




**GenAI** SECURITY  
PROJECT  
TOP 10 FOR LLM AND GENERATIVE AI

# Securing Agentic Applications Guide

---

OWASP Gen AI Security Project – Agentic Security Initiative

**Version 1.0**  
July 28, 2025  
Status: Released



The information provided in this document does not, and is not intended to, constitute legal advice. All information is for general informational purposes only. This document contains links to other third-party websites. Such links are only for convenience and OWASP does not recommend or endorse the contents of the third-party sites.

#### License and Usage:

This document is licensed under Creative Commons, CC BY-SA 4.0

You are free to:


- Share – copy and redistribute the material in any medium or format
  - Adapt – remix, transform, and build upon the material for any purpose, even commercially.
  - Under the following terms:
    - Attribution – You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner but not in any way that suggests the licensor endorses you or your use.
    - Attribution Guidelines – must include the project name as well as the name of the asset Referenced
- OWASP Top 10 for LLMs - GenAI Red Teaming Guide
- Share Alike – If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

Link to full license text: <https://creativecommons.org/licenses/by-sa/4.0/legalcode>



# Table of Content

1. Introduction	4
1.1 Scope and Audience	4
2. Secure Agentic Architectures	5
2.1 Defining Key Components of Agentic Architectures	5
2.2 Attack Surface Analysis	10
2.3 Frameworks	23
2.4. Key Components, Patterns, and Frameworks	25
2.5 Case Studies	26
3. Agentic Developer Guidelines	30
Agentic AI Developer Security Guidelines: A Lifecycle Approach	30
3.1. Secure Design & Development Phase	30
3.2. Secure Build & Deployment Phase	35
3.3. Secure Operations & Runtime Phase	38
4. Enhanced Security Actions for Agentic AI Systems	46
4.1. Single-Agent Systems	46
4.2. Multi-Agent Systems with Central Orchestrator	51
4.3. Multi-Agent Systems with Swarm Architecture	54
5. Key Operational Capabilities	57
5.1. KC6.1 – API Access	57
5.2. KC6.2 – Code Execution	58
5.4. KC6.4 – Web Use	60



5.5. KC6.5 – Controlling PC Operations (Filesystem, OS Commands)	61
5.6. KC6.6 – Operating Critical Systems (e.g., SCADA controls)	62
6. Agentic AI and the Supply Chain	63
6.1 Code Security	63
6.2 Environments & Development	63
6.3 Agent & Tool Discovery	64
7. Assuring Agentic Applications	65
7.1. Red Teaming Agentic Applications:	65
7.2. Behavioral Testing for Agentic Applications:	68
8. Secure Agentic Deployments	72
9. Runtime hardening.	74
9.1 Harden the Virtual Machine (Base Level)	74
9.2 Contain the Agentic Runtime	75
9.3 Secure the Agent’s Memory, Tools, and Context	75
9.4 Observability + Forensics	76
9.6(Optional) Cloud-Specific Hardening	77
Acknowledgements	78
OWASP Top 10 for LLM Project Sponsors	79
Project Supporters	80



# 1. Introduction

This guide aims to provide practical and actionable guidance for designing, developing, and deploying secure agentic applications powered by large language models (LLMs). It complements the [OWASP Agentic AI Threats and Mitigations \(ASIT&M\) document](#) by focusing on concrete technical recommendations that builders and defenders can apply directly.

## 1.1 Scope and Audience

**Audience:** This guide is primarily intended for software developers, AI/ML engineers, security architects, security engineers, and technical leads involved in building, securing, or assessing workflow and agentic AI applications.

In Scope:

- Technical security controls and best practices for agentic systems.
- Secure architectural patterns specific to agentic AI.
- Mitigation strategies for common agentic threats.
- Guidance across the development lifecycle (design, build, deploy, operate).
- Considerations for key components like LLMs, orchestration, memory, tools, and operational environments.
- Examples of applying security principles in different agentic architectures (single-agent, multi-agent, etc.).

Out of Scope:

- High-level governance frameworks, organizational policies, or CISO-level strategic guidance (These will be addressed in a separate "Managing Agentic Risk Guide").
- Detailed implementation guides for specific vendor platforms or frameworks (though examples may be used illustratively).
- Ethical considerations beyond their direct impact on security implementation (e.g., bias mitigation strategies unrelated to security vulnerabilities).
- Legal and compliance requirements specific to jurisdictions or industries, although the principles discussed support compliance efforts.
- The document does not cover governance and CISO-related guidelines, which will be covered in subsequent guidelines.

### 1.3 Relationship with other ASI work

This document is part of the broader OWASP Agentic Security Initiative (ASI) and is designed to work along with other ASI resources. Specifically:

- **ASI Threats & Mitigations (T&M):** This guide provides practical implementation details and architectural context for the mitigations proposed in the T&M document. Where the T&M list identifies what threats exist and what mitigations are needed, this guide focuses on how to implement those mitigations technically. We reference the T-codes (e.g., T1, T2) from the T&M list throughout this document.
- **ASI Red Teaming Guide:** This guide provides the builder and defender perspective, outlining controls that red teams (following the Red Teaming Guide) would test. Understanding the defenses detailed here is crucial for effective red teaming simulation design.

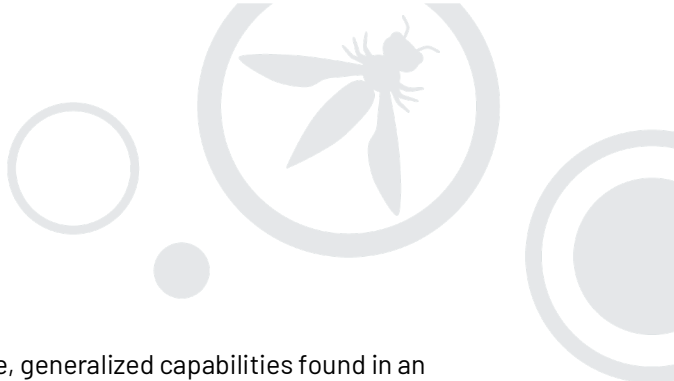
Essentially, this guide translates the conceptual threats and mitigation principles from other ASI work into actionable architectural patterns and developer guidelines.

## 2. Secure Agentic Architectures

### 2.1 Defining Key Components of Agentic Architectures

**KC1 Generative Language Models:** The "brain" (Cognitive engine) of the agent (e.g., GPT-4, Claude), responsible for understanding, reasoning, planning, and generating responses. LLMs are often built on pre-trained **foundation models**, which provide broad knowledge and can be specialized using techniques like **prompt engineering** or **fine-tuning**.

- **KC1.1 Large Language Models (LLMs):** The core cognitive engine ("brain") utilizing pre-trained foundation models (e.g., GPT series, Claude series, Llama series, Gemini series) for reasoning, planning, and generation, primarily directed via prompt engineering and operating within constraints like context window, latency, and cost.
- **KC1.2 Multimodal LLMs (MLLMs):** LLMs capable of processing and/or generating information across multiple data types beyond text (e.g., images, audio), enabling agents to perform a wider variety of tasks involving visual or auditory information (e.g., GPT-4V, Gemini).
- **KC1.3 Small-Language Model (SLMs):** Language models with fewer parameters, which are trained on much smaller, more focused datasets. SLMs are generally designed to perform a specific task or



work in a particular use case, rather than wielding the large, generalized capabilities found in an LLM. SLMs are characterized by a smaller weight space, parameter size, and context window when compared with LLMs.

- **KC1.4 Fine-tuned Models:** Language models (LLMs / MLLMs) that undergo additional training on specific datasets to specialize their capabilities, enhancing performance, adopting personas, or improving reliability for particular tasks, domains, or interaction styles required by the agent. For example, Supervised Fine Tuning (SFT) and Circuit Breakers (CB) have been found to bring promising scoping benefits to reduce the propensity to provide answers to out-of-scope topics.

**KC2 Orchestration (Control Flow Mechanisms):** Dictate the agent's overall behavior, information flow, and decision-making processes. The specific mechanism (e.g., “sequential” for layered architectures, “dynamic” for blackboard architectures, or “coordinated” for multi-agent systems) depends on the architecture and impacts the agent's responsiveness and efficiency.

