S-ORA: Situated Reasoning and Asynchronous Tool Use for Language Agents

Luca Tonelli[1], Alessandro Ricci[1][0000−0002−9222−5092], and Andrei Ciortea[2,3][0000−0003−0721−4135]

[1] Department of Computer Science and Engineering, Alma Mater Studiorum University of Bologna, Italy {luca.tonelli11@studio.,a.ricci@}unibo.it
[2] School of Computer Science, University of St.Gallen, Switzerland andrei.ciortea@unisg.ch
[3] Inria, Université Côte d'Azur, CNRS, I3S, France

Abstract. The ability to use tools is an essential feature of language agents. In the current tool use paradigm, however, tools are local or remote procedures invoked synchronously with respect to the agent's reasoning process—blocking further progress until a result is returned. This becomes problematic for long-running operations or concurrent tool use. Furthermore, language agents typically operate tools based on minimal descriptions that lack procedural guidance and safety constraints. In this paper, we introduce a fundamentally different tool use paradigm inspired by the Agents & Artifacts (A&A) metamodel: we model tools as domain objects with their own lifecycle and state, whose usage interface is inherently asynchronous. To support this paradigm, we introduce tool manuals—operational knowledge that complements existing tool descriptions with detailed functional specifications, procedural knowledge, and safety constraints. We also present S-ORA (Situate-Observe-Reason-Act), an architecture that operationalizes the CoALA framework and extends the ReAct cycle with two phases: Situate (learning from manuals and focusing attention on relevant tools) and Observe (perceiving environmental changes asynchronously). We demonstrate our approach through two scenarios: managing a simulated nuclear reactor with critical safety constraints and enabling tool-mediated coordination among agents. Results show that S-ORA agents equipped with tool manuals can successfully manage long-running operations, follow usage protocols and safety constraints, and coordinate through shared tools—capabilities not easily achievable with the current synchronous tool use paradigm.

Keywords: Tool Learning · Cognitive Architectures · Language Agents.

1 Introduction

Tools enable Large Language Models (LLMs) to overcome fundamental limitations, such as access to finite training data, the inability to perform precise calculations, and the tendency to hallucinate. In the literature, a *tool* is defined as an external module or function that the LLM invokes by generating the appropriate function call with its arguments, and whose output is incorporated

into the LLM's context [5, 16]. Tools are essential for enabling LLMs to interact with the external world [12, 8, 15], both to broaden their functional scope [20, 4] and enhance problem-solving capabilities [7]—making tool use a defining characteristic of *language agents* in both research [5] and practice [1].

However, the recent developments around tools for language agents overlook a research line explored in the Engineering Multi-Agent Systems (EMAS) community over the past two decades, which promotes the *environment as a first-class design and programming abstraction* in MAS [17]. Notably, the Agents and Artifacts (A&A) metamodel [6] introduces the notion of *artifact* to explicitly model tools that agents can dynamically build, discover, share, and use to support their activities. Artifacts differ fundamentally from current tools for language agents, which typically operate as synchronous function calls: they are inherently asynchronous, meaning agents remain responsive and their reasoning processes do not block while using them. This asynchronous nature is essential for long-running operations and for agents that must use multiple artifacts concurrently—capabilities that the current synchronous tool use paradigm in language agents cannot support.

To address these limitations, we make three contributions:

1. an *enhanced tool use paradigm* for language agents, inspired by A&A: we model tools as domain objects with their own lifecycle and state, whose usage is inherently asynchronous;
2. *tool manuals*: operational knowledge that complements existing tool descriptions to provide language agents with procedural guidance and safety constraints for effective and safe tool use;
3. the *S-ORA cognitive architecture*: we operationalize the CoALA conceptual framework [13], extending the ReAct cycle [21] with two new phases—*Situate*, for learning from manuals and focusing attention on relevant tools, and *Observe*, for perceiving environmental changes asynchronously.

We validate our approach through two demonstrators that test key capabilities: a simulated nuclear reactor for temporal constraints and safety-critical operations[4], and a multi-agent coordination scenario for efficient interaction through shared asynchronous tools.

This paper is structured as follows. Section 2 introduces background and related work on language agents and tool use. We present our enhanced tool use paradigm and tool manuals in Section 3, and the S-ORA cognitive architecture in Section 4. Section 5 reports on our implementation and the demonstrators.

2 Background and Related Work

We provide background on language agents, current tool use in language agents, and the environment as a first-class abstraction in engineering MAS.

[4] The nuclear reactor simulation is a toy scenario designed to emphasize specific capabilities, not a realistic application.

2.1 From LLMs to Language Agents

Large Language Models (LLMs) generate text by predicting the next token in a sequence, drawing on knowledge acquired during training. While this mechanism enables impressive capabilities and applications, LLMs are fundamentally limited by the data seen during training and their inability to interact with dynamic environments [5]. To address this, *language agents* embed LLMs within decision loops that alternate between reasoning and acting, using external tools to access real-time information, execute computations, and ground responses in environmental feedback [16].

ReAct [21] structures this decision loop by interleaving reasoning steps—internal thoughts that enable planning, goal decomposition, and error recovery—with external actions that invoke tools to retrieve results. This synergy between reasoning and acting yields consistent improvements over both pure reasoning and action-only approaches on knowledge-intensive and decision-making benchmarks. However, the ReAct loop executes tool calls synchronously: each tool invocation blocks the agent's reasoning until the tool returns a result, limiting applicability to long-running operations or scenarios requiring concurrent tool use. Moreover, ReAct provides no structured account of memory or learning mechanisms beyond the context window.

The CoALA framework [13] addresses these architectural limitations by drawing on classical cognitive architectures such as SOAR [3]. CoALA organizes language agents along three dimensions: *information storage*, structured into working, semantic, episodic, and procedural memories; *action space*, comprising internal actions for memory operations and external actions for environment interaction; and the *decision procedure*, an interactive loop for planning and execution. The S-ORA architecture operationalizes this framework while extending the ReAct cycle with two new phases to support asynchronous tool use and situated reasoning, as described in Section 4.

