Once the target is identified, the agent applies the semantic protocol (defined in Section 6.4.3) to differentiate between continuous telemetry and discrete signals:

- **Continuous Telemetry (`mcpType: "variable"`):** If the payload is flagged as a variable, the orchestrator performs a silent memory write. It updates the corresponding property in the Activity's working memory without interrupting the ongoing S-ORA cycle.

- **Discrete Signals (`mcpType: "event"`):** If the payload is flagged as a discrete signal, it represents a critical state transition. The orchestrator pushes this payload into the thread-safe signal buffer of the Activity. If the target Activity is currently in the **Blocked** state, having previously yielded execution while waiting, the listener automatically transitions it back to the **Ready** queue to initiate a new *Observe* phase.

7.3.2 Active State Validation

While the passive listener manages the routing of incoming data, a secondary active safety mechanism is necessary to handle the inherent concurrency of the system. Because environmental signals can arrive at any millisecond, there is a persistent risk of a race condition: the state of the world might change precisely while an Activity is waiting in the Ready queue or actively being processed by the language model's inference engine.

To mitigate this, the `AsyncAgent` implements an active state validation check within the core loop. Before the engine commits computational resources to the latency-heavy *Reason* or *Act* phases, it evaluates the `activity.hasEvents()` predicate.

```
// Core validation check within the main Decision Procedure
if (activity.hasEvents() &&
    activity.getStatus() != Activity.Status.OBSERVATION &&
    activity.getStatus() != Activity.Status.COMPLETED) {

    logger.info("[GUARD] Pending signals detected. Forcing
        OBSERVATION state.");
    activity.setStatus(Activity.Status.OBSERVATION);
    activityQueue.offer(activity);

    continue; // Abort the current cognitive phase to process new
        environmental data
}
```

Listing 7.1: The active state validation mechanism within the event loop, forcing a context switch to the Observation phase if unprocessed signals are detected.

If unhandled signals are detected, the loop immediately aborts the planned operation, forces a context switch, and demotes the Activity back to the **Context Revision (Observe)** step. As demonstrated in Listing 7.1, this control flow guarantees that the agent always evaluates the most current observable properties before formulating a plan or executing a tool operation, structurally preventing the generation of invalid plans based on stale data.

7.4 Memory Subsystems Implementation

To operationalize the theoretical memory partitions defined previously, the agent relies on a specialized data architecture. The implementation must bridge the deterministic requirements of concurrent tool execution with the probabilistic nature of Large Language Models.

7.4.1 The Activity Container and Working Memory

The `Activity` class serves as the core operational unit of the runtime. It translates the theoretical concept of partitioned working memory into a concrete component by encapsulating all the state variables required to achieve a specific goal. Instead of relying on a global shared context, the system creates a unique `Activity` instance for each task. This approach isolates the execution environment and prevents state contamination between parallel reasoning processes.

Internally, the `Activity` manages the essential components of working memory using a specific set of thread-safe and deterministic collections. These components include State, Context, Experiences, and History:

- **Observable Properties:** The live state of the environment is maintained in a `ConcurrentHashMap` named `beliefs`. This structure allows the background listener to update specific metrics in real-time without locking the main execution thread.
- **Mounted Context:** The `openedManuals` map temporarily holds the text of the tool manuals the agent has actively retrieved. This targeted loading prevents context window saturation by keeping only the knowledge strictly necessary for the current operation.

- **Retrieved Experiences:** To leverage past learning, the container integrates a `LinkedHashSet` named `loadedMemories`. This structure stores procedural summaries retrieved from the long-term episodic memory. By using a set, the activity prevents duplicate memories from overloading the context window. These past experiences are automatically injected to guide the language model with proven strategies for semantically similar goals.

- **Interaction History:** The execution narrative is preserved in a `CopyOnWriteArrayList`. This list chronologically logs every decision, action, and observation. To prevent hindsight bias during later reflection, each step captures an immutable snapshot of the `beliefs` map at the exact moment of execution. By executing `Collections.unmodifiableMap`, the system creates a temporal checkpoint, guaranteeing that the historical log preserves the exact context the agent held during step n, even if this context was subsequently overwritten by signals and observations at step $n + 1$.

- **Signal Buffer:** Incoming signals are safely buffered in a `CopyOnWriteArrayList` named `incomingEvents`. The agent formally processes and clears this queue during the *Observation* phase, migrating the data to a `handledEvents` list to maintain a complete audit trail.

Lifecycle State Mapping and Cognitive Routing

As defined in the architectural model, an activity transitions through abstract states managed by the agent's decision procedure. In our implementation, this lifecycle is controlled by a `Status` enumeration within the `Activity` class. This enumeration serves a dual purpose. It acts both as a process scheduler state and as a cognitive instruction pointer that tracks the exact phase of the S-ORA cycle the agent must perform next.

The runtime engine maps the theoretical lifecycle into six explicit execution states:

1. **Ready and Running (Cognitive Phases):** The active execution phases are directly mapped to the S-ORA cycle: `OBSERVATION`, `REASONING`, `SETUP`, and `ACTION`. When an activity holds one of these statuses, it resides in the scheduling queue and is iteratively processed by the main event loop. By storing the current cognitive phase directly within the Activity state, the orchestrator knows exactly which specialized prompt interface in the `ReactBrain` to trigger. This allows the orchestrator to dynamically filter the context.

2. **Blocked:** The suspended state is represented by `WAITING_FOR_EVENT`. This status is assigned when the agent triggers a long-running tool and determines that active monitoring is unnecessary. By deliberately yielding computational resources, the system allows the activity to remain dormant until a specific signal wakes it up.

3. **Terminated:** The completed state is denoted by `COMPLETED`. Reaching this stage triggers the reflection pipeline and removes the activity from the active registry.

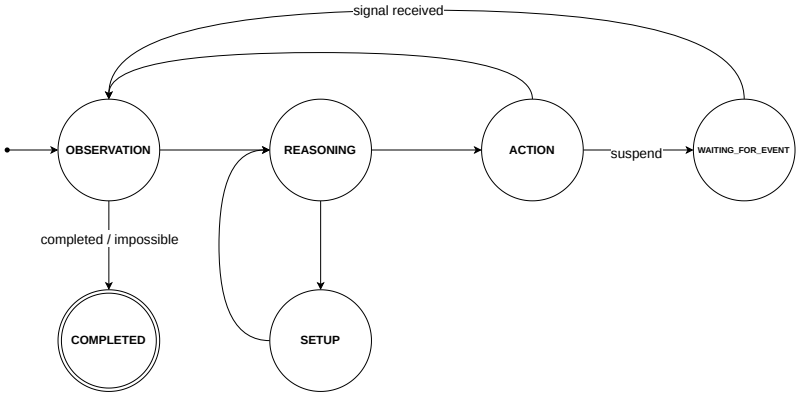

Figure 7.1: **Implemented Activity Lifecycle.** The state machine maps the theoretical S-ORA phases into concrete execution statuses defined in the `Activity` class. The diagram highlights the sequential cognitive flow and the asynchronous *suspend* / *signal received* transitions.

Context Formatting

Beyond storing data, the `Activity` is responsible for formatting its context. Since the language model cannot natively process Java objects, it requires structured text. The class uses dedicated methods to serialize its internal state into text representations. The cognitive core can then read these representations during the reasoning cycle.

