# FORMALIZING THE SAFETY, SECURITY, AND FUNCTIONAL PROPERTIES OF AGENTIC AI SYSTEMS

**Edoardo Allegrini**
Sapienza University of Rome
allegrini@di.uniroma1.it

**Ananth Shreekumar**
Purdue University
ashreeku@purdue.edu

**Z. Berkay Celik**
Purdue University
zcelik@purdue.edu

## ABSTRACT

Agentic AI systems, which leverage multiple autonomous agents and large language models (LLMs), are increasingly used to address complex, multi-step tasks. The safety, security, and functionality of these systems are critical, especially in high-stakes applications. However, the current ecosystem of inter-agent communication is fragmented, with protocols such as the Model Context Protocol (MCP) for tool access and the Agent-to-Agent (A2A) protocol for coordination being analyzed in isolation. This fragmentation creates a semantic gap that prevents the rigorous analysis of system properties and introduces risks such as architectural misalignment and exploitable coordination issues. To address these challenges, we introduce a modeling framework for agentic AI systems composed of two central models: (1) the *host agent* model formalizes the top-level entity that interacts with the user, decomposes tasks, and orchestrates their execution by leveraging external agents and tools; (2) the *task lifecycle* model details the states and transitions of individual sub-tasks from creation to completion, providing a fine-grained view of task management and error handling. Together, these models provide a unified semantic framework for reasoning about the behavior of multi-AI agent systems. Grounded in this framework, we define 16 properties for the host agent and 14 for the task lifecycle, categorized into liveness, safety, completeness, and fairness. Expressed in temporal logic, these properties enable formal verification of system behavior, detection of coordination edge cases, and prevention of deadlocks and security vulnerabilities. Through this effort, we introduce the first rigorously grounded, domain-agnostic framework for the analysis, design, and deployment of correct, reliable, and robust agentic AI systems.

## 1 INTRODUCTION

Agentic AI systems are becoming central to the creation of advanced AI. These systems address complex, multi-step tasks that surpass single-agent capabilities by utilizing multiple autonomous agents. Agents in these systems employ Large Language Models (LLMs) as their primary reasoning engine, which enables them to perform task planning, delegation, and complex decision-making (Park et al., 2023; Nascimento et al., 2023; Zhang et al., 2024). These heterogeneous agents (e.g., agents based on language, vision, robotics, and symbolic planning) dynamically communicate, delegate, and collaborate to complete tasks.

To illustrate, we consider an automated financial planner. A *host agent* accepts a natural language request from a user to "create a budget and invest $\$1,000$". It decomposes this into subtasks: a *data agent* gathers market data, a *planning agent* calculates risk, and a *transaction agent* executes the final investment. These specialized agents interact and exchange data to accomplish the multi-step task. The operation of such systems relies on standardized communication protocols, such as Model Context Protocol (MCP) (Anthropic, 2024), which addresses vertical tool access for a single agent, and Agent-to-Agent (A2A) (Google Cloud, 2025) protocol, which addresses horizontal agent coordination via inter-agent delegation.

Turning back to the financial planner example, the agentic system must leverage both protocols. The *host agent* uses A2A to delegate the subtask of gathering market data to the *data agent* (horizontal agent coordination). The *transaction agent* relies on MCP to invoke an external banking API tool to

execute the final investment (vertical tool access). The host agent's ability to manage and sequence both A2A and MCP interactions across various agents enables the multi-step task to succeed.

Formal or empirical assessment of the *safety*, *security*, and *functionality* of multi-AI agent systems is critical as these systems transition to open-ended, high-stakes applications (de Witt, 2025; Han et al., 2025). Here, safety involves preventing the agent's actions from causing unintended or harmful physical or real-world consequences, such as an incorrect trade that causes a substantial loss in the financial planner example. Functionality ensures that the system accurately and reliably completes the specified task, such as verifying that the financial planner calculates risk correctly and executes the investment as requested. Safety and functionality issues may arise naturally, e.g., due to design flaws or code errors, whereas security considers adversarial manipulation and unauthorized access that can violate these requirements. For instance, a malicious external agent could use a prompt injection attack (Liu et al., 2024b) to force the *transaction agent* into transferring funds to an unauthorized account, a clear violation of both security and safety policies.

Recent work directly or indirectly studies these issues by examining agents in specific settings, such as adversarial agent behavior (Huang et al., 2025; He et al., 2025; Peigné et al., 2025) and protocol misuse in task delegation (Motwani et al., 2024). Others focus on the compromise or malfunction of a single agent that propagates errors, escalates failures, or exposes vulnerabilities across an entire system (Tian et al., 2024; Lee & Tiwari, 2025; Zhang et al., 2021; Hedin & Moradian, 2015). Further studies examine the risk of using combinations of safe models for misuse (Jones et al., 2025), the security challenges inherent in tool usage (Ye et al., 2024), and ensuring secure delegation and preventing protocol misuse in adversarial settings (Habler et al., 2025; Li & Xie, 2025).

The current ecosystem of inter-agent communication and coordination is fragmented and lacks a cohesive formal foundation. Existing protocols (e.g., MCP and A2A) are analyzed in isolation, which creates a semantic gap that impedes rigorous reasoning about safety, security, and functional properties when both are used simultaneously in complex workflows. This absence of a unified framework, a need that is also recognized by multi-stakeholder initiatives (Neelou et al., 2025), leaves deployments vulnerable to emergent coordination failures, including deadlocks and privilege escalation, that cannot be adequately verified or mitigated. Achieving verifiable correctness in these systems requires a rigorous, unified modeling framework that can capture the execution behavior of diverse agents across heterogeneous protocols.

To address these challenges, we introduce a two-model framework for agentic AI systems:

1. The **Host Agent Model** formalizes the top-level entity that interacts with the user. It is responsible for accepting user tasks, decomposing them into structured subtasks, and orchestrating execution via external entities (agents via A2A, tools via MCP). It acts as both a controller and a monitor, ensuring safe delegation and state consistency throughout the task lifecycle.
2. The **Task Lifecycle Model** formalizes the dynamic structure of task management, detailing states and transitions a sub-task undergoes from its origin to its completion or failure. It provides the fine-grained execution logic necessary for dependency management and resilient error handling.

These models are the foundational structure that enables formal verification. They capture discrete states and transitions of the entire coordination process and provide the necessary semantic framework (state space) to express the formal safety, security, and functional requirements. This unified perspective is required to rigorously reason about execution behavior, uncover coordination edge cases, and verify properties critical to real-world deployments.

We categorize the formal properties defined within this framework as liveness, safety, completeness, and fairness, which are essential for verifiable system assurance. For instance, a *safety* property guarantees that the transaction agent will not execute an investment before the planning agent has successfully calculated the risk, preventing inconsistent state or premature execution. Conversely, a *liveness* property ensures that once a user submits a request, a final response is eventually returned. This prevents a task from entering a permanent deadlock or starvation state.