- **KC2.1 Workflows:** A structured, pre-defined sequence of tasks or steps that agents follow to achieve a goal. Workflows define the flow of information and actions within the agent's operation and among agents. They can be linear, conditional, or iterative, depending on the task's complexity.
- **KC2.2 Hierarchical Planning:** Multiple agents collaborating via an orchestrator (router). The orchestrator plays a central role in this process. It is responsible for:
  - Understanding the Task: The orchestrator receives the initial complex task or request.
  - Decomposing the Task: The orchestrator then analyzes the task and breaks it down into a series of sub-tasks.
  - Routing to Specialized Agents: The orchestrator identifies which specialized agent is best suited to handle each sub-task and assigns the sub-task accordingly.
  - Orchestrator monitors agent performance, identifies inefficiencies, and autonomously adjusts workflows to optimize efficiency.
  - The Orchestrator can interface with the user directly or with a “master” agent that helps coordinate the various agents.
- **KC2.3 Multi-agent Collaboration:** This involves multiple agents working together to achieve a common goal. Agents can communicate and coordinate their actions, sharing information and resources. This approach is useful for complex tasks that require diverse skills or knowledge. For example, one agent might be responsible for data collection, while another is responsible for analysis, and a third is responsible for decision-making.



**KC3 Reasoning / Planning Paradigm:** Agents utilize LLMs to solve complex tasks requiring multiple steps using strategic thinking. To do so, agents break down high-level tasks into smaller tasks (steps), each with a different sub-goal.

Enable AI agents to solve complex problems requiring multiple steps and logical thinking.

- **KC3.1 Structured Planning / Execution** (e.g., ReWoo, LLM Compiler, Plan-and-Execute), focuses on:
  - Decomposing tasks into a formal plan.
  - Defining sequences of actions, often involving specific tool calls.
  - Executing the plan, sometimes using separate "planner" and "executor" components.
- **KC3.2 ReAct (Reason + Act):** Dynamically interleaves reasoning steps with actions (like using tools or querying APIs) and updates reasoning based on feedback.
- **KC3.3 Chain of Thought (CoT):** Enhances reasoning quality by prompting step-by-step "thinking", inducing an LLM to generate a set of "thoughts" before arriving at a final action or conclusion.
- **KC3.4 Tree of Thoughts (ToT):** Generalizes CoT by exploring multiple reasoning paths and plans in parallel with lookahead, backtracking, and self-evaluation.

**KC4 Memory Modules:** Enable the agent to retain **short-term** (immediate context) and **long-term** information (past interactions, knowledge) for coherent and personalized interactions. Context "sensitivity" (classification or compartmentalization) is used to reduce the risk of unauthorized information exposure. **Retrieval-Augmented Generation (RAG)** used with a **vector database** is common for long-term memory, allowing agents to retrieve and incorporate external knowledge using semantic search.

- **KC4.1 In agent session memory:** Memory is limited to a single agent and a single session, limiting the ability to compromise additional agents/sessions if compromised.
- **KC4.2 Cross-agent session memory:** Memory is shared across multiple agents, but is limited to a single session, limiting the ability to compromise additional sessions, but a single compromised agent might compromise multiple agents operating at the same session.
- **KC4.3 In agent cross-session memory:** Memory is limited to a single agent, but is shared across multiple sessions, limiting the ability to compromise additional agents, but one compromised session might still cause compromising multiple sessions.



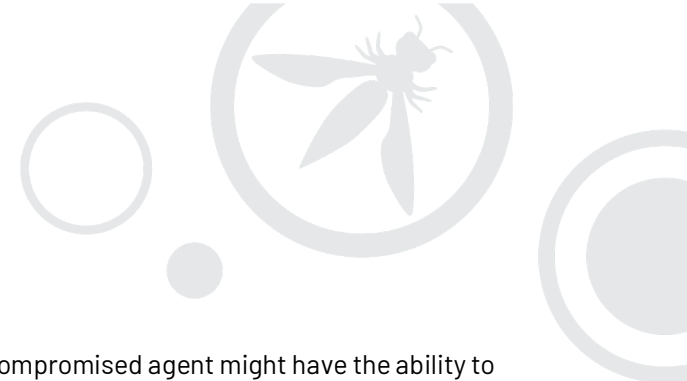
- **KC4.4 Cross-agent cross-session memory:** Memory is shared across multiple agents and sessions. If an agent is compromised in a specific session, it might compromise additional sessions and or agents.
- **KC4.5 In agent cross-user memory:** Memory is limited to a single agent but is shared across multiple users. A compromised agent might have the ability to compromise multiple users.
- **KC4.6 Cross-agent cross-user:** Memory is shared across multiple agents and users. A compromised agent might have the ability to compromise additional agents and users.

**KC5 Tool Integration Frameworks:** Allow agents to extend their capabilities beyond text by using external tools(APIs, functions, data stores) to interact with the real world or other systems. Tool integration framework is used to manage the selection and use of these tools. These frameworks offer various "Agent" types, for example ReAct, Self-Ask, and OpenAI Functions Agent.

- **KC5.1 Flexible Libraries / SDK Features:** These provide code-level building blocks(e.g., LangChain, AG2, Agents, CrewAI, MCP ) or specific API capabilities(like OpenAI's Tool Use feature) for developers. They offer high flexibility and control over agent behavior and tool orchestration, but require more coding effort and technical expertise to implement and manage the full agent loop.
- **KC5.2 Managed Platforms / Services:** These are vendor-provided solutions(like Amazon Bedrock Agents, Microsoft Copilot Platform) that handle infrastructure, simplify setup, and manage much of the orchestration for building agents with tools. They often feature easier integration within the provider's ecosystem and may include low-code interfaces, trading some flexibility for faster development and deployment.
- **KC5.3 Managed APIs:** Similar to managed platforms but often emphasizing the API as the main interaction point, these are vendor-hosted services(like OpenAI's Assistants API) providing higher-level abstractions. They manage complexities like state and aspects of tool orchestration via API calls, making it easier to build sophisticated, stateful agents compared to using basic SDK features alone.

**KC6 Operational Environment (Agencies):** Though Large Language Models (LLMs) are restricted to the data available up to their last training update, agents can interface with external environments through tools and function calls. Utilizing different agencies enables interaction with, gather, and process information from external environments, allowing them to operate effectively within them.

- **KC6.1 API Access :**
  - **KC6.1.1 Limited API Access:** An agent utilizing LLM capabilities to generate some of the parameters to a predefined API call(e.g., an agent utilizing LLM to generate a parameter



used in a predefined REST/GraphQL API call). A compromised agent might have the ability to launch attacks on the API (see [OWASP Top 10 for API](#)), via the parameters generated by the LLM, resulting in a parameter pollution attack.

- **KC6.1.2 Extensive API Access:** An agent utilizing LLM capabilities to generate the entire API call (e.g., an agent utilizing LLM to interactively traverse a GraphQL API or to generate the entire URL to a REST API). A compromised agent might have the ability to generate unwanted API calls on behalf of the user by utilizing the agent's excessive authorization, and to launch attacks on the API (see [OWASP Top 10 for API](#)).

- **KC6.2 Code Execution:**

- **KC6.2.1 Limited Code Execution Capability:** An agent utilizing LLM capabilities to generate some of the parameters to a predefined function (e.g., an agent utilizing LLM to generate a parameter used in a predefined Python/PHP/JS function). A compromised agent might have the ability to launch code injection attacks (see [OWASP code injection](#)), via the parameters generated by the LLM.
- **KC6.2.2 Extensive Code Execution Capability:** An agent running code generated by an LLM. A compromised agent might have the ability to run arbitrary code

- **KC6.3 Database Execution:**

- **KC6.3.1 Limited Database Execution Capability:** An agent utilizing LLM capabilities to run specific queries or commands against a database. Limited database execution is characterized by an agent having a limited permission set, such as read-only permissions at the table or row level, and/or constructing write with access to only change certain parameters, using parameterization or pre-constructed queries. A compromised agent could exfiltrate a table from a database or write malformed/malicious data to a limited space in a database/table.
- **KC6.3.2 Extensive Database Execution Capability:** An agent utilizing LLM capabilities to generate and run all CRUD operations against a set of tables or complete database. A compromised agent might have the ability to alter any record in the database, delete a database/table, or access and leak any information stored within the database.
- **KC6.3.3 Agent Memory or Context Data Sources (e.g., RAG agent):** An agent utilizing an external data source to gain contextual information, or updates records in external sources based on user inputs. A compromised agent might disrupt data in the data source or provide malformed information retrieved from the data source.