2.2 Tool Use in Language Agents

The dominant paradigm for tool use in language agents treats tools as synchronous function calls [12, 16]. The agent generates a structured API call, executes it, and blocks until the tool returns a result before continuing. Tools are typically categorized by their function: *perception* tools provide information (e.g., web search), *action* tools modify external state (e.g., sending an email), and *computation* tools perform calculations that LLMs cannot [16]. This synchronous tool use paradigm becomes problematic for long-running operations: when a tool call takes seconds or minutes to complete, the agent's reasoning is blocked—it cannot pursue other activities, respond to environmental changes, or manage concurrent operations. The agent must wait for each tool call to finish before proceeding.

More recently, the Model Context Protocol (MCP) [2] aims to standardize tool use across frameworks by defining a *uniform interface* for discovering and using tools and other resources (e.g., data files). Among other features, MCP allows

agents to retrieve a list of tools available on a given server, along with descriptions of their functionality and interface. By promoting the uniform discovery and use of tools, MCP facilitates the development of agents in open environments. Furthermore, MCP uses server-sent events (SSE) and HTTP streaming to connect agents with remote tools. While these channels support asynchronous data transport—enabling incremental results and progress updates—current agent frameworks typically use them only to surface progress to users. The agent's decision loop still blocks until the tool call completes before continuing—the transport is asynchronous, but the tool use paradigm is not.

2.3 Environments in Multi-Agent Systems

From an agent-oriented software engineering perspective, tools in language agents can be understood as components that are part of the application environment where the agent is logically situated, properly designed to provide services or functions to agents. In the EMAS literature, this view has been investigated by research exploring the environment as a first-class design and programming abstraction in MAS [17, 10].

Among others, the Agents and Artifacts (A&A) conceptual model introduced the concept of *artifact* as first-class abstraction to represent environmental objects [6], modelling resources and tools that agents can share and use to support their activities. An environment in A&A can be designed and programmed in terms of a dynamic set of artifacts, organised into workspaces, possibly distributed over the network [11].

In A&A, the concept of artifact was inspired by Activity Theory [9], which highlights the importance that the environment (artifacts, tools) has for humans to shape and support their activities, including both individual and collaborative ones. Accordingly, the artifact model proposed in A&A has been inspired by artifacts and tools in human environments. An artifact (tool) in A&A is an entity encapsulating some designed function, eventually having an *observable state* and a *usage interface* exposing proper affordances that allow agents to interact with it, as well as a *manual* to understand how to use it [14].

3 Empowering Tools for Language Agents

Inspired by artifacts in the A&A metamodel, we introduce an enhanced model for tools and tool learning in language agents—in which tools are not merely synchronous function calls but domain objects with their own lifecycle and state, and whose usage interface is inherently asynchronous.

We present our enhanced tool model in Section 3.1, introduce tool manuals in Section 3.2, and describe the tool usage process in Section 3.3. Tool manuals, inspired by artifact manuals in A&A, complement existing tool descriptions (e.g., MCP tool descriptions [2]) with detailed functional descriptions and procedural knowledge to support tool learning and effective tool use.

3.1 Enhanced Tool Model

We adopt the stance that tool use is inherently asynchronous for agents: an ongoing activity may be suspended while waiting for an event, but the agent itself remains responsive while waiting for a tool call to complete. In the basic tool use paradigm for language agents, *tools* are local or remote procedures that are called when generating a completion for a user query (see Section 2.2). Tool execution is synchronous with respect to the agent's reasoning process and blocks further progress until a result is returned. In our asynchronous tool use paradigm, a tool is closer to the notion of an *artifact* in A&A: it is a domain object with its own control flow and internal state, with which agents can interact through a *usage interface*. Tools exist and evolve independently of any given agent and can be shared by multiple agents. Furthermore, an agent may use multiple tools concurrently, each encapsulating different functionality.

The usage interface exposes three types of affordances: *operations*, which an agent can invoke, and *observable properties* and *signals*, which an agent can perceive. Operations represent the external actions provided by a tool. Observable properties expose a persistent (observable) state[5], while signals represent transient events that occur within tools and carry information that may be relevant to agents. Operation execution can modify observable properties and generate signals. Unlike the basic tool use paradigm, this enables language agents to monitor tool state continuously and asynchronously through observations. Operations may also return results synchronously when needed for response completion, in which case the invoking activity is suspended until the operation returns.

Observable properties represent persistent ground truth from the environment, while signals convey transient event information that may also contribute to an agent's situational awareness. We hypothesize that this design can help language agents mitigate hallucinations and make better contextual decisions by grounding their reasoning in observable environmental state rather than conversation history alone. In addition, this design improves agent efficiency by eliminating the need to repeatedly poll the environment for state relevant to ongoing activities.

3.2 Tool Manuals

Following our enhanced model, tools are domain objects with an asynchronous usage interface. Tool manuals provide agents with the procedural knowledge required to interact with these objects effectively and safely, as well as descriptions of their functionalities—that is, the purposes for which the tools were designed. Manuals complement the tool descriptions already common in protocols such as the MCP or UTCP[6], which typically include only the tool's high-level functionality and invocation instructions (see Section 2.2). Specifically, we structure manuals into six parts:

[5] Like objects in OOP, a tool (artifact) can also have a non-observable state, related to technical/implementation aspects.

[6] https://www.utcp.io/, accessed: 27.04.2026.