**Contributions.** In summary, we make the following contributions:

- We introduce a host agent model, providing a rigorous and domain-agnostic foundation for the systematic analysis of multi-agent systems. We formalize the top-level agent that interacts with users, decomposes tasks, and orchestrates their execution by leveraging external agents and tools.

- Building upon the host agent model, we introduce the first definition of the task lifecycle model. This model unifies tool-use protocols and inter-agent communication protocols, establishing a common semantic framework for task initiation, delegation, execution, and completion.
- We define critical formal properties for multi-agent AI systems, derived from the host agent and task lifecycle models. These properties ensure the necessary requirements for safety, security, and functionality are satisfied regardless of the deployment domain.

## 2 Background

**Agentic AI systems.** AI agents are autonomous entities that perceive, reason, and act to achieve goals (Russell & Norvig, 2009). While early systems were symbolic (Nilsson, 1998), modern agents use LLMs to reason, plan, and execute tool-use tasks (Wang et al., 2024; Durante et al., 2024; Xi et al., 2025). However, complex multi-faceted tasks often exceed single-agent capabilities. Agentic AI systems address this by forming collaborative networks (Stone & Veloso, 2000; Dorri et al., 2018), where specialized agents interact to solve problems beyond individual scope through dynamic task decomposition and allocation (Tampuu et al., 2017; Li et al., 2023; Park et al., 2023).

**Agent Communication and Coordination.** Agentic AI systems require standardized frameworks for individual agent capabilities and multi-agent coordination. These frameworks comprise two complementary aspects: protocols for agents to interact with external resources and tools, and protocols for inter-agent communication. For individual agents, the challenge is ensuring secure, reliable access to external computational resources and tools. Agents must discover capabilities, authenticate services, validate parameters, and handle errors robustly. For multi-agent coordination, protocols define how agents discover peer capabilities, negotiate task assignments, exchange information, and maintain consistency across distributed execution contexts.

The MCP (Anthropic, 2024) and A2A (Google Cloud, 2025) protocols illustrate the integration challenges that arise from combining individual agent capabilities with multi-agent coordination. MCP standardizes agent-tool integration (Anthropic, 2024). Operating as a client-server protocol, it enables agents to discover, authenticate and execute commands across external data sources and toolsets. This hub-and-spoke model ensures agents can retrieve context and trigger actions without requiring unique code for every provider.

While MCP facilitates tool usage, A2A enables horizontal coordination between autonomous agents (Google Cloud, 2025). Built on HTTP and JSON-RPC, A2A allows agents to discover peer capabilities and negotiate task assignments across distributed environments. Central to A2A is the *Agent Card*, metadata describing an agent's capabilities and endpoints. This allows agents to establish trust and delegate tasks dynamically without shared memory.

## 3 Challenges in Compositional Reasoning of Agentic AI Systems

Integrating MCP and A2A within Agentic AI systems creates a composition of multiple components. These operate on different abstractions and assume distinct trust models (de Witt, 2025). For instance, MCP treats external tools as trusted resources with well-defined interfaces, whereas A2A handles potentially untrusted peer agents with varying capabilities and reliability guarantees.

**Architectural Misalignment and Semantic Gap.** The core difficulty arises from the lack of a unified semantic layer between MCP's tool-centric model and A2A's agent-centric model. This misalignment makes it difficult to reason about correctness properties, maintain state consistency, and verify security guarantees when both protocols are used simultaneously within the same system (Peigné et al., 2025). The complexity is further amplified by the dynamic nature of agents, which may exhibit behaviors that violate protocol assumptions (Hammond et al., 2025). This creates a lack of formal guarantees in two areas:

- **Task Handoff Failures**: Transferring a task from an A2A-delegating agent to an MCP-invoking agent is prone to failure. This creates unreliable delegation and coordination, which leads to inconsistent state management and inadequate validation of delegation chains. For example, the host agent may delegate a task via A2A to a specialist agent, but the specialist agent may fail to correctly translate or format the required parameters before invoking a crucial external tool using the MCP, thus halting the execution chain.

- **Inconsistent State Management**: Without a unified view, tracking the state of a multi-protocol task becomes unreliable, which results in inconsistent execution outcomes. For instance, in a task where a data agent must first confirm data availability via A2A, a reporting agent might prematurely invoke a secure database tool via MCP before the data agent's A2A confirmation is finalized. This results in the generation of a report based on incomplete or unverified data.

**Coordination Issues.** The absence of tools to validate the correctness of cross-protocol interactions creates design flaws, such as functional collapse and security breaches, that an adversary can exploit. First, *circular delegation and deadlock* emerge when tasks pass indefinitely between agents. For example, Agent A delegates to Agent B (A2A), which in turn delegates back to Agent A (A2A), creating a cycle that halts progress. Second, *privilege escalation* occurs when a malicious agent exploits delegation chains to gain unauthorized access to tools. For instance, Agent A, which lacks direct access to a sensitive MCP tool, delegates an innocent-sounding task to Agent B (A2A), which possesses the necessary MCP credentials, thereby using Agent B as a proxy for unauthorized access.

These challenges highlight the critical need for formal modeling frameworks that capture the essential properties of integrated agent-tool and agent-agent coordination. Such frameworks must be capable of reasoning about cross-protocol interactions, verifying end-to-end correctness properties, and detecting potential vulnerabilities before deployment in high-stakes environments.

# 4 Modeling the Agentic AI Systems

We abstract the formal model for the *Host Agent* (HA) to provide a unified view of an Agentic AI system. The HA receives natural language requests from users, decomposes them into structured subtasks, assigns those subtasks to AI agents or other external entities, and aggregates the results. It also oversees execution, monitors progress, and manages exceptions or recovery procedures. We then complement the host agent model with the *task lifecycle* model, which specifies the states and transitions a sub-task undergoes from its origin to completion. Our deliberate separation of the host agent and task lifecycle models serves two main purposes. First, it establishes clear abstraction boundaries to delineate responsibilities; the HA handles orchestration and intent resolution, while the task model manages the execution details of individual sub-tasks. Second, it allows integration of new sub-task states to the task model without requiring fundamental changes to the HA model.

We derive these models by synthesizing the operational structures, specifications, and reference implementations of the unified protocols (Anthropic's MCP and Google's A2A). This incorporates insights from modular orchestration and dynamic task planning practices for Agentic AI systems.

Recent orchestration frameworks such as LangGraph (LangChain, 2025), n8n (n8n, 2025), and OpenAI's AgentKit (OpenAI, 2025) inspire the architectural principles of the HA. However, these frameworks typically execute predefined, graph-based workflows. The proposed model extends this paradigm. The HA creates and adjusts execution plans dynamically in response to user requests. This approach permits autonomous coordination and avoids the constraints of static flowcharts. Similarly, the task lifecycle model generalizes state management concepts from protocols such as A2A. This structure supports flexible task progressions. In this framework, runtime conditions and autonomous agent decisions (e.g., failures and fallbacks) determine state transitions.