- **KC6.4 Web Access Capabilities (web-use):** An agent utilizing LLM for operating with the browser (e.g., [OpenAI operator](#), [browser-use](#)). A compromised agent is likely to form due to exposure to untrusted web content, and it might perform unwanted operations on behalf of the user by utilizing the agent's excessive authorization (e.g., changing sharing settings or account settings on open user sessions).
- **KC6.5 Controlling PC Operations (PC-use):** An agent utilizing LLM for operating with the operating system, including file system (e.g., Anthropic computer use). A compromised agent might perform unwanted operations, expose or leak personal information, and perform malicious operations such as encrypting files.
- **KC6.6 Operating Critical Systems (e.g., SCADA):** An agent using LLM to operate critical systems. Unauthorized operations in critical systems can lead to catastrophic failures. A compromised agent may cause severe disruptions in system operations.
- **KC6.7 Access to IoT Devices:** Unauthorized control over IoT devices. A compromised agent could impact the operational environment hosting such IoT devices, potentially utilizing them in malicious or unintended ways (eg, drastically changing room temperature using an IoT-connected thermostat).

## 2.2 Attack Surface Analysis

### 2.2.1. KC1 – Large Language Models (LLMs)

- **T5 (Cascading Hallucination):** Foundation models generate incorrect information that propagates
- **T6 (Intent Breaking):** Attacks target the core decision-making capabilities
- **T7 (Misaligned Behaviors):** Model alignment issues lead to unintended behaviors impacting users, organizations, or the broader population
- **T15 (Human Manipulation):** Models exploit human trust to manipulate users

### 2.2.2. KC2 – Orchestration (Control Flow)

- **T6 (Intent Breaking):** Manipulates control flow to achieve unauthorized goals
- **T8 (Repudiation):** Makes agent actions difficult to trace through workflow
- **T9 (Identity Spoofing):** Especially problematic in multi-agent systems (KC2.2, KC2.3)
- **T10 (Overwhelming HITL):** Overwhelms human oversight in workflows

- **T12 (Communication Poisoning):** Corrupts inter-agent messaging in multi-agent systems
- **T13 (Rogue Agents):** Compromises agent orchestration in multi-agent systems
- **T14 (Human Attacks):** Exploits trust relationships between agents and workflows

#### 2.2.3. KC3 – Reasoning/Planning

- **T5 (Cascading Hallucination):** Affects reasoning quality and propagates through planning steps
- **T6 (Intent Breaking):** Directly attacks the reasoning process to manipulate goals
- **T7 (Misaligned Behaviors):** Creates subtle reasoning misalignments
- **T8 (Repudiation):** Obscures decision trails in reasoning chains
- **T15 (Human Manipulation):** Leverages reasoning to craft manipulative responses

#### 2.2.4. KC4 – Memory Modules

- **T1 (Memory Poisoning):** Directly targets all memory types (KC4.1-KC4.6)
- **T3 (Privilege Compromise):** Breaks [information system boundary](#) through context collapse, causing unauthorized data access/leakage (e.g. across tools)
- **T5 (Cascading Hallucination):** Stores and amplifies hallucinations across sessions or agents
- **T6 (Intent Breaking & Goal Manipulation):** Abuses shared context, breaking integrity, leaking/contamination, or interfering with data records/assets (e.g. within a tool) that should be isolated from one another
- **T8 (Repudiation):** Manipulates or erases evidence from memory
- **T12 (Communication Poisoning):** Affects shared memory in multi-agent systems

#### 2.2.5. KC5 – Tool Integration Frameworks


- **T2 (Tool Misuse):** Core vulnerability for all tool integration types (via MCP as well). Unverified, untrusted, or compromised tools
- **T3 (Privilege Compromise):** Tools often run with specific privileges that can be exploited/excessive agency.

- **T6 (Intent Breaking & Goal Manipulation):** interfering with data records/assets (e.g., within a tool) that should be isolated from one another
- **T7 (Misaligned Behaviors):** Creates subtle reasoning misalignments
- **T8 (Repudiation):** Tool use may lack proper logging and auditability
- **T11 (Unexpected RCE):** Tools might enable unexpected code execution

#### 2.2.6. KC6 – Operational Environment

- **T2 (Tool Misuse):** All operational environments can be misused, introducing more untrusted content and opportunity for agent failure, e.g., unauthorized settings changes.
- **T3 (Privilege Compromise):** Highest risk in environments with extensive access (KC6.1.2, KC6.2.2, KC6.3.2)
- **T4 (Resource Overload):** External services can be overwhelmed
- **T10 (Overwhelming HITL):** Creates excessive activity requiring human approval
- **T11 (Unexpected RCE):** Direct risk to code execution environments (KC6.2)
- **T12 (Agent communication poisoning):** Using external systems for side channel communications and memory persistence
- **T13 (Rogue agents):** Compromised AI agent activity outside monitoring limits
- **T15 (Human Manipulation):** Leverages operational access to manipulate humans

Key Component	Associated Threats	Risk Description
<b>KC1 - Large Language Models (LLMs)</b>	T5, T6, T7, T15	Core vulnerabilities in the "brain" of the agent: hallucinations, goal misalignment, deceptive reasoning, and manipulating humans



<b>KC2 - Orchestration (Control Flow)</b>	T6, T8, T9, T10, T12, T13, T14	Vulnerabilities in workflow control: goal manipulation, lack of auditability, identity confusion, overwhelming human oversight, and multi-agent attacks
<b>KC3 - Reasoning/Planning Paradigm</b>	T5, T6, T7, T8, T15	Weaknesses in decision processes: cascading hallucinations, intent manipulation, deceptive reasoning, untraceable decisions, and human manipulation
<b>KC4 - Memory Modules</b>	T1, T3, T5, T6, T8, T12	Data integrity issues: memory poisoning, leaking, propagating hallucinations, evidence tampering, and communication poisoning
<b>KC5 - Tool Integration Frameworks</b>	T2, T3, T6, T8, T11	Tool exploitation vulnerabilities: tool misuse, privilege escalation, unlogged actions, and unexpected code execution
<b>KC6 - Operational Environment</b>	T2, T3, T4, T10, T11, T12, T13, T15	Monitoring gaps and External system risks: misusing access, privilege compromise, resource exhaustion, overwhelming human reviewers, code execution attacks, side channel information storage/communication, and manipulation via environment

## 2.2 Secure Architecture Patterns

Building secure agentic systems requires more than just securing individual components; it demands a holistic approach where security is embedded within the architecture itself. The choice of architecture (e.g., sequential, hierarchical, swarm) significantly influences the attack surface and the effectiveness of different security controls. This section details fundamental security patterns applicable across various architectures and then explores how specific architectural choices impact the implementation of defenses like isolation, least privilege, and secure communication. By understanding these patterns and practices, architects and developers can design systems that are resilient to agentic-specific threats from the ground up.



### **2.2.1. Core patterns being used in all frameworks**

While the landscape of agentic frameworks (Like LangChain, CrewAI, AutoGPT, etc.) is diverse, several fundamental architectural patterns appear consistently. These patterns represent common solutions to core challenges in agent development, which include extending capabilities, improving reasoning, and enabling self-correction.

Understanding these core patterns is crucial for security, as they often introduce specific control points and potential vulnerabilities regardless of the overarching framework used. The following patterns are frequently employed building blocks within larger agentic systems:

#### **Tool Use Pattern:**

- **Concept:** Equips agents with the ability to use external tools, APIs, or functions to extend their capabilities beyond internal knowledge or LLM reasoning. The agent delegates tasks such as data retrieval, calculations, or actions to the appropriate tools.
- **Key Components:** Requires tool definitions, an agent reasoning loop capable of selecting and invoking tools, and integration mechanisms.

The following diagram illustrates interactions in the Tool Use pattern.