1. **Tool Metadata**: includes general metadata about the tool, such as category information (e.g., "Critical Infrastructure / Fluid Dynamics"), to facilitate dynamic loading into the agent's context window;
2. **Functional Description**: a short natural language description of the tool as a domain object and its intended purpose;
3. **Observable Properties**: definitions of observable properties that may populate the agent's working memory, such as the current state of an air conditioner (AC);
4. **Signals**: definitions of domain events that may be emitted by the tool, such as the AC reaching a target temperature;
5. **Operations**: definitions of commands to interact with the tool, including the commands' intended purposes, preconditions, and effects;
6. **Usage Protocols & Safety**: operating instructions, including safety constraints (if any) or conditions under which an activity must be suspended (e.g., to wait for specific signals).

This manual structure supports tool learning by instructing agents not only on what operations exist, as in current tool descriptions, but also on when and how to use them, what to expect, how to observe and interpret the tool's state, and what safety constraints to consider. For illustrative purposes, Appendix A presents a complete example of a manual for a hydraulic pump system—and Listing 1.1 shows a few extracts. The description on lines 1-3 provides a concise summary of the tool's main function and its role in the larger system. The `power_on_pump` operation on lines 9-17 describes the command to power on the pump and its effect on the pump's internal state: the transition from `OFF` to `NOMINAL` takes time, and completion is signaled via the `pump.presence_nominal` signal. This signal and how it is emitted are described on lines 21-25.

```
1  ## 1. Functional Description
2
3  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.
4
5   (...)
6
7  ### 2.2 Operations
8
9  * **Operation:** 'power_on_pump'
10
11 * *Description:* Energizes the high-pressure pumps.
12
13 * *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.
14
15 * *Preconditions:* 'pump_status' is 'OFF'. Requires 'ADMIN' authentication level (via
       'security_terminal').
16
17 * *Effects:* Transitions 'pump_status' to 'RAMPING'. Initiates the physics simulation for
       incremental pressure buildup.
18
19  (...)
20
21 ### 2.3 Signals
22
23 * **Signal:** 'pump.pressure_nominal'
```

```
24
25 * *Trigger:* Emitted automatically when pressure stabilizes at the target level (> 2500 PSI).
26
27 (...)
```

Listing 1.1: Examples of functional description, operation, and signal definitions from the manual of a hydraulic pump system in a nuclear reactor.

The pump's manual includes one usage protocol, which is shown in Listing 1.2. To operate the pump, the agent first has to be authenticated—and the manual points to the tool that should be used for this purpose (prerequisite on line 5). The protocol then specifies three sequential steps for safe operation:

1. line 7 instructs the agent to check if the state of the pump is `OFF`—and if so, to invoke the operation `power_on_pump`;
2. lines 9-11 instruct the agent to wait for the pump to reach the `NOMINAL` state; the protocol also describes the safety risk of creating a "water hammer" (i.e., hydraulic shock) if this phase is skipped; this safety risk is also highlighted through a warning on line 3 before the protocol details are given;
3. line 13 instructs the agent to invoke the `open_valve` operation (after the pump has reached the `NOMINAL` state in the previous step).