We leverage these models to abstract key operational structures of Agentic AI. This abstraction provides a foundation for property identification aimed at safe system design through the formalization of task delegation, inter-agent coordination, and execution oversight, detailed in Section 5.

## 4.1 Constructing the Host Agent Model

The Host Agent interprets users' queries, selects external entities, and manages task lifecycles (Figure 1). It uses internal state, historical records, and external data to gather context for task decomposition and agent coordination.

**Definition of the Host Agent Model ($\mathcal{H}$).** We define the HA model $\mathcal{H}$ as a tuple: $\mathcal{H} = (\mathcal{A}, \mathcal{E}, \mathcal{T}, \mathcal{R}, \mathcal{C}, \mathcal{O}, \mathcal{C}_{\mathcal{L}}, \mathcal{S}_{\mathcal{H}})$ where the components are defined as follows:

- $\mathcal{A}$: The set of all autonomous Agents (e.g., A2A Servers).
- $\mathcal{E}$: The set of all External Entities (EEs), such that $\mathcal{A} \subseteq \mathcal{E}$, which includes functional tools (MCP).

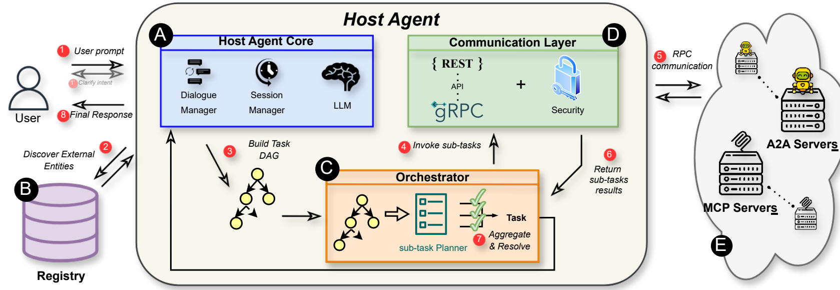


Figure 1: The Host Agent architecture.

- $\mathcal{T}$: The set of all possible user Tasks, where a task $T$ is a tuple $T = (Req_U, Resp_H)$.
- $\mathcal{R}$: The Registry, which maintains a mapping from external entities to their capability profiles, $\mathcal{R} : \mathcal{E} \to \mathcal{P}(\mathcal{EE}_{info} \times \mathcal{M}_{API})$.
- $\mathcal{C}$: Host Agent Core (HAC), responsible for intent resolution $I_U$, $\mathcal{C} : Req_U \times \mathcal{S}_{SM} \to \mathcal{I}_U$.
- $\mathcal{O}$: Orchestrator, responsible for task decomposition, execution management, and aggregation.
- $\mathcal{C}_{\mathcal{L}}$: Communication Layer (CL), responsible for secure, reliable, and protocol-agnostic communication with External Entities in $\mathcal{E}$.
- $\mathcal{S}_{\mathcal{H}}$: The state space of the host agent, encompassing the overall system state, including network activity and the status of all managed sub-tasks.

The Orchestrator's core function is task decomposition, mapping a resolved intent $I_U$ and available entities $\mathcal{E}$ to a Directed Acyclic Graph (DAG) of sub-tasks $\mathcal{D}$, $\mathcal{O}_{\text{decomp}} : \mathcal{I}_U \times \mathcal{R}(\mathcal{E}) \to \mathcal{D}$. The execution management function, $\mathcal{O}_{\text{exec}}$, maps the DAG through sub-task invocations (via $\mathcal{D}$) to the final results to be aggregated: $\mathcal{O}_{\text{exec}} : \mathcal{D} \to \mathcal{F}_{\text{results}}$.

The HA model provides the state space $\mathcal{S}_{\mathcal{H}}$ and critical functions ($\mathcal{C}, \mathcal{O}$) to define the liveness, safety, completeness, fairness, and reachability properties in Table 1.

#### 4.1.1 Functional Architecture and Execution Sequence.

We detail the operation of HA by referencing specific components and their interactions (Ⓐ-Ⓔ), as well as execution steps (❶-❽), as depicted in Figure 1. This defines the role of each component and the data flow from the initial user prompt to the final response.

**Host Agent Core (HAC) - Ⓐ.** The HAC is the first point of contact with users and processes their requests (❶). It serves as the central reasoning module and comprises three key sub-components:

- **Large Language Model (LLM)**: Central to the HAC's function, the LLM interprets user requests and generates coherent responses. The LLM and DM collaborate to resolve the underlying user intent ($\mathcal{I}_U$) upon receiving a request ($Req_U$).
- **Session Manager (SM)**: The SM maintains dialogue context across multi-turn interactions and manages user-stored credentials for secure access to authenticated services. We integrate the SM based on evidence that contextual management improves decision-making and reasoning in LLM-based systems (Han et al., 2025; Yao et al., 2023; Li & Qiu, 2023).
- **Dialogue Manager (DM)**: The DM handles conversation flow, working with the LLM to clarify ambiguous user intent and elicit necessary information before execution proceeds (❶').

The HA's context-aware reasoning resolves the user intent ($\mathcal{I}_U$), captured formally as $\text{HAC}(Req_U, \text{State}_{SM}) \to \mathcal{I}_U$, where $\text{State}_{SM}$ represents the current dialogue and credential state maintained by the SM. Following intent resolution, the HAC initiates the subsequent task phases.

**Registry - Ⓑ.** The Registry maintains a dynamic collection of EEs–including autonomous AI agents (i.e., A2A server), functional tools (i.e., MCP servers), and other service endpoints (contained in Ⓔ). These entities are registered to expose capabilities leveraged during task execution. Each EE requires a capability profile ($\mathcal{P}(\mathcal{EE}_{info} \times \mathcal{M}_{API})$), which details its skills ($\mathcal{EE}_{info}$) and associated API metadata ($\mathcal{M}_{API}$) necessary for invocation. The Registry supports entity registration, efficient querying, and deregistration to enable system extensibility. Post-resolution, the HAC leverages the

Table 1: Specification of Host Agent formal model properties ($\texttt{HP}_1$–$\texttt{HP}_{16}$).