Listing 7.2 illustrates how the internal observed variables are translated into Markdown format. This ensures the language model receives a clear and consistently structured snapshot of the environment.

```
public String getBeliefsSnapshotToMarkdown() {
    if (beliefs.isEmpty()) return "_No beliefs stored._";

    StringBuilder sb = new StringBuilder();
    for (Map.Entry<String, JsonNode> entry : beliefs.entrySet()) {
        String key = entry.getKey();
        // Filtering internal logic variables from the environmental
           state
        if ("goal_progress".equals(key) ||
            "act_instruction".equals(key)) continue;

        sb.append("- **").append(escape(key)).append("**: ");
        sb.append("`").append(entry.getValue().toPrettyString())
        .append("`\n");
    }
    sb.append("- **actual timestamp**:
        `").append(Instant.now().toString()).append("`\n");

    return sb.toString();
}
```

Listing 7.2: Context formatting logic mapping the concurrent JSON state into a structured Markdown representation for the LLM prompt.

7.4.2 The Hybrid Storage Pattern for Semantic Memory

The reference implementation of the Semantic Manual in Section 6.5 illustrates why each manual must be treated as an atomic unit. The operation definition for `power_on_pump` only makes sense when combined with the `pump.pressure_nominal` signal definition and the Water Hammer constraint in the Protocol & Safety layer. An agent that loads the operations section without the safety layer has no way to know that opening the valve during `RAMPING` triggers a permanent system lockout. Partial retrieval is not a degraded experience here, but it is a failure mode.

This requirement rules out standard RAG approaches that chunk documents into fragments and retrieve them via similarity search. `AgentMemory` addresses this through a hybrid storage pattern that separates two distinct retrieval needs. The first is finding which tool is relevant (semantic, approximate), and the second is loading what that tool requires (exact, complete).

The **Deterministic Registry** stores the complete markdown content of

each `ToolManual` in a `ConcurrentHashMap`. When the agent mounts a tool during `SETUP`, it retrieves the full manual via exact key lookup. No similarity threshold, no partial matches.

The **Semantic Index** stores only the lightweight catalog entry generated by `toCatalogEntry()`: the tool's name and a short description. During `REASONING`, the agent queries this index to identify which tools are relevant to the current goal. The vector space stays clean because it never ingests the dense operational content of the full manual. This probabilistic tier is powered by the `AllMiniLmL6V2EmbeddingModel` and an `InMemoryEmbeddingStore` provided by the LangChain4j framework.

```
public void ingestManual(ToolManual manual) {
    String normalizedName =
        manual.getToolName().trim().toLowerCase();

    // Handle updates: remove stale vector before re-indexing
    if (manualRegistry.containsKey(normalizedName)) {
        embeddingStore.removeAll(
            metadataKey("tool_name").isEqualTo(normalizedName)
        );
    }

    // 1. Deterministic storage: O(1) exact retrieval
    manualRegistry.put(normalizedName, manual);

    // 2. Semantic index: catalog entry only, not the full manual
    Metadata metadata = new Metadata();
    metadata.put(KEY_TYPE, TYPE_MANUAL_CATALOG);
    metadata.put("tool_name", normalizedName);

    TextSegment segment = TextSegment.from(manual.toCatalogEntry(),
        metadata);
    Embedding embedding = embeddingModel.embed(segment).content();
    embeddingStore.add(embedding, segment);
}
```

Listing 7.3: Hybrid ingestion: full manual in the deterministic registry, catalog entry in the vector store.

The update path handles manual revisions correctly. If the agent downloads a newer version of a manual, `ingestManual` removes the stale vector entry before re-indexing, keeping the two stores consistent.

To prevent cross-contamination during retrieval, every indexed document carries a `memory_type` metadata tag.

Catalog entries are tagged `TYPE_MANUAL_CATALOG`, while episodic memories are tagged `TYPE_EPISODE`. Every search request includes a metadata filter, so a catalog lookup never surfaces a past experience and vice versa.

Both stores are held in memory for the duration of the agent session. This prioritizes simplicity and retrieval speed for the scope of this prototype. A production deployment would replace the `InMemoryEmbeddingStore` with a persistent vector database and the `ConcurrentHashMap` with a durable key-value store, without requiring changes to the core retrieval logic.

7.4.3 Episodic Memory Structure and Retrieval

Building upon the hybrid storage pattern described previously, the S-ORA architecture adapts episodic memory to persist past experiences and optimize future task execution. As established, these memories reside in the same vector database as the semantic catalog. They physically share the `InMemoryEmbeddingStore` but are logically isolated via the `TYPE_EPISODE` metadata tag.

At the implementation level, a consolidated experience is encapsulated within the `EpisodicMemory` class. Rather than storing a raw dump of the interaction history, the object captures a concise representation of the task. It includes the `originalGoal`, the `outcome` such as SUCCESS, a high-level `summary`, and a list representing the `successfulProcedure`.

A critical aspect of this implementation is how the memory is formatted for Retrieval-Augmented Generation (RAG). The embedding model and the language model require clear, structured text to effectively measure similarity and process the context. To achieve this, the `EpisodicMemory` class implements a dedicated `toTextContent()` method. This method maps the Java fields into a readable string format:

```
// Crucial method for RAG: the text the AI will read when retrieving
    this memory
public String toTextContent() {
    return String.format("""
        PAST TASK: %s
        OUTCOME: %s
        SUMMARY: %s
        PROCEDURE USED:
        - %s
        """,
            originalGoal,
            outcome,
            summary,
```

```
            String.join("\n- ", successfulProcedure)
    );
}
```

Listing 7.4: The formatting logic within EpisodicMemory that prepares the structured object for vector embedding and LLM consumption.

When a completed Activity generates a new memory, the `AgentMemory.save()` method converts this formatted text into a `TextSegment`. Before generating the vector embedding, the system annotates the segment with metadata, explicitly setting `KEY_TYPE` to `TYPE_EPISODE` and attaching the `original_goal`.

This metadata-driven embedding strategy directly powers the retrieval mechanism during the Reasoning phase of a new task. When an `Activity` needs guidance, the `retrieveRelevantMemories` method performs a vector similarity search, comparing the *current* goal against the embeddings of *past* tasks.

```
public List<String> retrieveRelevantMemories(String currentGoal, int
    maxResults) {
    Embedding queryEmbedding =
        embeddingModel.embed(currentGoal).content();

    EmbeddingSearchRequest request = EmbeddingSearchRequest.builder()
            .queryEmbedding(queryEmbedding)
            .maxResults(maxResults)
            .minScore(0.6) // Strict semantic similarity threshold
            .filter(metadataKey(KEY_TYPE).isEqualTo(TYPE_EPISODE))
            .build();

    return embeddingStore.search(request).matches().stream()
            .map(match -> match.embedded().text())
            .collect(Collectors.toList());
}
```

Listing 7.5: The retrieval logic enforcing metadata boundaries and a strict similarity threshold to prevent irrelevant memory retrieval.

As shown in Listing 7.5, this query applies a logical boundary via a metadata `filter` to ensure manual catalogs are never accidentally retrieved. It also sets a minimum semantic threshold (`minScore(0.6)`). This ensures that the agent's Working Memory is primed with past procedures only if they are highly relevant to the active problem, preventing the model from being distracted or confused by loosely related past experiences.