In addition, the protocol includes a note on line 15 to instruct the agent on how the pump is integrated into the larger nuclear reactor system—guiding the agent to the next tool (the `reactor_core`) required to complete the cooling sequence. This cross-tool guidance is essential for complex activities involving multiple tools.

```
1  ## 3. Protocol & Safety
2
3  **WARNING: WATER HAMMER RISK**
4
5  **PREREQUISITE:** System access requires ADMIN authentication via the 'security_terminal'
       tool.
6
7  1. ** Initialization :** Check 'pump_status'. If 'OFF', call 'power_on_pump'.
8
9  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.
10
11 * **Requirement:** Telemetry must confirm 'pump_status' is 'NOMINAL' before proceeding.
12
13 3. **Execution:** Call 'open_valve' to enable the flow path.
14
15 **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 1.2: Usage protocol for a hydraulic pump system in a nuclear reactor.

3.3 Tool Use Process

We now zoom in on the process of using enhanced tools and how it differs from the basic tool use paradigm for language agents. We restrict our discussion to tool-centric aspects—and discuss the S-ORA agent architecture in Section 4.

We break down the process of using a tool into five phases:

- **Discovery**: the agent discovers the tool at run time—for example, through MCP [2] or another tool calling protocol that supports tool discovery;
- **Learning**: the agent retrieves the tool's manual and loads it into its context; thus, the agent *learns* how to use the tool by reading its manual;
- **Focus**: the agent decides whether to subscribe to the tool's observable properties and signals to perceive relevant state changes and domain events;
- **Operation**: the agent invokes operations that return an immediate acknowledgment (but not necessarily the final result or outcome);
- **Suspension and Resumption**: the agent may decide to suspend the operation of a tool if the manual specifies waiting for a signal or observable property update before resuming.

The *discovery* phase is already supported by most tool calling protocols. It typically provides tool identifiers and short functional descriptions, enabling language agents to select tools relevant to their current goals.

The *learning* phase is specific to our tool use paradigm and extends the discovery phase: once an agent selects a relevant tool, it retrieves and reads the tool's manual to learn what the tool does, how it behaves, and how to operate it effectively and safely. This is a fundamental distinction from the basic tool use paradigm in approaches such as ReAct [21]: in our approach, agents cannot operate a tool blindly—they must first read its manual. We hypothesize that this learning phase mitigates the risk of hallucinated or misplaced tool calls by providing additional context and constraints, and helps agents derive effective courses of action from intended usage protocols.

In the *focus* phase, the agent decides whether to subscribe to the tool to perceive relevant state changes and signals. This enables the agent to selectively monitor only the parts of the environment relevant to its activities. Intentional focus is aligned with the active perception perspective [18] and is a central feature of the A&A metamodel [6]. This phase is mandated by our model for asynchronous tool use. For example, to operate the hydraulic pump as prescribed in Listing 1.2, the agent must first subscribe to the pump to monitor its operational state. The concrete subscription mechanism should be provided by the tool calling protocol. For example, our implementation extends MCP to manage such subscriptions (see Section 5 for details).

The *operation* phase is the main phase of using the tool: the agent invokes operations and perceives observable properties and signals. Unlike synchronous tool calls that block until completion, operations in our model return immediate acknowledgments confirming that the command was accepted. For example, invoking the `power_on_pump` operation shown in Listing 1.1 transitions the pump to the `RAMPING` state and returns an immediate acknowledgment, but the operation completes only when `pump.pressure_nominal` signal is emitted. The agent continues to perceive the pump's state while the operation is running.

The *suspension and resumption* phase is also specific to our asynchronous tool use model and occurs when an agent must wait for a signal or an update to an observable property before proceeding. Such an example is illustrated by the

second step of the pump's usage protocol in Listing 1.2: the protocol instructs the operating agent to wait for the `pump.pressure_nominal` signal before invoking the `open_valve` operation. The agent then suspends pump operation until the expected signal is received, thereby preventing an unsafe outcome (i.e., the water hammer). We discuss how agents manage the suspension and resumption of tool operation in the next section.

4 The S-ORA Cognitive Architecture

The enhanced tool use paradigm introduced in Section 3 is fundamentally different from the basic tool use paradigm in language agents—and requires a fundamentally different agent architecture that can balance *proactive* and *reactive* behavior, as in classical agent architectures. Specifically, the architecture should enable a language agent to pursue its activities while remaining responsive to environmental stimuli. To this end, we introduce *S-ORA (Situate-Observe-Reason-Act)*—a cognitive architecture that enables language agents to use enhanced tools asynchronously and to operate effectively in dynamic environments.

The S-ORA architecture operationalizes the CoALA framework [13] through a concrete realization of its modules for situated language agents. It also implements a decision cycle that extends ReAct's [21] reason-act loop with two key phases: *Situate* and *Observe*. In the *Situate* phase, agents orient themselves by selecting tools for the current activity, consulting their manuals, and deciding which tools they should focus on (i.e., subscribe to). In the *Observe* phase, agents perceive and interpret environmental changes through observable properties and signals to which they are subscribed. This extended cycle enables language agents to manage multiple concurrent long-running activities while maintaining awareness of their evolving environment.

We present the cognitive architecture in Section 4.1 and describe the S-ORA decision cycle in Section 4.2.

4.1 Cognitive Architecture Overview

We define the agent architecture by operationalizing the various modules in the CoALA framework [13] (see Section 2.1), as detailed in the architectural diagram in Appendix B (Figure 2). As proposed in CoALA—and, more generally, in cognitive architectures—the agent stores information in different types of memory modules. The **working memory** maintains the agent's ongoing activities, perceptual input from observable properties and signals, in-use tool manuals, and other contextual knowledge relevant to the current decision cycle. The **semantic memory** captures the agent's long-term knowledge about the world. For an S-ORA agent, this includes a catalog of available tools (their identifiers and functional descriptions) and their complete manuals. The **procedural memory** includes the implicit knowledge stored in LLM weights, which the agent can query to derive or revise a plan for its current activity. CoALA also supports explicit procedural knowledge (e.g., libraries of plans), but we focus on LLM's

parametric knowledge in this work. The **episodic memory** stores relevant experiences, such as successful activity completions, which may be retrieved for guidance in future activities.

An *activity* is the central unit of work for an S-ORA agent: it is a means to achieve a *goal* and has a *context* that represents a filtered view of the environment relevant to the activity. This context includes in-use tool manuals and activity-specific perceptual input and is stored in working memory. An agent can pursue multiple activities concurrently, where only one activity is executed in each decision cycle (see Section 4.2). It can also drop an activity if it is no longer desirable or achievable, or it can suspend the activity while waiting for external events and conditions. Figure 3 in Appendix B illustrates the life cycle of an activity, which can be in one of four states:

- *running*: the agent is reasoning or acting towards the goal;
- *blocked*: the agent is waiting for external events (e.g., signals from tools) to proceed with the activity;
- *ready*: the agent can pick and pursue the activity;
- *terminated*: the activity was completed or dropped.

The concurrent execution of activities is managed by the **decision procedure** module, which executes the agent's source code and implements the S-ORA decision cycle, discussed in detail in Section 4.2. In each cycle, this module selects one action from an action space composed of *internal* and *external* actions.