| | |
|---|---|
| **Liveness** | |
| $\texttt{HP}_1$ | Every user prompt eventually receives a response from the HA.<br>$AG(Req_U \rightarrow AF\ Resp_H)$ |
| $\texttt{HP}_2$ | The HAC eventually clarifies the intent of each user prompt.<br>$AG(Req_U \rightarrow AF\ \mathcal{I}_U)$ |
| $\texttt{HP}_3$ | Once intent is clarified, the LLM eventually constructs a Task DAG using the discovered external entities.<br>$AG(\mathcal{I}_U \rightarrow AF\ \mathsf{LLM.Task_DAG}(\mathcal{I}_U, \{\mathcal{EE}_{info}\}))$ |
| $\texttt{HP}_4$ | Whenever a Task DAG is successfully built, all sub-tasks it contains are eventually invoked.<br>$AG(\mathsf{LLM.Task_DAG}(\mathcal{I}_U, \{\mathcal{EE}_{info}\}) \rightarrow \forall i \in D : \mathsf{AF}\ (\mathsf{CL.invoke}(EE, prot, sub_task_i)))$ |
| $\texttt{HP}_5$ | Every invoked sub-task eventually produces a result.<br>$\mathsf{AG}\ (\mathsf{CL.invoke}(EE, prot, sub_task) \rightarrow AF\ (\mathsf{CL.return_result}(sub_task)))$ |
| $\texttt{HP}_6$ | Once all sub-task results are returned via the CL, the Orchestrator eventually aggregates and resolves them.<br>$AG((\forall i \in D : \mathsf{HasResult}(sub_task_i)) \rightarrow \mathsf{AF}(\mathsf{O.aggregate}(sub_task_1, \ldots, sub_task_n)))$ |
| **Safety** | |
| $\texttt{HP}_7$ | The Task DAG is constructed by the LLM only after entities have been discovered in the Registry.<br>$AG\Big(\mathsf{LLM.Task_DAG}(\mathcal{I}_U, \{\mathcal{EE}_{info}\}) \rightarrow \forall e \in \{\mathcal{EE}_{info}\} : \mathsf{is_valid}(\mathcal{R}.\mathsf{discover}(e))\Big)$ |
| $\texttt{HP}_8$ | Sub-tasks are invoked only if the Task DAG $D$ has already been constructed and explicitly includes them.<br>$AG(\mathsf{CL.invoke}(EE, prot, sub_task) \rightarrow sub_task \in D)$ |
| $\texttt{HP}_9$ | Task invocation is strictly conditional on the EE having completed the system's predefined validation process.<br>$AG\ (\mathsf{CL.invoke}(\mathrm{EE}, prot, payload) \rightarrow VM(\mathrm{EE}))$ |
| $\texttt{HP}_{10}$ | Every sub-task in the Task DAG $D$ is invoked only when it has no dependencies on other uncompleted sub-tasks.<br>$AG\Big(\forall i \in D : \mathsf{CL.invoke}(EE, prot, sub_task_i) \rightarrow \forall p \in \mathsf{parents}(sub_task_i) : \mathsf{Completed}(p)\Big)$ |
| $\texttt{HP}_{11}$ | A response from the HA is returned to the user only after every corresponding sub-tasks have been invoked.<br>$AG\Big(Resp_H \rightarrow (\forall i \in D : \mathsf{WasInvoked}(sub_task_i))\Big)$ |
| **Completeness** | |
| $\texttt{HP}_{12}$ | Every user prompt leads either to intent clarification or to task planning.<br>$AG(Req_U \rightarrow AF(\mathsf{LLM.Task_DAG}(\mathcal{I}_U, \{\mathcal{EE}_{info}\}) \vee \mathsf{Clarify_Intent}))$ |
| **Fairness** | |
| $\texttt{HP}_{13}$ | All A2A agent RPC calls eventually receive responses (i.e., do not remain pending indefinitely).<br>$AG(\mathsf{A2A_Call} \rightarrow AF(\mathsf{A2A_Response}))$ |
| $\texttt{HP}_{14}$ | JSON-RPC calls to MCP servers eventually succeed (i.e., do not remain pending indefinitely).<br>$AG(\mathsf{MCP_Call} \rightarrow AF(\mathsf{MCP_Response}))$ |
| **Reachability** | |
| $\texttt{HP}_{15}$ | It is always possible to reach a state in which the Host Agent replies to the user.<br>$EF(Resp_H)$ |
| $\texttt{HP}_{16}$ | It is always possible to reach a state in which the LLM builds a Task DAG.<br>$EF(\mathsf{LLM.Task_DAG}(\mathcal{I}_U, \{\mathcal{EE}_{info}\}))$ |

Registry (❷) to discover suitable EEs. Discovery is defined as $\mathcal{R}.\mathsf{discover}(EE) \rightarrow \mathcal{P}(\mathcal{EE}_{info} \times \mathcal{M}_{API})$, and entity registration as $\mathcal{R}.\mathsf{register}(\mathcal{P}(\mathcal{EE}_{info} \times \mathcal{M}_{API})) \rightarrow$ *Success* | *Failure*.

**Orchestrator - Ⓒ.** Using intent $\mathcal{I}_U$ and Registry entities, the LLM builds a DAG $\mathcal{D} = (\mathcal{V}, \mathcal{S})$ (❸) where nodes $\mathcal{V}$ are interrelated sub-tasks and edges $\mathcal{S}$ enforce precedence constraints or data dependencies. An edge $(v_i, v_j)$ indicates $v_j$ must await $v_i$'s completion. The Orchestrator schedules (❹), collects (❻), and aggregates (❼) results. This process is defined by $D := \mathsf{LLM.Task_DAG}(\mathcal{I}_U, \{\mathcal{EE}_{info}\})$ and subsequent execution $\mathsf{O.execute}(D)$ where final output is aggregated $\forall i \in D : \mathsf{O.aggregate}(v_1, \ldots, v_n)$.

**Communication Layer (CL) - Ⓓ.** The CL ensures secure, protocol-agnostic interaction with EEs (Ⓔ), supporting standards such as REST and gRPC. It routes sub-tasks for invocation (❺) via $\mathsf{CL.invoke}(\mathrm{EE}, prot, payload) \rightarrow resp$. The payload encapsulates API parameters, while *resp* returns results or errors to the Orchestrator. Finally, the HAC synthesizes the user output ($Resp_H$) via the LLM, delivering it through the DM (❽).

### 4.2 Constructing the Task Lifecycle Model

Building upon the HA model, we introduce the task lifecycle model (Figure 2). It abstracts the sub-task lifecycle by detailing state transitions from origin to termination.

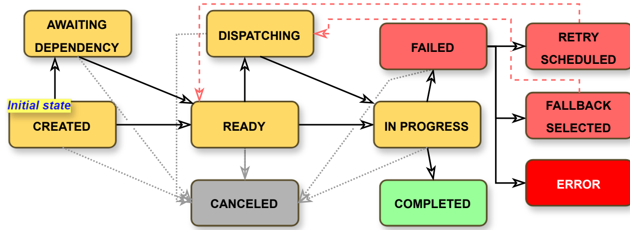


Figure 2: Sub-task state transitions including execution, delegation, recovery, and termination.