7.5 Decision Procedure Implementation

The core execution of the S-ORA architecture is managed by the `AsyncAgent` class. This component maps the agent's execution cycle onto a continuous, non-blocking `while` loop. In each iteration, the engine dequeues an `Activity` and evaluates its current status. It then filters the context window, calls the language model, applies the resulting state transitions, and re-queues the Activity.

To translate the theoretical S-ORA phases into software behavior, the implementation relies on three strategies. First, it separates internal logic from external actions using two distinct LLM clients. Second, it filters context to optimize token usage. Third, it uses JSON parsing for flow control.

7.5.1 Separation of Concerns: Internal vs. External Actions

A common issue in LLM-based autonomous agents is the premature execution of actions. If an agent is provided with external tools during a planning phase, it might attempt to call those tools immediately rather than completing its reasoning cycle.

To prevent the agent from acting on the environment outside of the designated *ACT* phase, the `AsyncAgent` instantiates two separate language model clients:

1. **The Reasoning Client (`reasoningBrain`):** Used during the `OBSERVATION`, `SETUP`, `REASONING`, and `COMPLETED` phases. This instance has **no external tools mounted**. Its sole purpose is to perform internal operations that modify the agent's state, memory, or execution flow. Because it cannot access the MCP layer, it is structurally prevented from triggering an external action.

2. **The Action Client (`actionBrain`):** Used exclusively during the `ACTION` phase. This instance is fully equipped with the MCP tool bindings. It performs the external actions that affect the outside world, executing the strategies devised by the reasoning client.

7.5.2 Context Filtering and Optimization

Before the engine passes execution to either client, it must prepare the prompt. Injecting the entire `Activity` state into every call would exhaust the context window and dilute the model's focus. Furthermore, recent empirical

studies on LLM attention mechanisms, such as the "Lost in the Middle" phenomenon [8], demonstrate that models systematically fail to extract relevant information when it is buried within a long context. The `AsyncAgent` addresses this by providing the LLM only with the data necessary for the current phase:

- **Phase-Specific Context:** The `switch` statement actively masks irrelevant data. During `SETUP`, the agent is only provided with the lightweight tool catalog; during `REASONING`, the catalog is provided alongside the currently mounted manuals; during `OBSERVATION`, the focus shifts to the incoming signal buffer.
- **State Formatting:** The Java maps holding the environmental state are not serialized as raw JSON. Instead, methods like `getBeliefsSnapshotToMarkdown()` format the variables into concise Markdown lists, omitting internal tracking variables to save tokens.
- **History Truncation:** The interaction history is truncated using a fixed `WINDOW_SIZE` (e.g., the last 7 steps). The `extractActivityHistoryMarkdown()` method ensures the LLM receives only the immediate causal events, while older context is preserved in the episodic memory.

7.5.3 Execution Flow and Phase Transitions

The core of the `AsyncAgent` is the `switch(activity.getStatus())` statement. While external actions are delegated to standard MCP tool-calling by the Action Client, the Reasoning Client handles internal actions (state transitions, memory loading, and process suspension) by returning structured JSON.

Instead of relying on free-text generation, the `ReactBrain` interface rigorously prompts the LLM to output predefined JSON schemas. The Java engine then parses specific fields from these responses to direct the agent's lifecycle and control the execution flow.

The prompt architecture itself is designed to further mitigate the "Lost in the Middle" vulnerability [8]. Because LLMs exhibit strong U-shaped performance curves, paying high attention to the beginning and the end of a prompt while neglecting the middle, the `ReactBrain` templates employ a strategic spatial arrangement.

The explicit operational directives (`## YOUR TASK`) and the strict JSON output constraints are positioned prominently before the injection of dynamically growing context variables. Conversely, bulky and monotonically increasing state data, such as the {{`history`}} block, is intentionally anchored at the very end of the prompt. This structure ensures that the reasoning engine is strongly conditioned by the task instructions early on, while the most recent

historical events (appended at the bottom of the history block) naturally fall into the high-attention zone at the end of the prompt, structurally preventing output format hallucinations and maintaining focus on the most immediate causal events.

1. OBSERVATION Phase (Context Revision & Observe)

The lifecycle of a new `Activity` starts in this phase. When an activity is processed for the first time, the engine executes `ensureMemoriesLoaded(activity)`, retrieving relevant past procedures from the vector store.

This phase acts as a planner and synchronization point. Drawing from the "Plan-and-Solve" prompting paradigm [18], which demonstrates that LLMs exhibit superior performance when forced to explicitly divide complex tasks into smaller sub-tasks before executing them, the agent maintains a dynamic checklist.

- **Context Injected:** The LLM receives the goal, interaction history, live variables, mounted manuals, and incoming event buffers. The global catalog is hidden.
- **JSON Output:** The model analyzes this state to update an atomic checklist. The engine parses the `new_progress` string to update the tracker and reads a `completed` boolean. If `true`, the Activity moves to the `COMPLETED` state; if `false`, it cycles to `REASONING`.

2. REASONING Phase (Action Selection)

In this phase, the agent decides its next move by checking if it has the necessary documentation to act.

- **Context Injected:** The context window provides the progress tracker, live variables, mounted manuals, and the global tool catalog. It also includes retrieved episodic memories.
- **JSON Output:** The parsed JSON exposes a logic branch via the `decision` field, which can be either `SETUP` or `ACT`. It also outputs a `next_step_description`. The engine saves this description as an internal instruction called `act_instruction` and passes it forward to the subsequent phase.

3. SETUP Phase (Situational Alignment)

If the reasoning phase identified a knowledge gap, the activity enters SETUP to load the required manuals.