The agent uses **internal actions** to interact with its memory modules. An S-ORA agent interacts with its semantic memory to *retrieve* and *store* tool manuals. It interacts with its working memory to *load* manuals from semantic memory, *unload* manuals that are no longer needed, or *filter* observable properties and signals relevant to the current activity. The agent interacts with its procedural memory to *infer* a plan for the current activity by querying an LLM. The agent can also interact with its episodic memory to *learn* from experience by saving a summary of the successful completion of an activity or to *consult* previous experiences (i.e., activity completion summaries). The load manual, unload manual, and filter actions support the *Situate* phase of the S-ORA decision cycle, described in more detail in the next section. In addition to memory actions, an S-ORA agent can invoke the *suspend* internal action to suspend the current activity while waiting for external events.

The agent uses **external actions** to interact with its external environment. An S-ORA agent can *invoke* operations on tools, *discover* available tools (e.g., via MCP tool discovery [2]), *retrieve* manuals from external repositories (for the initial loading into semantic memory), *focus* on and *unfocus* from tools to manage subscriptions to their observable properties and signals, and *send* messages to other agents. The focus and unfocus actions support the *Observe* phase of the S-ORA decision cycle.

4.2 The S-ORA Decision Cycle

The S-ORA decision cycle manages concurrent activities by selecting one activity to progress and executing at most one external action per cycle.

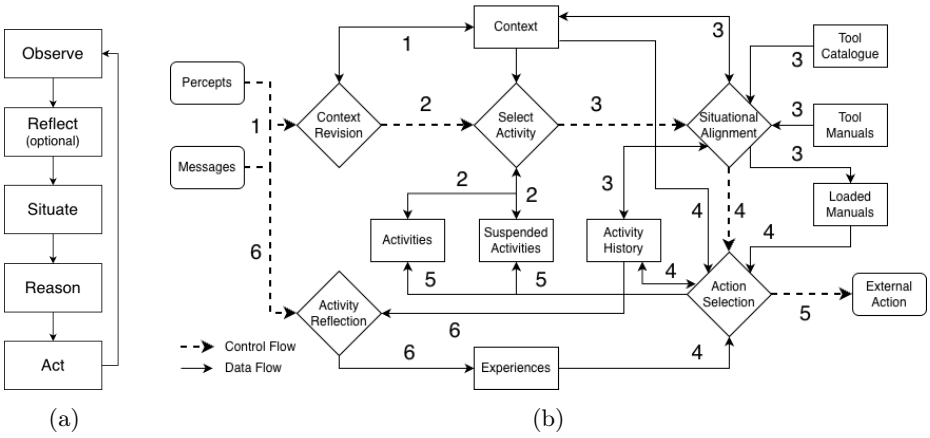

Fig. 1: The S-ORA decision cycle: (a) the four main phases of the decision loop, and (b) the six main steps of the decision procedure.

Figure 1a shows the cycle's main phases. In the **Observe** phase, the agent receives perceptual input asynchronously, where observable properties and signals are reflected passively in the agent's working memory. If the agent observes that an activity has completed successfully, it transitions to an optional **Reflect** phase, where it executes the *learn* internal action to summarize and store the successful completion in episodic memory. In the **Situate** phase, the agent selects relevant tools and adjusts its working memory—for example, by *loading* required manuals, *unloading* obsolete ones, and *filtering* the perceptual input to fit the needs of its current activity.

In the **Reasoning** phase, the agent infers a plan for the current activity—possibly revising a plan from a previous cycle—and selects the next action to advance it. The *Situate* phase may suggest prerequisite external actions for situated reasoning, such as to *retrieve* manuals from an external repository, *focus* on or *unfocus* from tools. These prerequisite actions should take priority unless a more urgent action is needed—for example, to respond to a critical signal. Otherwise, if no prerequisite or urgent actions are required, the agent selects the next external action that advances the plan, which is either to *send* a message to another agent or *invoke* a tool operation.

In the **Act** phase, the agent executes the external action selected in the *Reasoning* phase. For external actions that invoke tool operations, if the tool's manual specifies waiting for a signal or an observable property change before completion, the agent invokes the *suspend* internal action to suspend the current activity. The activity can resume once the expected external event is received.

As the S-ORA phases unfold, the decision cycle proceeds through the six main steps shown in Figure 1b. Rectangles represent the agent's internal state, stored in memory modules: the *tool catalogue* and *tool manuals* in semantic memory, *experiences* (i.e., successful activity completions) in episodic memory,

and all other state shown here is stored in working memory. Rounded boxes represent functions for receiving percepts or messages and executing external actions. Diamonds represent functions used in the S-ORA decision cycle. We describe each step as follows:

- *Step 1 (Context Revision)*: The decision cycle begins by retrieving newly received messages and the current set of percepts (defined by observable properties and signals in the environment). These are reflected in the agent's context (working memory) by the **Context Revision** function.
- *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. This function can also create a new activity—for example, when a new goal is delegated via a received message.
- *Step 3 (Situational Alignment)*: The **Situational Alignment** function adjusts the agent's working memory by selecting tools, loading/unloading manuals, and filtering observable properties and signals relevant to the selected activity.
- *Step 4 (Action Selection)*: The **Action Selection** function infers or revises a plan for the current activity to select the next action. In addition to manuals and the agent's current context, this function takes as input the **Activity History**, which is a log of all reasoning steps, actions, and results for the current activity to date. For newly created activities, it also *consults* **Experiences** from episodic memory—successful completions of similar activities that can guide planning.
- *Step 5 (Action Execution)*: The decision cycle completes by executing the external action selected in the previous step—and potentially suspending the current activity if the agent must wait for external events.
- *Step 6 (Activity Reflection)*: When the agent determines that an activity has completed successfully—for example, by observing signals and observable properties, or if all planned actions were completed successfully—it summarizes that activity's history into an experience to be stored in episodic memory and retrieved to guide future similar activities.

5 Implementation and Experience

We present our implementation of the S-ORA cognitive architecture in Section 5.1. We then demonstrate its capabilities through two scenarios: one for managing critical infrastructure (Section 5.2) and one for tool-mediated multi-agent coordination (Section 5.3). The implementation is available on GitHub.[7]

5.1 Implementation

We implemented the S-ORA cognitive architecture in Java using the LangChain4j library[8] and MCP [2]. The decision cycle uses a non-blocking event loop that

[7] https://github.com/TonelliLuca/Apprentice
[8] https://github.com/langchain4j/langchain4j

schedules concurrent activities, each maintaining its own partition in the working memory context for perceptual context, loaded manuals, and activity history. The semantic and episodic memories are implemented as in-memory (i.e., non-persistent) vector databases. The *infer* and *learn* internal actions—for synthesizing a plan and summarizing a successful activity completion, respectively—are implemented by prompting an LLM. The *filter* internal action is implicit: the engine automatically injects relevant context before each LLM call based on the activity's state. External actions are routed through MCP tool bindings. The following paragraphs describe how each step of the decision cycle in Figure 1b is implemented.

Context Revision. A background thread maintains persistent Server-Sent Events (SSE) connections to MCP tools, routing updates to activities via correlation identifiers. Observable property updates are written silently to working memory. Signals can trigger scheduling: *blocked* activities are moved to the *ready* state, while *running* activities can be interrupted to handle signals with priority. In our current implementation, we then use an LLM to interpret changes and determine whether the activity's goal has been achieved.

Activity Selection. Activities ready for selection reside in the scheduling queue, while blocked activities (waiting for signals) are temporarily removed until the expected signals arrive. Activities are selected in a round-robin fashion. Goals enter the system as new activities with unique identifiers for event routing and are immediately enqueued.

Situational Alignment. This step includes two mechanisms. First, the engine filters context before each LLM call, including only information relevant to the current activity's state. Second, the engine manages tool manuals via an LLM call: the LLM receives a catalog of available tools (from semantic memory) and currently loaded manuals, then outputs which manuals to load or unload. The engine resolves these against semantic memory and updates the current activity's working memory accordingly.

Action Selection. This step corresponds to the *infer* internal action: the LLM receives a windowed view of the activity's history (to prevent context saturation), current observations, and loaded manuals, then outputs the selected external action. Our implementation includes a fallback: if required manuals are missing from working memory, it returns to the previous step. This handles cases where the situational alignment LLM call missed required manuals, but reduces reactivity and can cause loops between the two steps (corresponding to the *Situate* and *Reason* phases)—an implementation limitation we leave for future work.

Action Execution. This step uses an LLM call to *invoke* the selected external action via MCP tools—the only step where MCP tool bindings are made available to the model for invocation. Our implementation splits external action selection (previous step) from MCP tool invocation (this step) into two LLM calls to