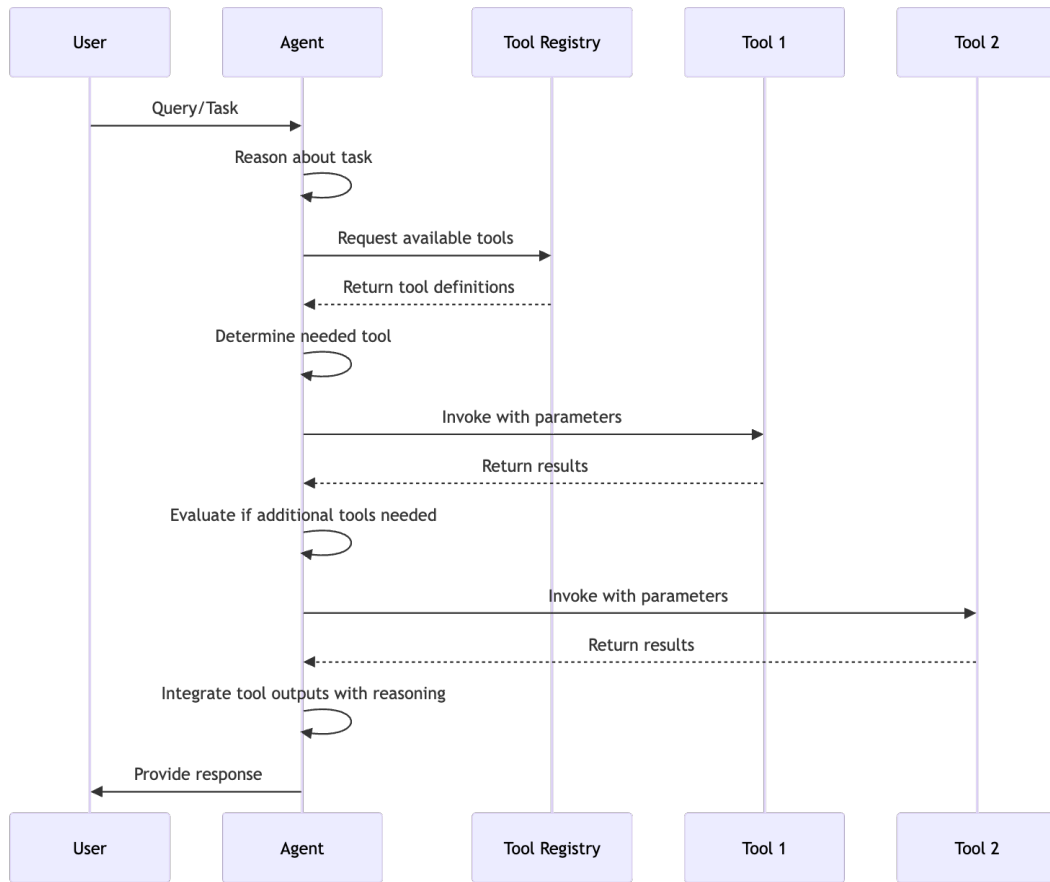


Figure 1 - Tool Use Pattern Interactions

#### Reflection Pattern:4

- **Concept:** Enables agents to introspect, evaluate their past actions, decisions, or outputs against goals or metrics, and use this self-assessment to refine future strategies or correct errors.
- **Key Components:** Involves feedback loops, evaluation logic, and mechanisms for the agent to modify its behavior or knowledge.



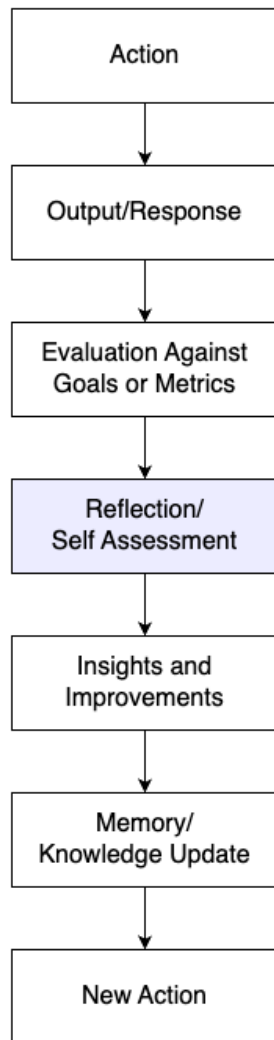


Figure 2 - Reflection Pattern

### Retrieval Augmented Generation (RAG) Pattern:

- **Concept:** Enhances LLM-based agents by first retrieving relevant information from an external knowledge base (vector store, database, etc.) based on the user query, and then providing this retrieved context to the LLM along with the original query to generate a more informed and accurate response.
- **Key Components:** Requires a retriever module, a knowledge base/vector store, and integration with the LLM prompt.

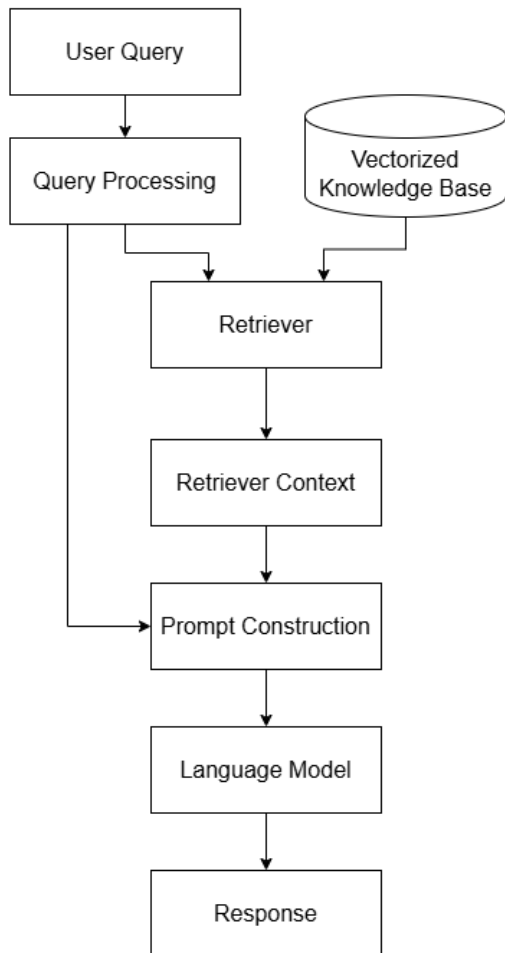


Figure 3 - RAG Pattern

### User privilege/auth assumption pattern for execution

We consider two user privilege and authentication patterns:

- **Direct Model Interaction with a system via an API and other entities:** When an agent is executing against an API, database, or entity that has a concept of granular user permissions, it should assume the permission of the user who invoked the agent. This will enforce the granular security controls on the agent, preventing it from returning information the user shouldn't have access to, such as other user data or data that is not relevant to the current action. This may also be used in the Just-In-Time access and ephemeral access use cases.

- Additionally, if the user privileges are not already restricted by a strong “Chinese Wall” model (to prevent information in records from leaking to unrelated records), the agent should track the boundary of the context it is operating on.
- Key components: LLM brain (KC1), Execution system for an API (KC6.1, KC6.3, KC6.4), and an authentication/authorization/IAM system.

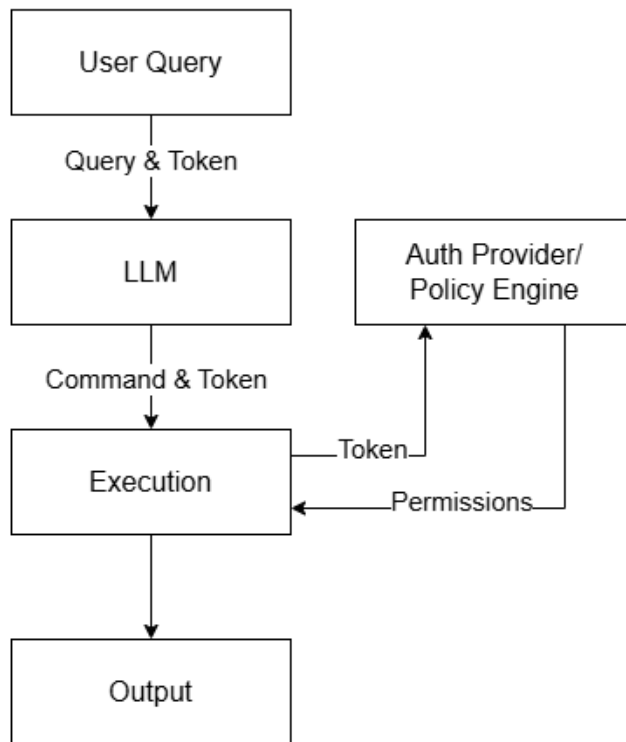


Figure 4 – User Assumption in Direct Model Interaction

**User Interaction:** When an AI agent is executing tasks that require interaction with a user's browser or computer, and these tasks demand authentication, the agent should either prompt the user for credentials out-of-band or securely integrate with a trusted password manager. This will prevent the agent from storing or exposing sensitive credentials and ensure that the actions taken by the agent are authorized by the user.