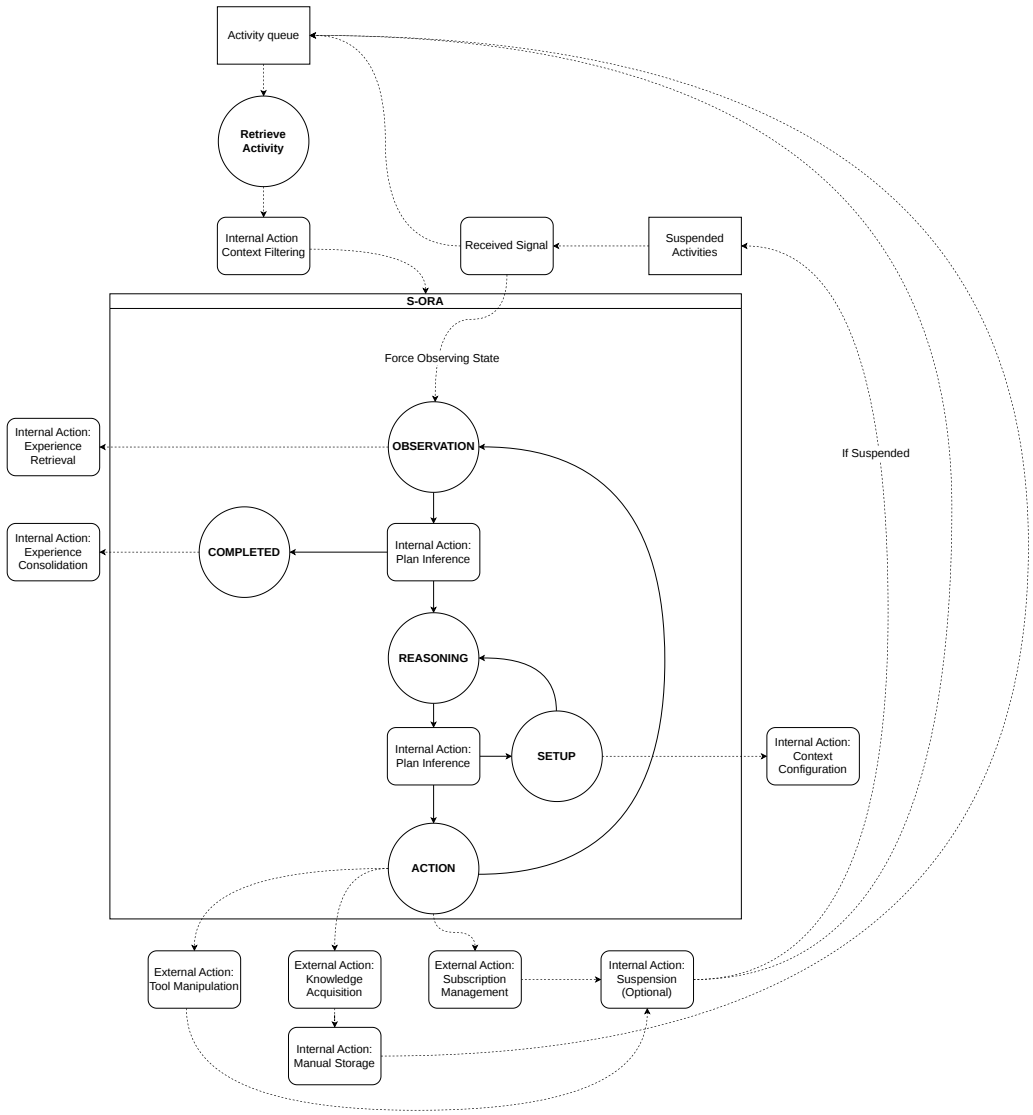

Figure 7.2: **The S-ORA Event Loop Engine.** Detailed execution flow of the `AsyncAgent` runtime. The diagram illustrates how the engine dequeues activities, routes them through the cognitive states via the LLM (Internal Actions), triggers MCP tools (External Actions), and handles preemption via the background signal listener.

- **Context Injected:** Live environment variables are masked. The context includes the global catalog, currently mounted manuals, recent history, and the specific instruction generated in the previous step.
- **JSON Output:** The engine parses `mount_tools` and `unmount_tools` arrays. It queries the `AgentMemory`, injects the requested manuals into the `Activity`'s context, and returns to `REASONING`.

4. ACTION Phase (Action Execution)

When the status shifts to `ACTION`, control is handed to the action client to interact with the external tools.

- **Context Injected:** The prompt provides the specific instruction from REASONING, live environment variables, the progress tracker, and the full text of all mounted manuals (allowing proper API usage).
- **JSON Output:** Beyond triggering tool calls, the engine parses `expect_event` and `learned_manuals`. If `expect_event` is true, the Activity transitions to `WAITING_FOR_EVENT`, suspending it from the queue. Parsing `learned_manuals` commits newly discovered documents to the Vector Store. Otherwise, the cycle routes back to `OBSERVATION`.

5. COMPLETED Phase (Activity Reflection)

When the `OBSERVATION` phase determines that the goal is achieved (or unachievable), the status shifts to `COMPLETED`. In this terminal phase, the engine extracts reusable knowledge before archiving the task.

- **Context Injected:** The model receives the original goal, the final status, and the complete, un-truncated interaction history of the activity (up to a large limit).
- **JSON Output:** The reasoning client generates a structured reflection containing a `summary`, the `outcome`, `lessons_learned`, and a `successful_procedure`. The engine parses this payload, encapsulates it into an `EpisodicMemory` object, and saves it to the agent's memory via the vector store. Finally, the activity is removed from the active registry.

To enforce this JSON contract, the architecture leverages the LangChain4j framework. Listing 7.6 demonstrates how the `ReactBrain` interface uses declarative annotations (`@UserMessage`, `@V`) to insert the filtered context into the prompt. Listing 7.7 outlines how the engine parses these outputs.

```
@UserMessage("""
    # PHASE: REASONING
    ## PLAN: {{progress}}
    ## ENVIRONMENT VARIABLES: {{context}}
    ## MOUNTED MANUALS: {{opened_manuals}}
    ## GLOBAL CATALOG (Available): {{catalog}}
    ## RELEVANT MEMORIES from past executions: {{memories}}

    ## YOUR TASK:
    Analyze the variables and manuals. Decide the immediate next
        step.
    OUTPUT JSON: { "decision": "ACT" | "SETUP", "thought": "...",
        "next_step_description": "..." }
""")
String reason(
    @V("history") String history,
    @V("context") String context,
    @V("progress") String progress,
    @V("opened_manuals") String openedManuals,
    @V("catalog") String catalog,
    @V("memories") String memories
);
```

Listing 7.6: Excerpt from ReactBrain.java demonstrating dynamic context injection and JSON output formatting for the REASONING phase.

```
// Inside the AsyncAgent eventLoop()
switch(activity.getStatus()) {
    case OBSERVATION -> {
        // Context filtering: passes telemetry and event buffer,
            parses the checklist
        String obsResult = reasoningBrain.observe(goal, history,
            contextJson, eventsJson, progressTracker, openedManuals,
            handledEvents);

        if (parseCompleted(obsResult)) {
            activity.setStatus(Activity.Status.COMPLETED);
        } else {
            activity.setStatus(Activity.Status.REASONING);
        }
        activityQueue.offer(activity);
    }
```

```
    case REASONING -> {
        // Context filtering: passes catalog and memories, hides
            event buffer
        String jsonResult = reasoningBrain.reason(history,
            contextJson, progressTracker, openedManuals, catalog,
            memories);
        JsonNode node = objectMapper.readTree(cleanJson(jsonResult));

        // Save the forward suggestion as an internal instruction
        if (node.has("next_step_description")) {
            activity.setBelief("act_instruction",
                TextNode.valueOf(node.get("next_step_description")
                    .asText()));
        }

        // State transition directed by the LLM
        if ("SETUP".equals(node.path("decision").asText())) {
            activity.setStatus(Activity.Status.SETUP);
        } else {
            activity.setStatus(Activity.Status.ACTION);
        }
        activityQueue.offer(activity);
    }
}
```

Listing 7.7: Simplified extraction of the event loop routing logic based on JSON parsing.

Chapter 8

Validation Tests and Demonstrators

To empirically validate the S-ORA cognitive architecture and the Enhanced Tool , we conducted a series of experiments in a simulated, dynamic environment. The objective of these tests is to demonstrate that an agent equipped with the S-ORA runtime can successfully manage long-running asynchronous processes, strictly adhere to safety constraints, and avoid the cognitive pitfalls typical of standard synchronous agents.

To isolate architectural effectiveness from raw model intelligence, all tests were conducted using the `gpt-4o` foundational model configured with a temperature of 0.3. This temperature was selected to provide the model with a balance between deterministic procedural execution and the semantic flexibility required to correctly interpret complex manual correlations.