mitigate hallucinated MCP tool calls. The outcome of the MCP tool call is returned as structured JSON. For *retrieve* actions, the engine stores the returned manual in semantic memory. For tool operations, the response indicates whether to continue or suspend the activity to wait for signals. If suspended, the engine removes the activity from the scheduling queue.

Activity Reflection. When the `Context Revision` step determines that an activity's goal has been achieved or is no longer achievable, the activity transitions to the *terminated* state. If the activity is completed successfully, the engine prompts an LLM to consolidate the activity's full history into a structured summary containing the original goal, final outcome, and a generalized successful procedure. This summary is embedded in episodic memory as an experience.

Implementation Limitations. Our current implementation has two limitations compared to the cognitive architecture proposed in Section 4. First, multi-agent communication is not yet implemented: the system operates as a single agent that receives its initial goals programmatically. Second, the *focus* and *unfocus* internal actions are implicit: agents automatically focus on the tools they use, and subscriptions are managed by the engine rather than through explicit decisions. We intend to address both limitations in future work.

5.2 Demonstrator: Critical Infrastructure Management

The first demonstrator validates the S-ORA agent's ability to use enhanced tools asynchronously and to follow tool manuals—specifically, safety constraints.

Scenario. The agent controls a simulated nuclear reactor whose actuators (i.e., `hydraulic_control`, `reactor_core`) are implemented as enhanced tools with an independent physics simulation. System variables (e.g., `hydraulic_pressure`, `core_temp`) evolve at regular intervals once activated, with critical thresholds announced via SSE signals. The agent receives only the high-level goal:

> *"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."*

To succeed, the agent must navigate four compounding challenges: **(1) distractors**: the manual catalog includes two irrelevant manuals (`cafeteria_menu`, `cooling_tower_maintenance`) alongside three operational ones (`hydraulics`, `reactor_operations`, `employee_handbook`); **(2) authentication**: the agent must retrieve credentials from the `employee_handbook` (not provided in the goal) to activate the system; **(3) timing trap**: opening the valve while pressure builds triggers an irreversible water hammer and system lockout—the agent must wait for the `pump.pressure_nominal` signal (cf. Listing 1.2). **(4) affordance trap**: the `reactor_core` interface exposes four unlabeled buttons, of which only one initiates the coolant flush (the others trigger core meltdown, a security lockout, or a radiation containment breach). The correct mapping is encoded exclusively in the `reactor_operations` manual and cannot be inferred from the tool schema alone.

Results. The S-ORA agent successfully completed the task consistently with `gpt-4o-mini` and the temperature set to 0.0. Key execution phases: *knowledge acquisition*—the agent identified and loaded the three relevant manuals while ignoring distractors; *authentication*—after reading the `employee_handbook`, the agent retrieved the credentials, and discovered and used a `security_terminal` tool to log in, activating the physics simulation and SSE for telemetry data and signals; *suspension*—after invoking the `power_on_pump` operation of the `hydraulic_control` tool, the agent followed the tool's manual and suspended the activity waiting for a `pump.pressure_nominal` signal; *resumption*—when the signal arrived, the agent resumed and confirmed the pump was in state `NOMINAL`, safely invoked the `open_valve` operation (avoiding the water hammer), and pressed the correct button as prescribed by the `reactor_operations` manual. A second suspension followed until a `core.stabilized` signal confirmed success (as specified in the manual of the `reactor_core` tool).

Discussion. This demonstrator illustrates three key mechanisms. First, the *Situate* phase mitigates hallucination by forcing the agent to read manuals before using tools—the agent does not need to guess which button to press. Second, tool manuals enable the agent to learn both usage protocols and safety constraints—here, how to authenticate and operate the reactor while avoiding a water hammer effect and system lockout. Third, asynchronous tool use enables efficient handling of long-running operations: after invoking `power_on_pump`, the agent suspends the activity (consuming zero reasoning tokens on it while the physics simulation runs) and resumes automatically when the `pump.pressure_nominal` signal arrives, eliminating polling and respecting the physical timing constraint.

Testing revealed model sensitivity: `gpt-4o` at temperature 0.3 occasionally generated overly granular plans with explicit passive steps (e.g., *"Monitor core temperature"*), causing the agent to yield repeatedly without progressing to the next physical action. Switching to `gpt-4o-mini` at temperature 0.0 eliminated this behavior. Thorough evaluation of model sensitivity remains future work.

5.3 Demonstrator: Tool-Mediated Multi-Agent Coordination

The second demonstrator showcases tool-mediated coordination for S-ORA agents.

Scenario. An enhanced tool is shared by multiple agents and exposes an integer counter initialized to 1. Every increment triggers a `counter.change` signal perceived by all agents focused on the counter. Two S-ORA agents have the following symmetric goals:

> **Agent ODD:** *"Find the counter and increment it only if the number is ODD; stop when it exceeds 5."*
> **Agent EVEN:** *"Find the counter and increment it only if the number is EVEN; stop when it exceeds 5."*

Results. Both agents completed their tasks successfully, producing the correct alternating count sequence from 1 to 6. After retrieving the tool's manual and focusing on the tool, both agents received the initial state of 1. Agent EVEN, upon observing an odd value, suspended its activity and waited for the `counter.change` signal. Agent ODD incremented to 2, triggering the signal and waking up agent EVEN. Agent EVEN incremented to 3, waking up agent ODD in turn. This reactive alternation continued until both agents independently recognized that the terminal condition had been met during their *Observe* phases.

Discussion. This experiment demonstrates a tool-mediated coordination pattern that is difficult to achieve with synchronous tool use in language agents: the agents would need to continuously poll the counter's state—wasting resources and introducing latency due to polling intervals. Instead, S-ORA agents can use tools to coordinate efficiently: the agents remain dormant, consuming reasoning tokens only when reacting to relevant environmental changes. This pattern scales naturally to scenarios with many agents coordinating through shared tools without polling overhead. Efficiency and scalability evaluation remains future work.

6 Conclusions and Future Work

The three contributions presented in this paper—an enhanced tool model where tools are stateful domain objects, tool manuals providing operational guidance and safety constraints, and the S-ORA architecture for situated reasoning—enable language agents to manage long-running operations and use multiple tools concurrently. Unlike current synchronous tool use paradigms, in which agent reasoning blocks during tool calls, S-ORA agents suspend activities while awaiting operation completion or relevant signals, remaining responsive throughout. Our demonstrators validate that this approach supports effective and safe tool use, and allows agents to share and exploit enhanced tools to support their activities—capabilities difficult to achieve with synchronous function calls. However, this validation is preliminary: current benchmarks focus on synchronous, stateless interactions, and fully validating our approach requires developing benchmarks for asynchronous, stateful environments.

Beyond the immediate evaluation need, these contributions establish a foundation for a broader research agenda we intend to pursue. Key directions include: (i) further extending the tool model with additional features inspired by A&A, such as the dynamic creation of tools and workspace-based organization of distributed environments; (ii) conducting systematic benchmarking of S-ORA agents against existing baselines and requirements for safety and efficiency, specifically evaluating trade-offs between responsiveness and effective tool use when using multiple LLM calls per cycle; (iii) exploring agent-to-agent communication patterns that leverage S-ORA's asynchronous architecture; and (iv) identifying high-level cognitive design patterns for tool use in language agents, extending existing work on tool-free patterns [19] to characterize strategies for asynchronous tool use, tool composition, and tool-mediated coordination.

Appendix A Complete Example of a Tool Manual

This appendix presents a complete manual for a hydraulic pump system. The manual is written in Markdown syntax.