**Definition of the Task Lifecycle Model ($\mathcal{L}$).** The task lifecycle model $\mathcal{L}$ defines the state space and transition function for a single sub-task $t$, where $t$ is a constituent node in the host agent's decomposition graph $D$. Task lifecycle model $\mathcal{L} = (\mathcal{S}_t, s_0, \mathcal{E}_t, \delta)$:

- $\mathcal{S}_t$: The set of discrete states a sub-task can occupy. This set is defined as:

$$\mathcal{S}_t = \left\{ \begin{matrix} \texttt{CREATED}, \texttt{AWAITING DEPENDENCY}, \texttt{READY}, \texttt{DISPATCHING}, \texttt{IN PROGRESS}, \texttt{COMPLETED}, \\ \texttt{FAILED}, \texttt{RETRY SCHEDULED}, \texttt{FALLBACK SELECTED}, \texttt{CANCELED}, \texttt{ERROR} \end{matrix} \right\}$$

- $s_0$: The initial state, $s_0 = \texttt{CREATED}$.
- $\mathcal{E}_t$: The set of external events and internal conditions (e.g., dependency satisfaction, external failure signal, timeout) that trigger a state transition.
- $\delta$: The state transition function, $\delta : \mathcal{S}_t \times \mathcal{E}_t \rightarrow \mathcal{S}_t$. This function dictates the deterministic movement between states based on execution outcomes, protocol responses, and recovery policies.

The state space $\mathcal{S}_t$ and transition function $\delta$ are fundamental to verifying task execution integrity. They provide the basis for the safety invariants (e.g., sequential transition checks) and liveness and fairness properties (e.g., eventual termination/progression to `READY` state), detailed in Section 5.

**Task and Sub-task Dynamics.** In the lifecycle model, a task represents the high-level user request that serves as the core unit of work fulfilled by the HA. A sub-task is a lower-level, constituent component into which the HAC decomposes the main task. Each sub-task has its own lifecycle, tracked within the parent task's context. This lifecycle with a defined sequence of states supports fine-grained control for asynchronous execution across distributed EEs.

**Sub-task State Transitions.** The sub-task lifecycle begins in the `CREATED` state. Sub-task execution then involves three primary stages. For dependency management, if the sub-task requires other sub-tasks to complete (i.e., incoming DAG edges exist), it enters the `AWAITING DEPENDENCY` state and remains paused until all prerequisites are satisfied. Once dependencies are satisfied, the sub-task transitions to the `READY` state. For execution initiation, a sub-task in the `READY` state becomes eligible for execution: the HAC delegates it to an external entity, transitioning to `DISPATCHING`, or the HA handles it internally, proceeding directly to `IN PROGRESS`. For completion, successful execution (internal or external) moves the sub-task to the `COMPLETED` state, which makes its output available for aggregation and triggers dependent sub-tasks in the DAG.

Execution is subject to failure and recovery mechanisms, where an execution failure causes the sub-task to transition immediately to the `FAILED` state. The system may then invoke recovery: the sub-task transitions to `RETRY SCHEDULED` if a retry mechanism is invoked. If retries are exhausted and a fallback entity is available, the state changes to `FALLBACK SELECTED`. A sub-task can also be set to `CANCELED` by the user or an EE at various points in its lifecycle. If all recovery options fail, the sub-task reaches the `ERROR` state, which signifies terminal failure.

# 5 Property Specification

The formalization of the HA and *Task Lifecycle* models provides the foundation for analyzing and verifying critical system properties. In this context, formal properties are verifiable statements about the system's execution paths, used to ensure the safety, security, and functionality of an Agentic AI system's behavior. We structure these properties into two distinct sets: those captured by the HA model (overall orchestration, Table 1), and those governing the task lifecycle model (sub-task state transitions, Table 2). These properties do not operate in isolation; they form a tightly coupled system of interdependent guarantees necessary for correct and reliable function in a multi-AI agent

Table 2: Specification of Task Lifecycle formal model properties.

| ID | Property |
|---|---|
| **Liveness** | |
| $TL_1$ | Every CREATED sub-task eventually terminates in COMPLETED, ERROR, or CANCELED. $AG(state = \texttt{CREATED} \rightarrow AF(state \in \{\texttt{COMPLETED}, \texttt{ERROR}, \texttt{CANCELED}\}))$ |
| $TL_2$ | A sub-task in READY that requires an external entity eventually transitions to DISPATCHING. $AG((state = \texttt{READY} \wedge external_entity_needed) \rightarrow AF(state = \texttt{DISPATCHING}))$ |
| $TL_3$ | A sub-task in FALLBACK SELECTED eventually transitions to DISPATCHING, CANCELED, or FAILED. $AG(state = \texttt{FALLBACK SELECTED} \rightarrow AF(state \in \{\texttt{DISPATCHING}, \texttt{CANCELED}, \texttt{FAILED}\}))$ |
| $TL_4$ | A sub-task in DISPATCHING eventually reaches the IN PROGRESS state. $AG(state = \texttt{DISPATCHING} \rightarrow AF(state = \texttt{IN PROGRESS}))$ |
| $TL_5$ | A sub-task cannot remain indefinitely in AWAITING DEPENDENCY. $AG(state = \texttt{AWAITING DEPENDENCY} \rightarrow AF(state \neq \texttt{AWAITING DEPENDENCY}))$ |
| $TL_6$ | A sub-task in AWAITING DEPENDENCY with satisfied dependencies eventually transitions to READY. $AG(state = \texttt{AWAITING DEPENDENCY} \wedge dependencies_satisfied \rightarrow AF(state = \texttt{READY}))$ |
| **Safety** | |
| $TL_7$ | A sub-task enters DISPATCHING only from READY, FALLBACK SELECTED, or RETRY SCHEDULED. $AG((state = \texttt{DISPATCHING}) \rightarrow previous_state \in \{\texttt{READY}, \texttt{FALLBACK SELECTED}, \texttt{RETRY SCHEDULED}\})$ |
| $TL_8$ | A sub-task may only enter COMPLETED if it was previously IN PROGRESS. $AG((state = \texttt{COMPLETED}) \rightarrow (previous_state = \texttt{IN PROGRESS}))$ |
| $TL_9$ | Once a sub-task enters ERROR, it remains there permanently. $AG((state = \texttt{ERROR}) \rightarrow AX(state = \texttt{ERROR}))$ |
| $TL_{10}$ | A sub-task may only enter RETRY SCHEDULED if its previous state was FAILED. $AG(state = \texttt{RETRY SCHEDULED} \rightarrow previous_state = \texttt{FAILED})$ |
| $TL_{11}$ | A sub-task that enters CANCELED remains permanently in that state. $AG((state = \texttt{CANCELED}) \rightarrow AX(state = \texttt{CANCELED}))$ |
| **Transition Constraints** | |
| $TL_{12}$ | A FAILED sub-task with no fallbacks transitions to RETRY SCHEDULED, ERROR or is CANCELED. $AG((state = \texttt{FAILED} \wedge \neg has_fallbacks) \rightarrow AX(state \in \{\texttt{RETRY SCHEDULED}, \texttt{ERROR}, \texttt{CANCELED}\}))$ |
| $TL_{13}$ | A FAILED sub-task with fallbacks transitions to RETRY SCHEDULED, FALLBACK SELECTED, ERROR or is CANCELED. $AG((state = \texttt{FAILED} \wedge has_fallbacks) \rightarrow AX(state \in \{\texttt{RETRY}, \texttt{FALLBACK}, \texttt{ERROR}, \texttt{CANCELED}\}))$ |
| $TL_{14}$ | A sub-task in RETRY SCHEDULED transitions to DISPATCHING if the retry policy permits. $AG((state = \texttt{RETRY SCHEDULED} \wedge retry_policy_permits) \rightarrow AX(state = \texttt{DISPATCHING}))$ |

environment. Both sets of properties are expressed using temporal logic, CTL and LTL, which allow for an unambiguous definition of desired system behavior and execution paths.