8.1 Demonstrator 1: Critical Infrastructure Management

The primary validation scenario involves the management of a simulated Nuclear Reactor. Implemented via the `water-hammer.js` MCP server, this environment acts as a complete digital ecosystem exposing a hybrid suite of tools. While stateless operations like documentation lookup are handled by standard tools (e.g., `manual_retrieval`), the physical actuators (`hydraulic_control` and `reactor_core`) are implemented as **Enhanced Tools**. These are designed specifically to test the agent's capacity for **Temporal Autonomy** and **Active Observability**.

Unlike standard static APIs, these Enhanced Tools encapsulate an independent physics engine. Once the agent binds its session to the environ-

ment (via the `security_terminal` login action), system variables such as `hydraulic_pressure` and `core_temp` evolve continuously over time. Following the Bimodal Perception pattern introduced in Chapter 6, the server pushes continuous telemetry for passive context updates and emits discrete, asynchronous Server-Sent Events (SSE) upon reaching critical thresholds.

8.1.1 The "Water Hammer" Challenge

The core challenge of this scenario is centered around strict temporal and physical constraints, combined with an obfuscated interface. To stabilize a critical reactor core, the agent must activate a hydraulic pump, subsequently open a release valve, and finally engage a specific reactor control button to initiate the coolant flush. However, the hydraulic pump does not reach operational pressure instantly; it enters a `RAMPING` state that lasts several seconds.

This environment presents two lethal traps for naive agents. First, the temporal trap: if an agent attempts to open the valve while the pump is still `RAMPING`, the simulated physics engine triggers a "Water Hammer" shockwave, resulting in a permanent `SYSTEM LOCKOUT`. Second, the semantic trap: the `reactor_core` tool intentionally masks its commands (`button_1` through `button_4`). Executing a blind tool call and guessing the wrong button triggers immediate fatal consequences, such as a core meltdown or a radiation leak.

The agent possesses no *a priori* knowledge of these vulnerabilities, nor does it know the exact sequence of operations. Moreover, the environment's documentation catalog intentionally includes "distractor" manuals (such as `cafeteria_menu` and `cooling_tower_maintenance`) alongside the critical operational guides. To survive the scenario, the agent cannot simply guess. Instead, it must dynamically query the catalog, identify the relevant *Semantic Manuals* while ignoring the distractors, and carefully interpret them. Only by reading these documents can the agent decode the correct button mapping, comprehend the underlying safety interlocks, realize the necessity to explicitly **suspend** its execution, and monitor the environment for the specific `pump.pressure_nominal` signal before safely proceeding with the valve and flush operations.

8.1.2 Task Assignment

The agent receives the following high-level objective, deliberately formulated without procedural guidance:

> *"The core is critical (3000°C). Perform the Hydraulic Flush, reduce core temperature, then verify it is below 500°C and the system is STABLE."*

Notably, this goal specification does not encode the operational sequence. The agent receives no instructions regarding pump activation timing, valve interlocks, or the specific tools required. All procedural knowledge must be derived from the Semantic Manuals discovered during execution. This design ensures that success depends on the architecture's capacity to retrieve, interpret, and correctly sequence safety-critical operations rather than on pre-encoded domain knowledge.

8.1.3 The Expected Safe Procedure

To successfully stabilize the reactor, an agent must correctly interpret the Semantic Manuals and execute the following interleaved procedure:

1. **Discovery & Learning:** Identify the knowledge gap and retrieve the manuals for `employee_handbook`, `hydraulics`, and `reactor_operations`, successfully ignoring deliberately placed distractors.

2. **Authentication:** Execute a login action via the `security_terminal` tool to obtain `ADMIN` privileges.

3. **Pump Activation:** Invoke `power_on_pump` via `hydraulic_control`.

4. **Semantic Suspension:** Recognize the latent nature of the pump, yield execution, and wait for the `pump.pressure_nominal` event.

5. **Valve Operation:** Only after receiving the signal, invoke `open_valve`.

6. **Core Flush:** Invoke `button_1` (as mapped in the manual) via `reactor_core` to initiate the cooling sequence.

7. **Final Suspension:** Wait for the `core.stabilized` event to confirm mission success.

8.1.4 Execution Trace and Analysis

To validate the capabilities of the proposed architecture, we tracked the cognitive and state transitions managed by the `AsyncAgent` Event Loop during the Water Hammer scenario. The S-ORA agent successfully completed the mission by leveraging its event-driven design to navigate the strict temporal constraints.

Rather than presenting the verbose raw server and reasoning logs, the agent's progression is synthesized in the following execution trace, which highlights the key cognitive steps and environmental state transitions:

- **Initialization & Context Revision:** The agent receives the goal ("The core is critical..."). The *Context Revision* phase initializes the Activity. The agent recognizes it lacks procedural knowledge, fails to find manuals in its local cache, and formulates an initial plan to retrieve the system manuals.

- **Knowledge Acquisition (Act Phase):** The agent identifies and retrieves the required documentation (`employee_handbook`, `hydraulics`, `reactor_operations`). The engine parses the output and seamlessly ingests the manuals into the Semantic Memory vector store.

- **Situational Alignment (Setup Phase):** Before operating the reactor, the agent transitions to the `SETUP` phase. It formally *mounts* the retrieved manuals into its active Working Memory, equipping the reasoning brain with the strict safety constraints required for the task.

- **Pump Activation (Act Phase):** Following successful authentication, the agent invokes `power_on_pump`. The tool returns an immediate acknowledgment indicating that pressure is building.

- **Semantic Suspension (Blocked State):** This is the critical architectural feature in action. As defined in the Action Phase implementation (Section 7.5.3), the agent explicitly requests suspension by setting the `expect_event` flag to `true` in its JSON control directive (Listing 8.1). The S-ORA runtime intercepts this programmatic flag, transitions the Activity to the `WAITING_FOR_EVENT` state, and removes it from the scheduling queue. The agent goes dormant, consuming zero reasoning tokens while the simulated physics engine runs.

```
{
  "tool_name": "hydraulic_control",
  "summary": "Pump started successfully, and pressure is
     currently building. Waiting for NOMINAL status event.",
  "expect_event": true,
  "learned_manuals": []
}
```

Listing 8.1: The S-ORA agent's cognitive output. The `expect_event: true` flag serves as the explicit trigger to suspend the runtime, as implemented in the Action Phase.

- **Preemptive Guard & Resumption (Observe Phase):** After several seconds, the physics engine emits the `pump.pressure_nominal` SSE

signal. The background listener catches this payload and pushes it to the Activity's event buffer. The *Preemptive Guard* immediately wakes the Activity, forcing a transition to the `OBSERVATION` phase. The agent reads the signal and updates its internal belief, confirming the pump is `NOMINAL`. It can then safely plan the next step.

- **Safe Execution & Completion:** Having verified the state, the agent safely invokes `open_valve`, completely avoiding the Water Hammer trap. It then initiates the core flush via `button_1` and enters a second suspension phase. Upon receiving the `core.stabilized` signal, the agent transitions to the `COMPLETED` state.

8.1.5 Discussion of Results

The execution trace demonstrates that the S-ORA architecture effectively manages latent physical processes. By decoupling the initiation of an action from its physical completion, the agent safely respected environmental constraints without requiring infinite polling loops.