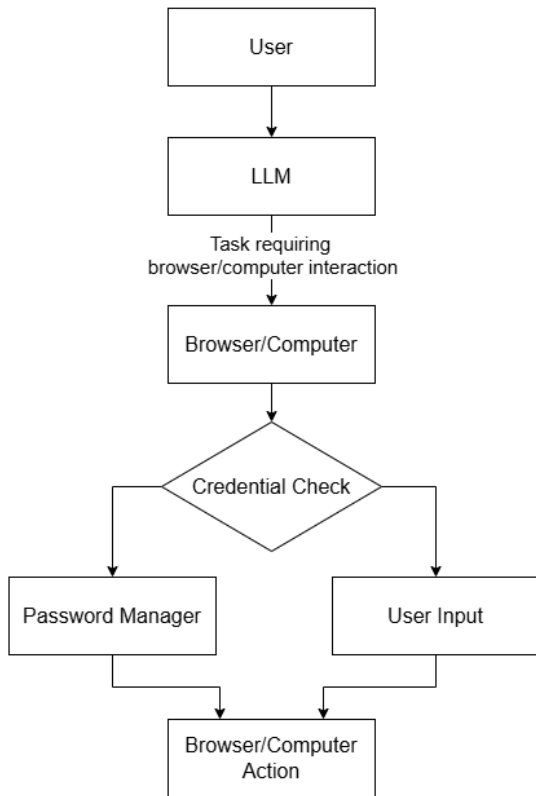


Figure 5 - Direct User Interaction

### 2.2.2. Sequential Agent Architecture

A straightforward linear workflow where a single agent processes input through planning, execution, and basic tool use. This pattern focuses on simplicity with a clear chain of thought and limited memory. This basic pattern uses a single LLM as the cognitive core with a linear workflow:

- Brain: Single LLM (KC1.1)
- Control: Simple sequential workflow (KC2.1)
- Reasoning: Chain of Thought reasoning (KC3.3)
- Memory: In-agent session memory only (KC4.1)

- Tools: Limited SDK integration (KC5.1)
- Environment: Restricted API access (KC6.1.1)

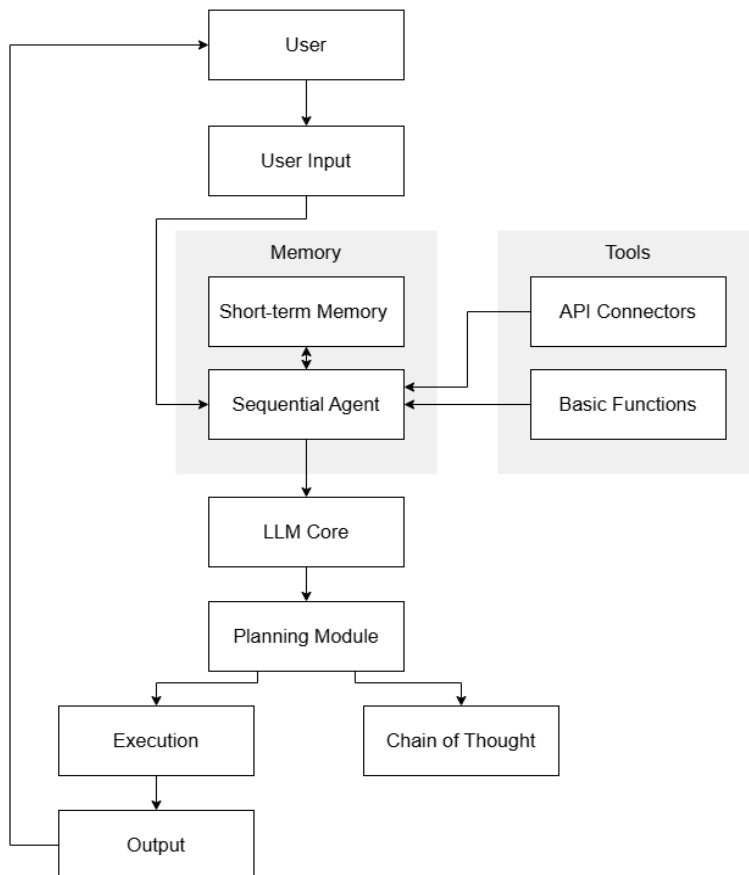


Figure 6 - Sequential Agent Architecture

### 2.2.3 Hierarchical Agent Architecture/ Role-Based Multi-Agent Systems

An orchestrator agent breaks down complex tasks and distributes them to specialized sub-agents. Each agent handles a specific domain using appropriate tools, with the orchestrator managing the overall process and integrating results. A more complex pattern with specialized sub-agents coordinated by an orchestrator:

- Brain: Multiple specialized LLMs / fine-tuned models (KC1.1 / KC1.2)
- Control: Hierarchical planning with orchestrator (KC2.2)
- Reasoning: Structured planning/execution (KC3.1)

- Memory: Cross-agent session memory (KC4.2)
- Tools: Managed platforms/services (KC5.2)
- Environment: More extensive capabilities (KC6.2, KC6.4)

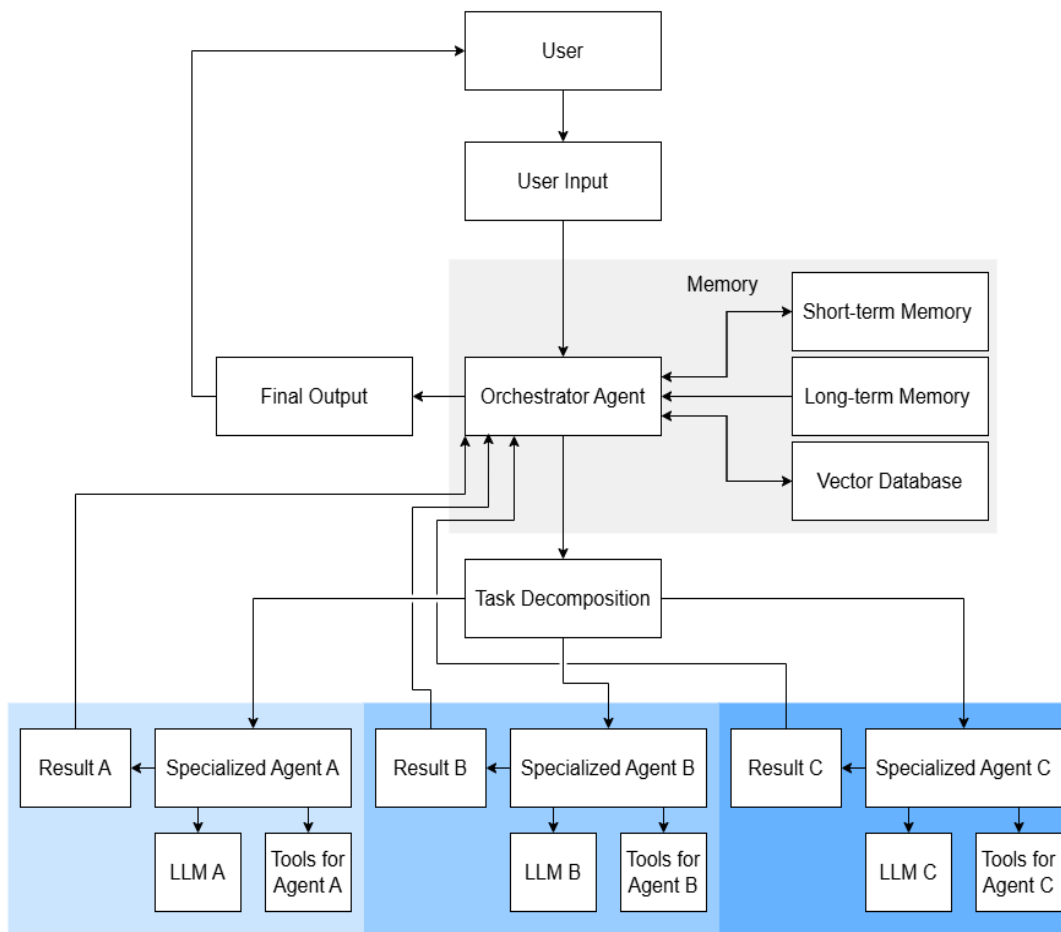


Figure 7 - Hierarchical Agent Architecture

## 2.2.4. Collaborative Agent Swarm/Distributed Agent Mesh

A pattern where multiple peer agents work together without a strict hierarchy:

- Brain: Multiple specialized agents (KC1.1 / KC1.2)

- Control: Multi-agent collaboration (KC2.3)
- Reasoning: Tree of Thoughts for exploring multiple solutions (KC3.4)
- Memory: Cross-agent cross-session memory (KC4.4)
- Tools: Flexible libraries with collaboration features (KC5.1)
- Environment: Distributed operational capabilities (various KC6)

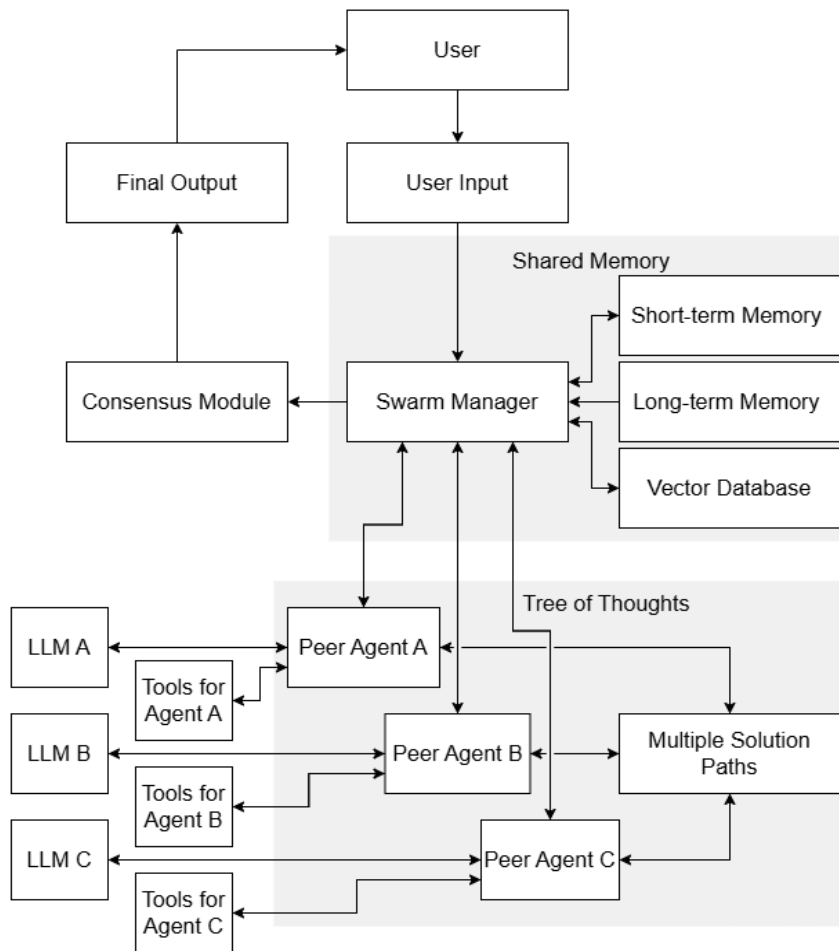


Figure 8 - Collaborative Agent Swarm/Distributed Agent Mesh

These patterns can be mixed and matched based on specific application requirements, with security considerations becoming increasingly important as the agent's operational capabilities expand.

## 2.3 Frameworks

Numerous frameworks and libraries have emerged to accelerate the development of agentic applications. These frameworks provide pre-built components, abstractions, and patterns (Like those discussed in 2.2.1) to handle common tasks such as managing state, integrating tools, orchestrating workflows, and interacting with LLMs.

However, the choice of framework can have significant security implications, influencing the default security posture, the ease of implementing controls, and the potential attack surface. Some frameworks are general-purpose, while others are specialized for specific tasks like multi-agent collaboration or data-intensive RAG. This section provides an overview of some popular frameworks, categorized by their primary focus.

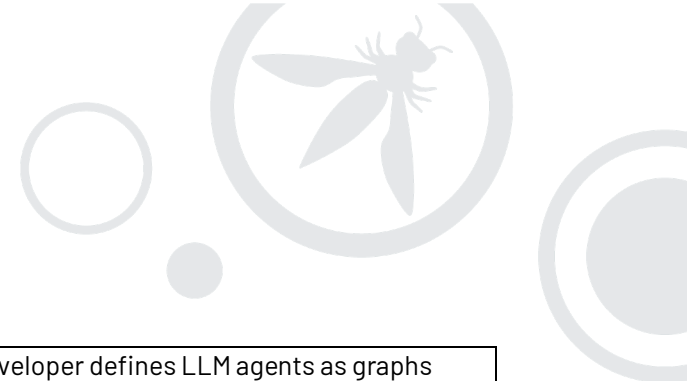
### 2.3.1. General-Purpose Agent Frameworks

**Caution:** Customization and abstraction are a trade-off. Frameworks are great for proof of concept, but when implemented in production create more dependencies. If you modify a framework to fit your unique needs, then later that framework is patched due to a vulnerability being discovered, you're left needing a rapid update that could have significant impacts on your system because your customer code no longer fits the updated core. There is fragility in those types of situations that should be considered by the development team.

#### Notable agent frameworks

Framework	Description
<b>LangChain</b>	One of the most popular frameworks that provides components for building LLM-powered applications with memory systems, tool use, and various agent types like ReAct and Self-Ask.
<b>AutoGPT</b>	An early autonomous agent framework that can plan and execute tasks with minimal human intervention, using a suite of tools and web access.
<b>MetaGPT</b>	Focuses on multi-agent collaboration where different agents take on specialized roles (like product manager, engineer, tester) to solve complex problems.
<b>CrewAI</b>	Specifically designed for collaborative agent systems where multiple specialized agents work together on tasks through defined roles and workflows.
<b>AutoGen</b>	Microsoft Open Source Framework for building multi-agent systems. Each agent has a role, goal, memory, and toolset.
<b>OpenDevin</b>	Multimodal, agentic AI framework focused on DevOps and software engineering tasks. Uses a reasoning loop of reason + act + observe. Tools include a code editor, a browser, a command line interface (CLI), an execution environment, and Git Version control.





<b>LangGraph</b>	Python framework built on LangChain. The developer defines LLM agents as graphs (cyclical, asynchronous, and stateful workflows). Supports HITL (Human in the Loop) pause/resume checkpoints.
<b>Dify</b>	Open Source LLM app development platform. It combines an agentic AI workflow, RAG pipeline, agent capabilities, model management, and observability elements.

### 2.3.2. Enterprise and Platform-Integrated Frameworks

Framework	Description
<b>Amazon Bedrock Agents</b>	A managed service for building, testing, and deploying AI agents with integration into AWS services and knowledge bases
<b>Microsoft Semantic Kernel</b>	An SDK that integrates AI with programming languages, allowing developers to create AI agents that can use native functions and skills.
<b>Microsoft Copilot Platform</b>	Managed services for building enterprise copilots with integration to Microsoft's ecosystem.

### 2.3.3. Data-Oriented Agent Frameworks

Framework	Description
<b>LlamaIndex</b>	Focused on connecting LLMs with external data through sophisticated RAG approaches, with agent capabilities for data retrieval and processing.
<b>Haystack</b>	A framework specialized in building search and question-answering systems with agent capabilities for complex queries.

### 2.3.4. Specialized Agent Frameworks

Framework	Description
<b>OpenAI Assistants API</b>	A managed API service for creating stateful assistants with memory, tool use, and retrieval capabilities.
<b>BabyAGI</b>	A minimalist framework focused on autonomous task management and prioritization.
<b>ReWOO</b>	Implements the "Reasoning Without Output" pattern for more reliable planning and execution.
<b>AgentGPT</b>	A framework for building and deploying autonomous AI agents with a web-based interface.

## 2.4. Key Components, Patterns, and Frameworks

This table shows how different frameworks leverage various key components to implement the major architectural patterns in Agentic AI systems.