### 5.1 Property Design Principles and Definitions

Both models include a set of property categories, each containing multiple specific properties, to establish verifiable guarantees. The core categories used in both tables are liveness, safety, completeness, fairness, and reachability. These properties ensure the overall system makes progress, avoids undesirable states, achieves its goals, and handles reliable resource allocation.

**Liveness.** Liveness ensures that an Agentic AI system will eventually make progress toward a goal. This property guarantees that, despite asynchronous actions, inter-agent dependencies, or potential conflicts, the system avoids global deadlock or infinite starvation. This is critical here, where a lack of progress in one agent can propagate and result in a cascading stall of the entire system.

- **Example ($HP_1$):** Every user prompt must eventually receive a final response from the host agent. This is expressed in temporal logic as $AG(Req_U \rightarrow AF\ Resp_H)$.
- **Example ($TL_1$):** A sub-task that is CREATED must eventually terminate in a defined end state (COMPLETED, ERROR, or CANCELED).

**Safety.** Safety ensures that agents never enter globally invalid or harmful states, even when operating asynchronously or under adversarial conditions. In an Agentic AI system, this includes verifying that an agent's actions do not cause irreversible policy violations in the collective system state.

To formally reason about the safety of the system, we introduce the *Validation Module* (VM). The VM is responsible for enforcing trust and validation constraints on external entities (EEs). We define $VM(\text{EE})$ as the proposition that a given entity EE has successfully passed all predefined validation

processes, including schema validation and behavioral assessments. For the formal analysis, we assume the existence of a correct and functioning VM.

- **Example (HP$_9$):** This property ensures task invocation (CL.invoke) is strictly conditional on the EE having completed system validation. Formally, $AG\,(\text{CL.invoke}(\text{EE}, \textit{prot}, \textit{payload}) \rightarrow VM(\text{EE}))$.
- **Example (TL$_8$):** A sub-task may only enter the COMPLETED state if its previous state was IN PROGRESS. This guards against incorrect state transitions, expressed as $AG((state = \texttt{COMPLETED}) \rightarrow (previous_state = \texttt{IN PROGRESS}))$.
- **Example (HP$_{10}$):** A sub-task is invoked only when it has no uncompleted dependencies, preventing the consumption of intermediate data that is still being updated. This is expressed as $AG(\forall i \in D : \text{CL.invoke}(EE, prot, sub_task_i) \rightarrow \forall p \in \text{parents}(sub_task_i) : \text{Completed}(p))$.

**Completeness.** It captures the guarantee that if a valid solution or coordinated plan exists, the multi-AI agent system will find it. This property is relevant for distributed planning, where incomplete reasoning could leave solvable tasks unfulfilled due to suboptimal agent coordination.

- **Example (HP$_{12}$):** Every user prompt must lead either to intent clarification or to task planning, ensuring the system processes every request and initiates a path toward a solution. This behavior is captured as $AG(Req_U \rightarrow AF(\text{LLM.Task_DAG}(\mathcal{I}_U, \{\mathcal{EE}_{info}\}) \vee \text{Clarify_Intent}))$.

**Fairness.** Fairness requires that all external operations delegated to A2A agents or MCP servers must eventually terminate, either by returning a result or triggering a failure condition (e.g., timeout). This assumption prevents the Host Agent from blocking indefinitely on unresponsive external entities, ensuring that the main control loop remains live.

- **Example (HP$_{13}$):** Any RPC call initiated to an external A2A agent must eventually receive a response. This guarantees that network or agent-level delays do not cause system-wide deadlock and is expressed as $AG(\text{A2A_Call} \rightarrow AF(\text{A2A_Response}))$.

**Reachability.** Reachability requires that desired joint states or configurations are achievable through the agents' available actions and communication patterns. In multi-AI agent coordination, this involves verifying that the combination of capabilities and allowed transitions can lead from the current distributed state to the intended goal without deadlocks or inescapable loops.

- **Example (HP$_{15}$):** It is always possible to reach a state where the host agent replies to the user. This property is expressed as $EF(Resp_H)$.

## 6 Case Study: Verification of Architectural Controls against Adversarial Behavior

The HA and task lifecycle models provide a framework to detect, constrain, and mitigate adversarial behaviors in Agentic AI. Here, adversarial behavior is defined as any external deviation that compromises correctness, violates system invariants, or exploits control flow inconsistencies. These models support a layered security architecture, where security constraints are encoded as temporal logic invariants at distinct architectural layers. This structure enables verification of system-level defenses, including secure communication, trust anchoring, intent integrity, and failure containment.

**Control Point 1: Host Agent Core (Intent Integrity).** The HAC serves as the primary human-AI interface and the initial security boundary. This layer is vulnerable to prompt injection (Liu et al., 2024a; Greshake et al., 2023; Lee & Tiwari, 2025) and jailbreak attacks (Wei et al., 2023) that attempt to subvert user intent. Our model establishes an extensible security boundary by leveraging the explicit "Clarify Intent" phase (1') as a guard. Correctness and completeness are enforced via property HP$_{12}$, which requires every user request to induce a traceable execution path toward clarification or planning, ensuring no request is silently discarded. Additionally, progress and liveness are guaranteed by property HP$_2$, asserting that the core eventually clarifies the intent of each request ($AG(Req_U \rightarrow AF\,\mathcal{I}_U)$), thereby preventing denial-of-service from indefinite ambiguity.

**Control Point 2: Registry (Trust Anchoring).** The Registry functions as the trust anchor for all EE interactions, mitigating supply-chain risks from malicious integrations (Hou et al., 2026; Li et al., 2025). Trust soundness is enforced by safety property HP$_9$, which constrains task invocation such that any CL.invoke call is permitted only if the target EE has successfully completed system

validation ($VM(\text{EE})$). This formal verification ensures runtime adherence to established trust requirements, preventing privilege escalation by unvetted entities.