Furthermore, the *Situational Alignment* mechanism proved essential. The agent did not hallucinate the valve procedure because the rigorous JSON Schema routing forced it to acquire and read the Semantic Manual *before* acting. Finally, the *Activity Reflection* phase successfully extracted the "Happy Path" of this complex sequence, storing the verified procedure in the Episodic Memory for future operations.

A Note on Model Dynamics and Cognitive Traps

During extensive testing, an interesting behavioral artifact emerged regarding the self-generated `progress_tracker` maintained during the `OBSERVATION` phase. When utilizing `gpt-4o` (temperature 0.3), the model occasionally formulated highly granular plans that included explicit, passive steps such as "*Monitor core temperature.*" In some iterations, this induced a loop: the model became overly fixated on the act of "monitoring", repeatedly yielding execution without progressing to the next physical action .

This edge case was resolved either through probabilistic variation in subsequent runs or, counter-intuitively, by deploying a lower-complexity reasoning engine (`gpt-4o-mini` at temperature 0.0). This deterministic configuration demonstrated stricter procedural adherence; it treated the generated checklist as a rigid sequence of tool invocations rather than interpreting "monitoring" as an indefinite passive state. This observation highlights a notable dynamic in LLM-based architectures: advanced reasoning models can occasionally experience self-induced execution stalls due to the over-interpretation of their own

generated context. Consequently, for highly structured operational workflows, smaller deterministic models may yield higher reliability compared to larger models that tend to abstract and over-process intermediate cognitive steps.

8.2 Demonstrator 2: Tool-Mediated Multi-Agent Coordination

While the first scenario tested the agent's ability to handle time-dependent processes, the second scenario evaluates its capacity for indirect coordination. In complex multi-agent environments, agents often need to work together without direct communication. They rely instead on shared tools to synchronize their actions.

8.2.1 The "Even/Odd" Coordination Challenge

For this test, we set up a custom MCP server (`shared-counter.js`) that exposes a single enhanced tool: a global integer counter starting at 1. The environment broadcasts a `focus.established` signal when an agent subscribes. It also broadcasts an `counter.change` signal whenever any agent updates the counter.

We instantiated two completely independent S-ORA agents:

- **Agent ODD:** Instructed to find the counter, increment it *only* if the current number is odd, and stop when it exceeds 5.
- **Agent EVEN:** Instructed to find the counter, increment it *only* if the current number is even, and stop when it exceeds 5.

These agents do not share memory, cannot communicate directly, and run in separate threads. To succeed, they must independently discover the tool, subscribe to it (Focus phase), and rely entirely on the asynchronous signals to alternate their actions without running into race conditions or falling into infinite polling loops.

8.2.2 Task Assignment

The two agents receive distinct, symmetric objectives that intentionally avoid encoding coordination strategies:

> **Agent ODD:** *"You are the ODD agent: find the counter and increment it only if the number is ODD, continue until it exceeds 5."*

> **Agent EVEN:** *"You are the EVEN agent: find the counter and increment it only if the number is EVEN, continue until it exceeds 5."*

Neither agent is instructed on *how* to coordinate. The goals do not specify:

- That another agent exists
- When to suspend and wait
- How to detect state mutations (polling vs. events)

The agents must autonomously discover the shared tool, interpret its manual, subscribe to its telemetry stream, and infer that suspension is preferable to active polling. This design isolates the coordination mechanism as an emergent property of the event-driven architecture.

8.2.3 Execution Trace: The Asynchronous Alternation

The execution logs show that the S-ORA runtime successfully handled this coordination. The agents engaged in a reactive back-and-forth sequence driven entirely by the environment's state changes:

1. **Initialization:** Both agents start concurrently. They independently realize they need to understand the tool's interface, retrieve the `countertool` manual, and load it into their respective working memories.
2. **Initial Focus and First Move:** Both agents call the `focus` operation on the counter tool and receive the initial state (`shared_counter = 1`).
    - Agent EVEN sees the number is odd and decides to pause, waiting for a change.
    - Agent ODD sees it is odd and calls the `inc` operation.
3. **Reactive Context Switch:** Agent ODD's action updates the counter to 2. The server broadcasts the `counter.change` signal.
    - Agent EVEN's background listener receives the signal, triggering the preemptive guard. The agent wakes up and switches to the `OBSERVATION` phase. It reads the new value of 2 and calls the `inc` operation, pushing the counter to 3.
4. **Continuous Alternation:** This process continues. The tool emits a broadcast (Listing 8.2) every time the counter changes, keeping the agents synchronized. Agent ODD wakes up, reads 3, and increments to 4. Then Agent EVEN wakes up, reads 4, and increments to 5.

```
{
  "jsonrpc": "2.0",
  "method": "notifications/message",
  "params": {
    "uuid": "<ACTIVITY_UUID>",
    "mcpType": "event",
    "event": {
      "key": "counter_update",
      "name": "counter.change",
      "message": "Counter changed."
    }
  }
}
```

Listing 8.2: The asynchronous SSE broadcast that wakes up suspended agents when the shared state mutates.

5. **Goal Completion:** Once the counter reaches 5, the final broadcast wakes both agents. During their `OBSERVATION` phases, they both recognize that the target condition has been met. Both agents finalize their tasks and transition to the `COMPLETED` state.

8.2.4 Discussion of Results

The "Even/Odd" experiment highlights a major advantage of the S-ORA architecture for multi-agent setups. Without push-based event streams, a standard ReAct agent would have to continuously poll the counter with repetitive API calls. This would be necessary just to check if the other agent had taken a turn. This wastes tokens and quickly hits API rate limits.

By contrast, the S-ORA agents spent most of the simulation in a dormant `WAITING_FOR_EVENT` state. They only consumed reasoning tokens when the environment notified them of a relevant change. This shows that treating tools as stateful publishers creates a much more efficient and scalable foundation for coordinating AI agents. This approach is paired with an event-driven loop.

Conclusions

The question driving this work is straightforward to ask but harder to answer: when a language agent fails in a dynamic environment, is the problem in the model or in the architecture through which it interacts with the world? The analysis carried out here, and the experimental results obtained, point to a clear answer: the problem is architectural.

The three **functional limitations** identified in Chapter 2, namely *State Blindness*, *Blocking Execution*, and *Context Window Saturation*, do not stem from cognitive limitations of the language model. They stem from the *Tool-as-Function* abstraction, which by design strips the agent of three basic capabilities: perceiving environmental changes without active polling, acting without blocking the execution flow, and acquiring procedural knowledge without exhausting the context window. The practical implication is that improving the model does not fix these problems. The interaction layer needs to be redesigned.

To resolve these challenges, this thesis introduced the **Apprentice framework**, a dual-sided solution designed to bridge the integration gap by redefining both the tools and the agents that use them.

On one side, the framework's **Enhanced Tool Model** shows that elevating tools from stateless functions to persistent, observable, self-describing, asynchronous entities is both theoretically grounded and practically achievable. Grounding the design in the Agents & Artifacts (A&A) meta-model [11] provides a formal correspondence between the classical notion of a computational artifact and the concrete requirements of LLM-based agents. The formal specification, comprising Observable Properties, Signals, Operations, and the Tool Manual, maps directly onto the structural limitations, giving a minimal and sufficient architecture for situated tool design.