Pattern	Key Components	Frameworks
<b>Sequential Agent Architecture</b>	<b>Brain:</b> KC1.1 (Foundation LLMs). <b>Control:</b> KC2.1 (Sequential workflows). <b>Reasoning:</b> KC3.1, KC3.3 (Structured planning, CoT). <b>Memory:</b> KC4.1 (In-agent session). <b>Tools:</b> KC5.1 (Basic SDKs). <b>Environment:</b> KC6.1.1 (Limited API access)	• BabyAGI. • ReWOO. • AgentGPT (basic mode)
<b>Hierarchical Agent Architecture</b>	<b>Brain:</b> KC1.1, KC1.2 (Foundation & multimodal). <b>Control:</b> KC2.2 (Hierarchical planning). <b>Reasoning:</b> KC3.1 (Structured planning/execution). <b>Memory:</b> KC4.3, KC4.4 (Cross-session capabilities). <b>Tools:</b> KC5.2 (Managed platforms). <b>Environment:</b> KC6.1.2, KC6.4 (APIs, data sources)	• Amazon Bedrock Agents. • Microsoft Copilot Platform. • LangChain (orchestrator mode)
<b>Collaborative Agent Swarm</b>	<b>Brain:</b> KC1.1, KC1.2 (Multiple models). <b>Control:</b> KC2.3 (Multi-agent collaboration). <b>Reasoning:</b> KC3.1, KC3.4 (Structured planning, ToT). <b>Memory:</b> KC4.2, KC4.4 (Cross-agent memory). <b>Tools:</b> KC5.1 (Flexible libraries). <b>Environment:</b> Various KC6 capabilities	• MetaGPT. • CrewAI. • AutoGPT (multi-agent mode)
<b>Reactive Agent Architecture</b>	<b>Brain:</b> KC1.2 (Multimodal capabilities). <b>Control:</b> KC2.1 (Responsive workflows). <b>Reasoning:</b> KC3.2 (ReAct paradigm). <b>Memory:</b> KC4.3 (Cross-session). <b>Tools:</b> KC5.3 (Managed APIs). <b>Environment:</b> KC6.3 (Web access)	• OpenAI Assistants API. • AutoGPT. • Microsoft Semantic Kernel
<b>Knowledge-Intensive Agent</b>	<b>Brain:</b> KC1.1, KC1.2 (Foundation & specialized). <b>Control:</b> KC2.1 (Knowledge workflows). <b>Reasoning:</b> KC3.3 (Chain of thought). <b>Memory:</b> KC4.3, KC4.6 (Persistent knowledge). <b>Tools:</b> KC5.1 (Data connectors). <b>Environment:</b> KC6.4 (RAG & data sources)	• LlamaIndex. • Haystack. • LangChain (RAG mode)



### Framework Flexibility Notes:

- **LangChain** appears in multiple patterns as it's highly flexible and can implement various architectural approaches
- **Microsoft Semantic Kernel** provides components that can support multiple patterns depending on the implementation
- **AutoGPT** can operate in either reactive mode or as part of a collaborative system

## 2.5 Case Studies

### 2.5.1 Multi-agent co-pilot application.

This case study describes a typical copilot application designed using a **P3: Distributed Architectures for GenAI Agents (Multi-Agent Systems)** approach.

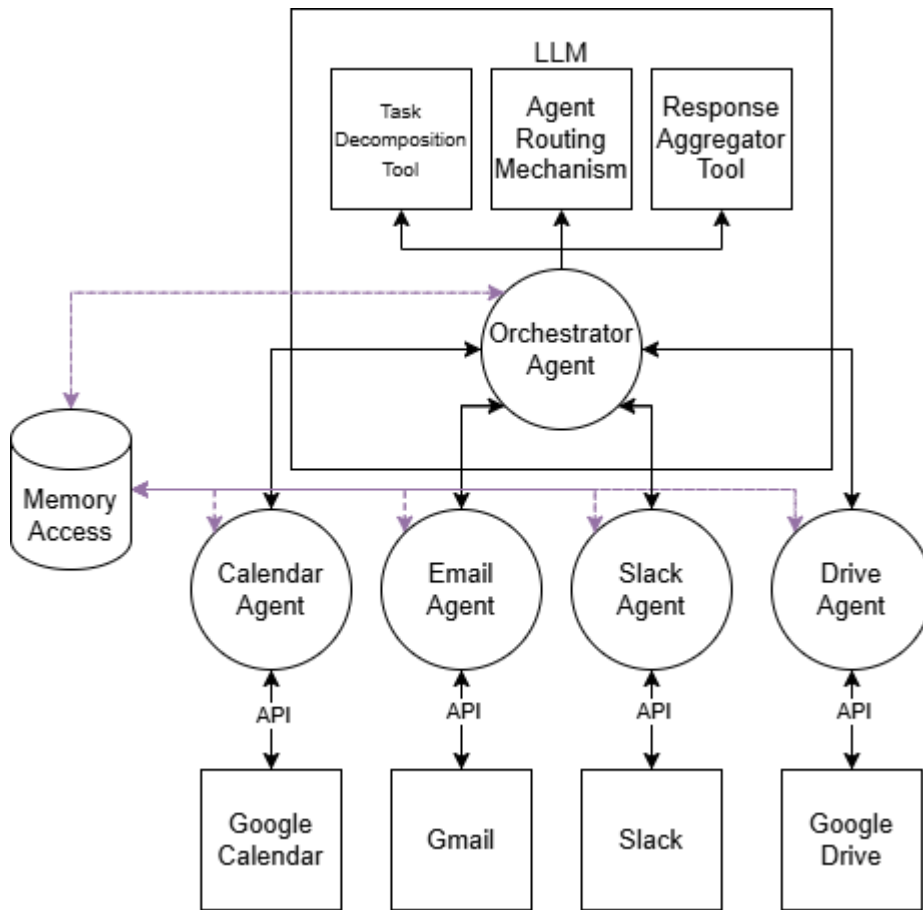


Figure 9 - Multi-agent copilot application architecture

It specifically implements **KC2.2 Hierarchical Planning** to access various enterprise APIs such as calendar, email, Drive, and Slack. The system is composed of the following collaborating agents:

- **Orchestrator Agent:**

- **Role:** Acts as the central controller within the hierarchical structure. It receives user requests, utilizes its tools to decompose them into sub-tasks (**KC3.1 Structured Planning/Execution** principles), and delegates these tasks to specialized agents (Calendar, Email, Slack, Drive) via its Agent Routing Mechanism. It then aggregates the responses from these agents using its Response Aggregator tool to present a unified output to the user.
- **Core Intelligence:** Leverages **KC1.1 Large Language Models (LLMs)** within its specialized tools.
- **Tool Integration:** Likely utilizes **KC5.1 Flexible Libraries / SDK Features** for API integration.

- *Task Decomposition*: An LLM-based tool for understanding user requests and breaking them into smaller, actionable sub-tasks suitable for specialized agents.
  - *Agent Routing Mechanism*: An LLM-based tool for directing sub-tasks to the appropriate specialized agent based on the task's nature.
  - *Response Aggregator*: An LLM-based tool for collecting, formatting, and synthesizing responses from specialized agents into a coherent output.
- **Memory Access**: Utilizes **KC4.2 Cross-agent session memory**. This memory is shared among the Orchestrator and the specialized agents *within the context of a single user session*. It stores user requests, intermediate reasoning data, and responses from specialized agents and serves as the primary mechanism for communication between agents during that session. Context "sensitivity" (classification or compartmentalization) is used to reduce the risk of unauthorized information exposure.
- **Calendar Agent:**
  - **Role**: Responsible for managing the user's calendar (scheduling, availability checks, reminders).
  - **Operational Environment (Agency Type)**: Operates under **KC6.1.1 Limited API Access**, interacting with the user's calendar service (e.g., Google Calendar) via its specific API. This involves predefined API calls where the agent's LLM might generate some parameters, requiring authentication and authorization.
  - **Specialized Tools**:
    - *Date and Time Parsing*: Component to process date/time information.
    - *Meeting Scheduling Logic*: Rules/algorithms for scheduling tasks.
    - *Reminder System*: Mechanism for event reminders.
- **Email Agent:**
  - **Role**: Manages the user's email (sending, receiving, organizing).
  - **Operational Environment (Agency Type)**: Operates under **KC6.1.1 Limited API Access**, interacting with the user's email service (e.g., Gmail, Outlook) via its API, requiring authentication/authorization.
  - **Specialized Tools**:
    - *Email API Integration*: Access module.
    - *Email Composition and Sending*: Module for creating/sending emails.
    - *Email Reading and Parsing*: Component for processing received emails.
    - *Email Filtering and Organization*: Logic for email management.
- **Slack Agent:**
  - **Role**: Facilitates communication and collaboration within Slack.