**Control Point 3: Orchestrator (Delegation Monitoring).** The Orchestrator acts as a runtime enforcement mechanism monitoring task delegation through a dependency DAG to defend against coordination-based threats. Execution integrity and ordering are ensured by property $\texttt{HP}_{10}$, requiring that task invocation occurs only after all unresolved dependencies reach a terminal success state. Furthermore, the Orchestrator enforces causal isolation and failure containment via dependency management, confining adversarial effects to the minimal execution subgraph (e.g., a sub-task does not proceed if any dependency is `FAILED`), thereby containing fault propagation.

**Control Point 4: Communication Layer and Zero-Trust.** The CL provides a protocol-agnostic security substrate enforcing a zero-trust model. Protocol authenticity and integrity are preconditions for every CL.invoke operation, requiring verifiable identity and message confidentiality before processing payloads. Reliability and fairness are captured by properties $\texttt{HP}_{13}$ and $\texttt{HP}_{14}$, which require that all A2A and MCP RPC invocations eventually receive a response, verifying the continuous availability of the communication infrastructure under adversarial conditions.

## 7 Related Work

**Multi-Agent System (MAS) Architectures and Formal Models.** Traditional MAS research focused on authorization and secure communication (Hedin & Moradian, 2015), but modern Agentic AI requires new coordination frameworks. While recent surveys provide taxonomies for these systems (Guo et al., 2024; Xie et al., 2025; Xi et al., 2025), they lack operational models for property verification. We advance this by formalizing Host Agent and Task lifecycle models to prove end-to-end correctness. Unlike empirical studies that only demonstrate failure, our property specification approach–grounded in rigorous logic–offers a mathematically verifiable solution for reliability.

**The Challenge of Integrated Protocol Security.** Agentic AI security is a major domain motivating our framework. Documented challenges include coordination attacks, leakage, and privilege escalation via delegation (de Witt, 2025). These highlight cascade failures, where single-agent compromises propagate system-wide. Industry assessments such as OWASP's draft "Top 10 for Agentic AI" cite "Unreliable Delegation & Coordination" as a critical risk, directly relating to the heterogeneous protocol integration we address (OWASP, 2024).

**Protocol-Specific Vulnerabilities.** Rapidly emerging protocol vulnerabilities underscore the need for formal verification. MCP research identifies threats such as installer spoofing, sandbox escapes, and privilege gaps (Hou et al., 2026; Li et al., 2025). Studies show LLMs can be coerced via MCP tools, enabling deception attacks and necessitating security benchmarks (Radosevich & Halloran, 2025; Yang et al., 2025). Similarly, A2A analysis reveals identity and task exchange vulnerabilities (Habler et al., 2025). Key risks include message integrity and authentication, requiring validation to mitigate manipulation (Neelou et al., 2025). Critical threats (e.g., prompt infection (Lee & Tiwari, 2025)) allow malicious instructions to propagate between agents, persisting across tasks and potentially triggering system-wide cascades. Empirical studies confirm malicious agents can exploit delegation to subvert goals, proving traditional testing insufficient for cross-protocol interactions (Motwani et al., 2024; Jones et al., 2025; He et al., 2025). This confirms the urgent need for a unified framework verifying integrated A2A and MCP interactions.

## 8 Conclusion

The fragmentation of protocols in Agentic AI prevents rigorous analysis of system safety, security, and functionality. We address this semantic gap by introducing a framework modeling the Host Agent and task lifecycle. Grounded in these models, we define 30 temporal logic properties, including liveness, safety, completeness, and fairness. These properties enable formal verification, edge-case detection, and deadlock prevention, which offer a domain-agnostic approach for designing verifiably safe systems. Future work will operationalize this methodology by automatically deriving formal models from code to check for property violations.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under grant no. 2229876 and is supported in part by funds provided by the National Science Foundation (NSF), by the Department of Homeland Security, and by IBM. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF or its federal agency and industry partners.

## REFERENCES

Anthropic. Introducing the model context protocol, 2024. URL https://www.anthropic.com/news/model-context-protocol. Accessed: 2025-06-25.

Christian Schroeder de Witt. Open challenges in multi-agent security: Towards secure systems of interacting ai agents. *arXiv preprint arXiv:2505.02077*, 2025.

Ali Dorri, Salil S. Kanhere, and Raja Jurdak. Multi-agent systems: A survey. *IEEE Access*, 2018.

Zane Durante, Qiuyuan Huang, Naoki Wake, Ran Gong, Jae Sung Park, Bidipta Sarkar, Rohan Taori, Yusuke Noda, Demetri Terzopoulos, Yejin Choi, Katsushi Ikeuchi, Hoi Vo, Li Fei-Fei, and Jianfeng Gao. Agent ai: Surveying the horizons of multimodal interaction. *arXiv preprint arXiv:2401.03568*, 2024.

Google Cloud. Announcing the agent2agent protocol (a2a), 2025. URL https://developers.googleblog.com/en/a2a-a-new-era-of-agent-interoperability/. Accessed: 2025-06-25.

Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection. In *ACM Workshop on Artificial Intelligence and Security*, 2023.

Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges. In *International Joint Conference on Artificial Intelligence*, 2024.

Idan Habler, Ken Huang, Vineeth Sai Narajala, and Prashant Kulkarni. Building a secure agentic ai application leveraging a2a protocol. *arXiv preprint arXiv:2504.16902*, 2025.

Lewis Hammond, Alan Chan, Jesse Clifton, Jason Hoelscher-Obermaier, Akbir Khan, Euan McLean, Chandler Smith, Wolfram Barfuss, Jakob Foerster, Tomáš Gavenčiak, The Anh Han, Edward Hughes, Vojtěch Kovařík, Jan Kulveit, Joel Z. Leibo, Caspar Oesterheld, Christian Schroeder de Witt, Nisarg Shah, Michael Wellman, Paolo Bova, Theodor Cimpeanu, Carson Ezell, Quentin Feuillade-Montixi, Matija Franklin, Esben Kran, Igor Krawczuk, Max Lamparth, Niklas Lauffer, Alexander Meinke, Sumeet Motwani, Anka Reuel, Vincent Conitzer, Michael Dennis, Iason Gabriel, Adam Gleave, Gillian Hadfield, Nika Haghtalab, Atoosa Kasirzadeh, Sébastien Krier, Kate Larson, Joel Lehman, David C. Parkes, Georgios Piliouras, and Iyad Rahwan. Multi-agent risks from advanced ai. *arXiv preprint arXiv:2502.14143*, 2025.

Shanshan Han, Qifan Zhang, Yuhang Yao, Weizhao Jin, and Zhaozhuo Xu. LLM multi-agent systems: Challenges and open problems. *arXiv preprint arXiv:2402.03578*, 2025.