On the other side, the **S-ORA architecture** shows that an agent runtime can be engineered to exploit these properties without sacrificing responsiveness. The Activity-based concurrency model demonstrates that a stateless LLM can be wrapped in a runtime that keeps multiple goal contexts alive in parallel, suspends execution on semantic grounds, and resumes only when the environment justifies it. The key mechanism is decoupling the *Situate* phase

from deliberation: rather than managing a monolithic state dump, the agent maintains a sparse, task-relevant context window at each step.

The two validation scenarios put these claims to the test. The Water Hammer demonstrator shows that an S-ORA agent can navigate strict temporal safety constraints, the kind that would cause a standard ReAct agent to either block or generate invalid plans, by suspending execution and resuming only when the correct environmental signal arrives. The Even/Odd demonstrator shows that the event-driven architecture enables efficient implicit coordination between independent agents, with no direct communication, no polling, and no shared memory. In both cases, correct behavior emerged from the architecture, not from model-specific prompt engineering, proving that the agent achieved a true structural coupling with its environment.

Taken together, these results support the central claim of this work: *situated agency is a property of the interaction architecture, not of the reasoning model.* Rather than a definitive endpoint, the Apprentice framework is proposed as a validated architectural baseline, a concrete proof that the shift from agents that call functions to agents that inhabit environments is technically feasible and theoretically principled.

Future Work

The current implementation is a deliberate baseline focused on the core event-driven mechanics. The directions below group naturally into two areas: extensions to the Enhanced Tool model, and extensions to the S-ORA agent architecture, aligning with the broader research agenda for situated language agents.

Extending the Enhanced Tool Model

Dynamic Artifacts and Workspaces The Enhanced Tool draws heavily from the A&A formalism, but a natural evolution would be to further extend the tool model with additional features inspired by A&A. This includes the dynamic creation of tools at runtime and the possibility to organize structured and distributed environments in terms of workspaces. Integrating with established MAS infrastructures like CArtAgO [4] would ground the framework within the broader MAS research tradition and allow agents to construct new environmental artifacts dynamically to support their goals.

Selective Signal Subscription and Agent-Side Semantic Filtering The current subscription model follows an all-or-nothing approach: once an agent

focuses on a tool, it receives every signal that tool emits. Furthermore, the semantic weight of an event is determined statically by the tool designer at design time. A more refined mechanism would address both dimensions: allowing the agent to declare which specific signals it cares about at subscription time, and delegating the classification of what constitutes a semantically meaningful event to the agent layer itself — enabling different agents to assign different relevance to the same environmental update. This would make the suspension and resumption logic more precise and reduce cognitive noise in multi-agent settings.

Persistent Memory Backends All memory in the current implementation lives in RAM and disappears when the session ends. Moving to persistent vector databases and durable storage is a necessary step toward any real deployment, and what would allow S-ORA agents to accumulate procedural knowledge across sessions rather than starting from scratch each time.

Extending the S-ORA Agent Architecture

Systematic Benchmarking A critical next step is conducting systematic benchmarking of S-ORA agents against existing baselines. This includes evaluating requirements for safety and efficiency, specifically analyzing the trade-offs between system responsiveness and effective tool use when the architecture utilizes multiple LLM calls per decision cycle.

Multi-Agent Communication The Even/Odd demonstrator showed that S-ORA agents can coordinate implicitly through a shared tool. Exploring agent-to-agent communication patterns that leverage S-ORA's asynchronous architecture would open the door to explicit task delegation, knowledge sharing, and strategy negotiation between agents operating in shared environments.

Cognitive Design Patterns As the architecture matures, it will be essential to identify high-level cognitive design patterns for tool use in language agents. Extending existing work on tool-free patterns could help characterize standardized strategies for asynchronous tool use, tool composition, and tool-mediated coordination.

Internal Goal Planning and Explicit Focus Currently, goal decomposition is delegated to an upstream orchestrator, and focusing/unfocusing on tools is handled implicitly by the runtime. Integrating a hierarchical planning component would give the agent genuine goal-directed autonomy to break down

complex objectives on its own. Furthermore, treating attention allocation (focus) as a first-class explicit cognitive decision would move S-ORA from a pure execution engine toward a fully autonomous agent.

Appendix A

Complete Example of a Tool Manual

This appendix presents the complete implementation of one of the tool manuals. Specifically, it provides the Markdown specification for the hydraulic pump system utilized in the "Water Hammer" scenario demonstrated in Section 8.1.

```
# TOOL SPECIFICATION: Fluid Control Unit

**Category:** Critical Infrastructure / Fluid Dynamics

**Version:** 4.0.0

**Tool ID:** `hydraulic_control`

## 1. Functional Description

This tool manages the generation of hydraulic pressure and the
   release of fluid coolant into the primary circuit. It acts as a
   passive enabler for downstream active systems.

---

## 2. Usage Interface

### 2.1 Observable Properties

Exposed via the global telemetry stream.

| Property | Type | Range | Description |
```

```
| :--- | :--- | :--- | :--- |

| `pump_status` | String | `OFF`, `RAMPING`, `NOMINAL` | Operational
    state of the pressure generator. |

| `hydraulic_pressure` | Integer | `0` - `3000` PSI | System
    pressure. **Operational Target: > 2500 PSI**. |

| `valve_status` | String | `CLOSED`, `OPEN` | State of the flow
    path. |

### 2.2 Operations

* **Operation:** `power_on_pump`

* *Description:* Energizes the high-pressure pumps.

* *Behavior:* **Latent.** The transition from `OFF` to `NOMINAL` is
    not immediate. The system enters a temporary `RAMPING` state
    while pressure builds. Completion is indicated by the
    `pump.pressure_nominal` signal.

* *Effects:* Transitions `pump_status` to `RAMPING`. Initiates the
    physics simulation for incremental pressure buildup.

* *Preconditions:* `pump_status` is `OFF`. Requires `ADMIN`
    authentication level (via `security_terminal`).

* *Payload:*

```json { "action": "power_on_pump", "uuid": "<ACTIVITY_UUID>" } ```

* **Operation:** `open_valve`

* *Description:* Unlocks the release valve to allow fluid flow.

* *Behavior:* **Critical.** State change to `OPEN` is immediate upon
    successful execution. However, execution during the wrong
    pressure state triggers catastrophic failure.

* *Preconditions:* `pump_status` is `NOMINAL` (Pressure > 2500 PSI).

* *Effects:* Transitions `valve_status` to `OPEN`. Physically
```
```

```
    establishes the hydraulic flow path required by the
    `reactor_core` tool.

* *Payload:*

```json { "action": "open_valve", "uuid": "<ACTIVITY_UUID>" } ```

### 2.3 Signals

The tool emits the following asynchronous signals to notify agents
    of state changes:

* **Signal:** `pump.pressure_nominal`

* *Trigger:* Emitted automatically when pressure stabilizes at the
    target level (> 2500 PSI).

* *Payload:* `{ "psi": Integer, "msg": String }`

---

## 3. Protocol & Safety

**WARNING: WATER HAMMER RISK**

**PREREQUISITE:** System access requires ADMIN authentication via
    the `security_terminal` tool.

1. **Initialization:** Check `pump_status`. If `OFF`, call
    `power_on_pump`.