```
# 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.
```

Appendix B Architecture and Lifecycle Diagrams

This appendix provides additional visual representations of the architecture to complement the theoretical model.
```

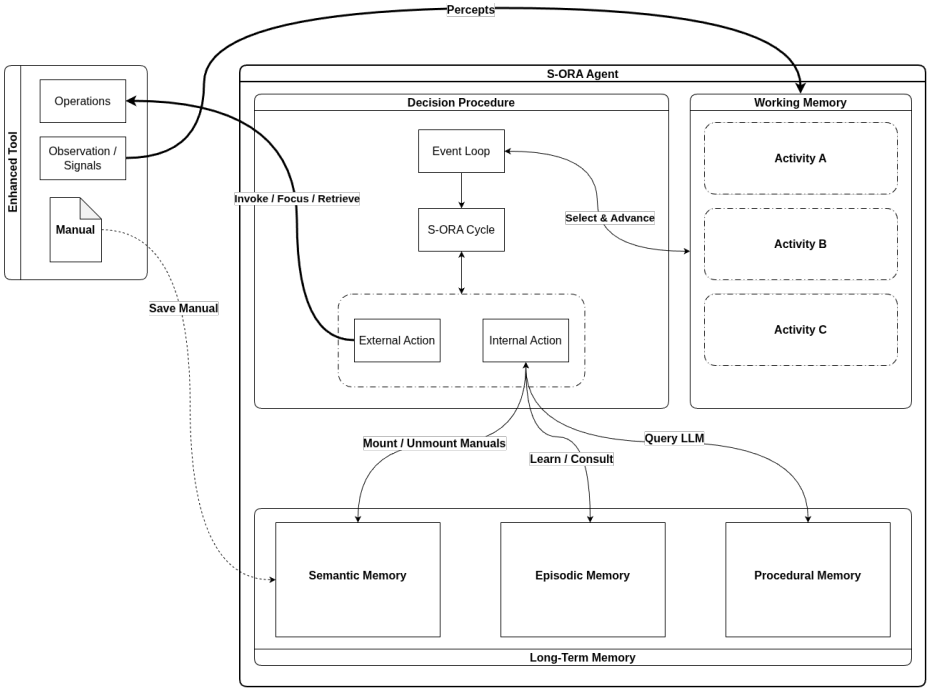

Fig. 2: **Overview of the S-ORA cognitive architecture.** The diagram illustrates the operationalization of the CoALA memory modules alongside the Event Loop engine. To support concurrent execution without state contamination, the Working Memory is structured as a federation of isolated activities. The Decision Procedure orchestrates execution by utilizing internal actions to manage memory state and external actions to interact asynchronously with the environment.