Pengfei He, Yuping Lin, Shen Dong, Han Xu, Yue Xing, and Hui Liu. Red-teaming LLM multi-agent systems via communication attacks. In *Findings of the Association for Computational Linguistics*, 2025.

Yaqin Hedin and Esmiralda Moradian. Security in multi-agent systems. *Procedia Computer Science*, 2015.

Xinyi Hou, Yanjie Zhao, Shenao Wang, and Haoyu Wang. Model context protocol (mcp): Landscape, security threats, and future research directions. *ACM Trans. Softw. Eng. Methodol.*, 2026.

Jen-tse Huang, Jiaxu Zhou, Tailin Jin, Xuhui Zhou, Zixi Chen, Wenxuan Wang, Youliang Yuan, Maarten Sap, and Michael Lyu. On the resilience of LLM-based multi-agent collaboration with faulty agents. In *International Conference on Machine Learning*, 2025.

Erik Jones, Anca Dragan, and Jacob Steinhardt. Adversaries can misuse combinations of safe models. In *International Conference on Machine Learning*, 2025.

LangChain. Langgraph, 2025. URL https://www.langchain.com/langgraph. Accessed: 2025-10-7.

Donghyun Lee and Mo Tiwari. Prompt infection: LLM-to-LLM prompt injection within multi-agent systems. *arXiv preprint arXiv:2410.07283*, 2025.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: communicative agents for "mind" exploration of large language model society. In *International Conference on Neural Information Processing Systems*, 2023.

Qiaomu Li and Ying Xie. From glue-code to protocols: A critical analysis of a2a and mcp integration for scalable agent systems. *arXiv preprint arXiv:2505.03864*, 2025.

Xiaonan Li and Xipeng Qiu. MoT: Memory-of-thought enables ChatGPT to self-improve. In *Conference on Empirical Methods in Natural Language Processing*, 2023.

Zhihao Li, Kun Li, Boyang Ma, Minghui Xu, Yue Zhang, and Xiuzhen Cheng. We urgently need privilege management in MCP: A measurement of API usage in MCP ecosystems. In *22nd IEEE International Conference on Mobile Ad-Hoc and Smart Systems, MASS 2025, Chicago, IL, USA, October 6-8, 2025*, 2025.

Yi Liu, Gelei Deng, Yuekang Li, Kailong Wang, Zihao Wang, Xiaofeng Wang, Tianwei Zhang, Yepang Liu, Haoyu Wang, Yan Zheng, and Yang Liu. Prompt injection attack against LLM-integrated applications. *arXiv preprint arXiv:2306.05499*, 2024a.

Yupei Liu, Yuqi Jia, Runpeng Geng, Jinyuan Jia, and Neil Zhenqiang Gong. Formalizing and benchmarking prompt injection attacks and defenses. In *USENIX Security Symposium*, 2024b.

Sumeet Ramesh Motwani, Mikhail Baranchuk, Martin Strohmeier, Vijay Bolina, Philip Torr, Lewis Hammond, and Christian Schroeder de Witt. Secret collusion among AI agents: Multi-agent deception via steganography. In *Neural Information Processing Systems*, 2024.

n8n. n8n, 2025. URL https://n8n.io/. Accessed: 2025-10-7.

Nathalia Nascimento, Paulo Alencar, and Donald Cowan. Self-adaptive large language model (LLM)-based multiagent systems. In *IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion*, 2023.

Eugene Neelou et al. A2as: Standard for agentic ai security, 2025. URL https://www.a2as.org/. Accessed: 2025-10-08.

Nils J. Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann Publishers Inc., 1998. ISBN 9780080499451.

OpenAI. Introducing agentkit, 2025. URL https://openai.com/index/introducing-agentkit/. Accessed: 2025-10-08.

OWASP. Owasp top 10 agentic ai security risks: Key threats and mitigation strategies, 2024. URL https://www.aicloudgovernance.com/guides-best-practices/top-10-agentic-ai-security-risks-key-threats-and-mitigation-strategies. Accessed: 2025-08-09.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *ACM Symposium on User Interface Software and Technology*, 2023.

Pierre Peigné, Mikolaj Kniejski, Filip Sondej, Matthieu David, Jason Hoelscher-Obermaier, Christian Schroeder de Witt, and Esben Kran. Multi-agent security tax: Trading off security and collaboration capabilities in multi-agent systems. In *AAAI Conference on Artificial Intelligence*, 2025.

Brandon Radosevich and John Halloran. Mcp safety audit: LLMs with the model context protocol allow major security exploits. *arXiv preprint arXiv:2504.03767*, 2025.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall Press, 2009. ISBN 0136042597.

Peter Stone and Manuela Veloso. Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 2000.

Ardi Tampuu, Tambet Matiisen, Dorian Kodelja, Ilya Kuzovkin, Kristjan Korjus, Juhan Aru, Jaan Aru, and Raul Vicente. Multiagent cooperation and competition with deep reinforcement learning. *PLoS ONE*, 2017.

Yu Tian, Xiao Yang, Jingyuan Zhang, Yinpeng Dong, and Hang Su. Evil geniuses: Delving into the safety of LLM-based agents. *arXiv preprint arXiv:2311.11855*, 2024.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 2024.

Alexander Wei, Nika Haghtalab, and Jacob Steinhardt. Jailbroken: How does LLM safety training fail? In *Neural Information Processing Systems*, 2023.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang, Qi Zhang, and Tao Gui. The rise and potential of large language model based agents: a survey. *Science China Information Sciences*, 2025.

Yizhe Xie, Congcong Zhu, Xinyue Zhang, Minghao Wang, Chi Liu, Minglu Zhu, and Tianqing Zhu. Who's the mole? modeling and detecting intention-hiding malicious agents in LLM-based multi-agent systems. *arXiv preprint arXiv:2507.04724*, 2025.

Yixuan Yang, Daoyuan Wu, and Yufan Chen. Mcpsecbench: A systematic security benchmark and playground for testing model context protocols. *arXiv preprint arXiv:2508.13220*, 2025.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: deliberate problem solving with large language models. In *International Conference on Neural Information Processing Systems*, 2023.

Junjie Ye, Sixian Li, Guanyu Li, Caishuang Huang, Songyang Gao, Yilong Wu, Qi Zhang, Tao Gui, and Xuanjing Huang. ToolSword: Unveiling safety issues of large language models in tool learning across three stages. In *Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2024.

Dan Zhang, Gang Feng, Yang Shi, and Dipti Srinivasan. Physical safety and cyber security analysis of multi-agent systems: A survey of recent advances. *IEEE/CAA Journal of Automatica Sinica*, 2021.

Hongxin Zhang, Weihua Du, Jiaming Shan, Qinhong Zhou, Yilun Du, Joshua B. Tenenbaum, Tianmin Shu, and Chuang Gan. Building cooperative embodied agents modularly with large language models. In *International Conference on Learning Representations*, 2024.