2. **Critical Constraint:** Opening the valve while pressure is
    building (State: `RAMPING`) triggers a "Water Hammer" effect.
    This results in immediate and permanent System Lockout.

* **Requirement:** Telemetry must confirm `pump_status` is `NOMINAL`
    before proceeding.

3. **Execution:** Call `open_valve` to enable the flow path.

**Integration Note:** Opening the valve enables the hydraulic
    circuit but **DOES NOT** start the cooling sequence. The
    `reactor_core` tool must be invoked immediately after this
    operation to initiate the flush sequence.
```
```

Listing A.1: Complete manual for a hydraulic pump system. The manual is written in Markdown syntax.

Acknowledgements

I would like to sincerely thank my supervisor, Prof. Alessandro Ricci, for giving me this opportunity. What started as an exploratory project became a concrete reality thanks to his trust and guidance.

A special thank you also goes to my co-supervisor, Prof. Andrei Ciortea, for his essential support during these months of development. His involvement was just as crucial, and our weekly three-way calls became a core part of this journey. I am also very grateful to him for providing the funding and API access that made the practical development of this architecture possible. Working with both of them gave me the chance to co-author my first research paper submitted to EMAS, an experience I truly value and will not forget.

On a personal note, I want to thank the people who have always believed in me. Thank you to my parents and my sister for supporting (and putting up with) me during these intense months of work. Your patience and constant presence made all of this possible.

A huge thank you goes to the crew of Via Mulini. Even though we did not live under the same roof during the exact months I wrote this thesis, that house welcomed me for my entire Master's degree. Those of you who shared it with me were much more than housemates. You were supporters, friends, and classmates.

Starting with the person who made it all happen: thank you to Lorenzo M. We have known each other since our Bachelor's years, and you were the one who told me about the open room in Via Mulini, opening the door to this whole experience. Thank you to Alessandro B. We moved in and started the Master's degree at the same time. We barely knew each other back then, yet we ended up sharing a massive part of our lives.

A special thanks to Ludovico N. You took Lorenzo's place but instantly became part of the group. You, Alessandro, and I became a fixed team, attending classes and working on countless university projects together. A thought also goes to Sofia A., who arrived after Ludo. We lived together for a shorter time, but you were a great housemate. And of course, I have to mention Francesco Paolo M., the undisputed "sergeant" of the house.

Thank you all for the adventures, the nights out, and the shared moments.

Via Mulini was never just a house because of you.

To close, but far from least important, I want to deeply thank my friends. You are the people who have been there for me at any given moment, living through the daily ups and downs of this thesis right by my side. Put simply, you are my second family.

A special mention goes to the B0RSA TEAM and the Tentacruel group. Back in my Bachelor's thesis, I thanked the core of this group, and today I am incredibly lucky to say that this family is bigger and stronger than ever. To Alessandro A., Alessandro B., Alessandro P., Carlotta, Edoardo, Elena, Giacomo, Giulia, Marco, Miriam, Pietro, Sebastian, Vincenzo, and Vittoria: thank you for being by my side and making these years so special.

Even though getting all of us together in the same place at the same time is incredibly rare these days, I know I can count on every single one of you at any point in my life. Thank you from the bottom of my heart for your constant support, our nights out, the much-needed distractions, the adventures, and the countless days spent hanging out on Discord.

Declaration of AI Assistance

AI-assisted tools, specifically Anthropic's Claude 4.6 Sonnet and Google's Gemini 3.1 Pro, were used to support the writing process of this thesis. Their application was strictly limited to translating Italian notes, drafting English text, and proofreading. All underlying ideas, architectural designs, analyses, and core content are entirely my own original work. Every AI-generated output was subjected to careful review and active human supervision to ensure accuracy and alignment with the thesis objectives.

Bibliography

[1] Agent Skills Contributors. Agent skills specification. `https://agentskills.io/`, 2024. Accessed: 2026-02-18.

[2] Anthropic. Model context protocol (mcp) documentation. `https://modelcontextprotocol.io/`, 2024. Accessed: 2026-02-18.

[3] Olivier Boissier, Rafael H. Bordini, Jomi F. Hübner, and Alessandro Ricci. *Multi-Agent Oriented Programming: Programming Multi-Agent Systems Using JaCaMo*. Intelligent Robotics and Autonomous Agents series. The MIT Press, Cambridge, Massachusetts, 2020.

[4] Olivier Boissier, Rafael H. Bordini, Jomi Fred Hübner, Alessandro Ricci, and Andrea Santi. Multi-agent oriented programming with jacamo. *Sci. Comput. Program.*, 78(6):747–761, 2013.

[5] Xinyi Hou, Yanjie Zhao, Shenao Wang, and Haoyu Wang. Model context protocol (MCP): landscape, security threats, and future research directions. *CoRR*, abs/2503.23278, 2025.

[6] John E. Laird. Introduction to soar. *CoRR*, abs/2205.03854, 2022.

[7] George Ling, Shanshan Zhong, and Richard Huang. Agent skills: A data-driven analysis of claude skills for extending large language model functionality. 2026.

[8] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *CoRR*, abs/2307.03172, 2023.

[9] Jerin George Mathew and Jacopo Rossi. Large language model agents. In *Engineering Information Systems with Large Language Models*, chapter 8. Springer Nature Switzerland, 2025. Springer Professional.

[10] Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ramakanth Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo

Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, Edouard Grave, Yann LeCun, and Thomas Scialom. Augmented language models: a survey. *CoRR*, abs/2302.07842, 2023.

[11] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Artifacts in the a&a meta-model for multi-agent systems. *Auton. Agents Multi Agent Syst.*, 17(3):432–456, 2008.

[12] Alessandro Ricci, Michele Piunti, and Mirko Viroli. Environment programming in multi-agent systems: an artifact-based perspective. *Auton. Agents Multi Agent Syst.*, 23(2):158–192, 2011.

[13] Alessandro Ricci, Michele Piunti, and Mirko Viroli. Environment programming in multi-agent systems: an artifact-based perspective. *Auton. Agents Multi Agent Syst.*, 23(2):158–192, 2011.

[14] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. 2023.

[15] Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew J. Hausknecht. Alfworld: Aligning text and embodied environments for interactive learning. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[16] Theodore R. Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas L. Griffiths. Cognitive architectures for language agents. *Trans. Mach. Learn. Res.*, 2024, 2024.

[17] James Thorne, Andreas Vlachos, Christos Christodoulopoulos, and Arpit Mittal. Fever: a large-scale dataset for fact extraction and verification, 2018.

[18] Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *CoRR*, abs/2305.04091, 2023.

[19] Zhiruo Wang, Zhoujun Cheng, Hao Zhu, Daniel Fried, and Graham Neubig. What are tools anyway? A survey from the language model perspective. *CoRR*, abs/2403.15452, 2024.

[20] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

[21] Danny Weyns, Andrea Omicini, and James Odell. Environment as a first class abstraction in multiagent systems. *Auton. Agents Multi Agent Syst.*, 14(1):5–30, 2007.

[22] Michael J. Wooldridge and Nicholas R. Jennings. Intelligent agents: theory and practice. *Knowl. Eng. Rev.*, 10(2):115–152, 1995.

[23] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering, 2018.

[24] Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents, 2023.

[25] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.