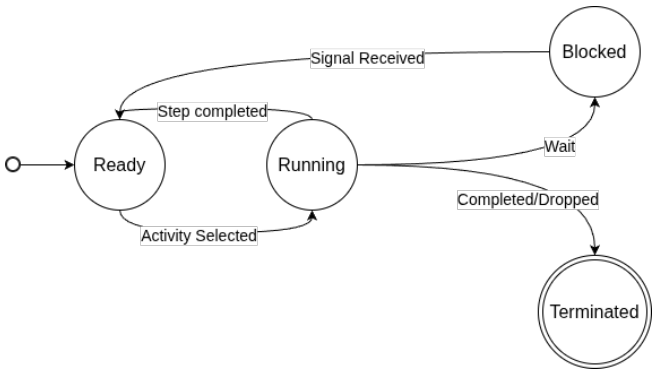

Fig. 3: **The Activity Lifecycle.** The diagram depicts the four operational states of an activity. An activity begins in the *ready* state (queued for execution). When selected by the agent, it transitions to *running*, where the S-ORA cycle advances the reasoning process. If an executed operation requires waiting for an event (e.g., a signal from an Enhanced Tool), the agent explicitly suspends the activity, transitioning it to the *blocked* state. Upon receiving the expected event, the activity is moved back to the *ready* queue. Finally, when the goal is achieved or deemed unachievable, the activity transitions to the *terminated* state, triggering activity reflection.

References

1. Anthropic: Building effective agents. https://www.anthropic.com/engineering/building-effective-agents (Dec 19 2024)
2. Anthropic: Model Context Protocol (MCP). https://modelcontextprotocol.io/ (2024), accessed: 28.04.2025
3. Laird, J.E.: The Soar cognitive architecture. MIT press (2019)
4. Lazaridou, A., Gribovskaya, E., Stokowiec, W., Grigorev, N.: Internet-augmented language models through few-shot prompting for open-domain question answering. arXiv preprint arXiv:2203.05115 (2022)
5. Mialon, G., Dessì, R., Lomeli, M., Nalmpantis, C., Pasunuru, R., Raileanu, R., Rozière, B., Schick, T., Dwivedi-Yu, J., Celikyilmaz, A., et al.: Augmented language models: a survey. arXiv preprint arXiv:2302.07842 (2023)
6. Omicini, A., Ricci, A., Viroli, M.: Artifacts in the a&a meta-model for multi-agent systems. Autonomous Agents and Multi-Agent Systems **17**(3), 432–456 (Dec 2008). https://doi.org/10.1007/s10458-008-9053-x, https://doi.org/10.1007/s10458-008-9053-x
7. Qin, Y., Hu, S., Lin, Y., Chen, W., Ding, N., Cui, G., Zeng, Z., Zhou, X., Huang, Y., Xiao, C., et al.: Tool learning with foundation models. ACM Computing Surveys **57**(4), 1–40 (2024)
8. Qin, Y., Liang, S., Ye, Y., Zhu, K., Yan, L., Lu, Y., Lin, Y., Cong, X., Tang, X., Qian, B., et al.: Toolllm: Facilitating large language models to master 16000+ real-world apis. arXiv preprint arXiv:2307.16789 (2023)
9. Ricci, A., Omicini, A., Denti, E.: Activity theory as a framework for mas coordination. In: International Workshop on Engineering Societies in the Agents World. pp. 96–110. Springer (2002)
10. Ricci, A., Piunti, M., Viroli, M.: Environment programming in multi-agent systems: an artifact-based perspective. Autonomous Agents and Multi-Agent Systems **23**(2), 158–192 (Sep 2011). https://doi.org/10.1007/s10458-010-9140-7, https://doi.org/10.1007/s10458-010-9140-7
11. Ricci, A., Piunti, M., Viroli, M.: Environment programming in multi-agent systems: an artifact-based perspective. Autonomous Agents and Multi-Agent Systems **23**(2), 158–192 (2011)
12. Schick, T., Dwivedi-Yu, J., Dessi, R., Raileanu, R., Lomeli, M., Hambro, E., Zettlemoyer, L., Cancedda, N., Scialom, T.: Toolformer: Language models can teach themselves to use tools. In: Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., Levine, S. (eds.) Advances in Neural Information Processing Systems. vol. 36, pp. 68539–68551. Curran Associates, Inc. (2023), https://proceedings.neurips.cc/paper_files/paper/2023/file/d842425e4bf79ba039352da0f658a906-Paper-Conference.pdf
13. Sumers, T.R., Yao, S., Narasimhan, K., Griffiths, T.L.: Cognitive architectures for language agents (2024), https://arxiv.org/abs/2309.02427
14. Viroli, M., Ricci, A., Omicini, A.: Operating instructions for intelligent agent coordination. The Knowledge Engineering Review **21**(1), 49–69 (2006)
15. Wang, H., Qin, Y., Lin, Y., Pan, J.Z., Wong, K.F.: Empowering large language models: Tool learning for real-world interaction. In: Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 2983–2986 (2024)
16. Wang, Z., Cheng, Z., Zhu, H., Fried, D., Neubig, G.: What are tools anyway? a survey from the language model perspective (2024), https://arxiv.org/abs/2403.15452

17. Weyns, D., Omicini, A., Odell, J.: Environment as a first class abstraction in multiagent systems. Autonomous agents and multi-agent systems **14**(1), 5–30 (2007)
18. Weyns, D., Steegmans, E., Holvoet, T.: Towards active perception in situated multi-agent systems. Applied Artificial Intelligence **18**(9-10), 867–883 (2004). https://doi.org/10.1080/08839510490509063, https://doi.org/10.1080/08839510490509063
19. Wray, R.E., Kirk, J.R., Laird, J.E.: Applying cognitive design patterns to general llm agents. In: International Conference on Artificial General Intelligence. pp. 312–325. Springer (2025)
20. Yao, S., Chen, H., Yang, J., Narasimhan, K.: Webshop: Towards scalable real-world web interaction with grounded language agents. Advances in Neural Information Processing Systems **35**, 20744–20757 (2022)
21. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., Cao, Y.: React: Synergizing reasoning and acting in language models (2023), https://par.nsf.gov/biblio/10451467