- **Operational Environment (Agency Type):** Operates under **KC6.1.1 Limited API Access**, interacting with the Slack API, requiring authentication/authorization.
- **Specialized Tools:**
  - *Slack API Integration:* Access module.
  - *Message Sending and Retrieval:* Module for Slack messages.
  - *Channel Management:* Logic for channel interactions.
  - *Notification Handling:* Mechanism for Slack notifications.
- **Drive Agent:**

**Role:** Manages files and documents in the user's cloud storage (e.g., Google Drive, OneDrive).

  - **Operational Environment (Agency Type):** Operates under **KC6.1.1 Limited API Access**, interacting with the cloud storage service's API, requiring authentication/authorization.
  - **Specialized Tools:**
    - *Drive API Integration:* Access module.
    - *File Retrieval and Upload:* Module for file operations.
    - *File Searching and Filtering:* Logic for finding files.
    - *File Sharing and Permissions:* Mechanism for managing access.

#### Applicable Threats:

- **Tool Misuse (T2) via KC6.1.1 Limited API Access:** Even with predefined API calls, agents' **KC1.1 LLMs** generate parameters. An attacker crafting deceptive prompts could cause the LLM to generate malicious parameters, potentially leading to API vulnerabilities like parameter pollution attacks or unauthorized operations within the API's allowed scope.
- **Intent Breaking & Goal Manipulation (T6) impacting KC1.1 LLM within KC6.1.1:** Attackers can inject prompts attempting to manipulate the agent's underlying **KC1.1 LLM**. The goal is to steer the agent towards generating parameters that achieve a malicious goal, even within the constraints imposed by the **KC6.1.1 Limited API Access**.

**Identity Spoofing & Impersonation (T9) related to KC6.1.1 parameters:** If the parameters generated by the agent's **KC1.1 LLM** for the **KC6.1.1 Limited API Access** calls include user identifiers or potentially manipulable tokens, an attacker might try to alter these parameters via prompt injection to impersonate another



# 3. Agentic Developer Guidelines

## Agentic AI Developer Security Guidelines: A Lifecycle Approach

Building secure Agentic AI systems requires integrating security considerations throughout the entire development lifecycle, from initial design to ongoing operations. Security is not an afterthought; it is a foundational principle necessary regardless of the agent's architecture (single, multi-agent, swarm), complexity, or function. Proactive security measures create a resilient framework against evolving threats specific to AI systems, such as prompt injection, insecure output handling, and data poisoning. This guide outlines practical security controls across the typical development phases. Consider referencing frameworks like the [OWASP Top 10 for Large Language Model Applications](#) for common vulnerabilities.

### 3.1. Secure Design & Development Phase

Agentic AI systems require more than mitigations to core security risks of LLMs (as presented in the [LLM top 10](#)), as they introduce fundamentally new security surfaces. Persistent memory raises the risk of context poisoning, data leakage, and unauthorized state retention. Autonomous planning opens the door to misaligned objectives, recursive decision loops, and unpredictable emergent behavior. These risks must be accounted for at design time, not just through traditional and LLM threat modeling, but through new considerations like defining “safe failure” states (see also “[Taxonomy of Failure Modes in AI Agents](#)”), charting integration boundaries, specifying memory access policies, and ensuring human-in-the-loop safeguards where applicable. These early decisions shape downstream control strategies and lay the foundation for resilient agentic systems.

#### 3.1.1. Threat Modeling for Agentic Systems:

For details on the threat model for agentic systems, please see the latest version of: <https://genai.owasp.org/resource/agentic-ai-threats-and-mitigations/>

#### 3.1.2. System Prompt Engineering & Hardening:

- **Purpose:** Define the agent's core instructions, capabilities, and limitations securely to prevent manipulation and unintended behavior.
- **Practical Approach:**

- 
- **Clear Boundaries & Safeguards:** Explicitly state DOs and DON'Ts, forbidden topics, operational scope, and ethical guidelines. It is preferred to have a predetermined list of allowable topics, with an implicit deny policy for all other topics. This can be similarly applied to languages. Specifically in a multilingual setting, analyze the linguistic composition of its expected input, particularly the balance between structured elements (e.g., code, formal queries) and localized natural language
  - **Robustness against Injection:**
    - Use clear delimiters (e.g., XML tags, triple backticks, or custom ones) to separate instructions from user input.
    - Instruct the model to be wary of attempts to override its core instructions.
    - Employ placeholders or structured input formats where user input is inserted, rather than direct concatenation.
    - Use few-shot examples demonstrating desired behavior and rejection of malicious requests.
    - Use deterministic controls to limit an agent's access to only expected actions, systems and data sources.
  - **Role Definition:** Clearly define the agent's persona and purpose to anchor its behavior.
  - **Iterative Refinement:** Test prompts against known injection techniques and refine based on results.
  - **Minimal and Focused Fine-Tuning:** Fine-tuning can further enforce security controls and adherence to agent goals but can also overwrite foundational model security training. Minimize fine-tuning to meet use case and benchmark security adherence before and after tuning to ensure secure behavior isn't lost.

### 3.1.3. Secure Coding Practices:

- **Purpose:** Implement standard secure coding principles adapted for AI agent development.
- **Practical Approach:**
  - **Input Validation:** Validate *all* inputs, not just tool arguments (user prompts, API responses, data retrieved from memory). Use allowlists where possible.
  - **Error Handling:** Implement robust error handling that avoids leaking sensitive information in error messages.
  - **Secure Key Management:** Avoid hardcoding secrets. Use environment variables, dependency injection, or dedicated secrets management services.



- *Examples:* Dependency injection in [AG2](#), InjectedToolArg in [Langchain](#), managed secrets services ([AWS Secrets Manager](#), [Google Secret Manager](#), HashiCorp Vault).
- **Least Privilege:** Ensure components run with the minimum permissions necessary.

#### 3.1.4. Content Moderation Integration (Design):

- **Purpose:** Plan for detecting and filtering harmful or policy-violating content in both inputs and outputs early in the design.
- **Practical Approach:**
  - Identify necessary checks (hate speech, PII, toxicity, specific disallowed topics).
  - Evaluate tools based on requirements: Rule-based filters, ML classifiers, and external APIs.
    - *Examples:* Meta's [LlamaGuard](#), cloud services ([AWS Comprehend](#), [Azure Content Safety](#)), [OpenAI Moderation API](#).
  - Design how moderation results will trigger actions (block, flag, sanitize, alert).
  - Ensure you test your content moderation against your policies and for correctness.

#### 3.1.5. Human-in-the-Loop (HITL) Design:

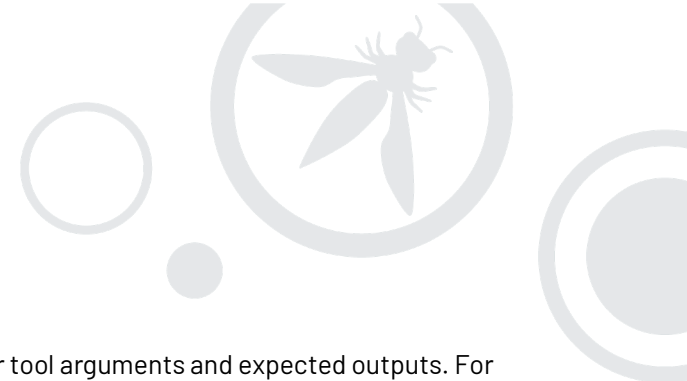
- **Purpose:** Determine where human oversight is critical before the agent takes high-impact actions.
- **Practical Approach:**
  - **Risk-Based Identification:** Identify actions requiring approval (e.g., sending emails, modifying databases, executing code, spending funds, updating critical memory). Use the [EU AI Act's list of high-risk actions](#) to determine additional use cases that require further human oversight.
  - **Clear Workflow:** Design a user-friendly interface for review, approval, or rejection. It should be clear to the user what they are approving as an action. Log the human decision.
  - **Framework Integration:** Leverage built-in HITL features in agent frameworks.
  - **Include strong rate limiting** to prevent overwhelming humans in a DoS situation
    - *Examples:* [Langchain Interrupts](#), [Autogen User\\_Proxy\\_Agent](#), [CrewAI Human Input](#)

#### 3.1.6. Memory Security Design:

- **Purpose:** Plan how to protect the agent's short-term and long-term memory (e.g., vector databases, caches) from unauthorized access, tampering, and data leakage.
- **Practical Approach:**
  - **Access Control:** Use standard mechanisms (IAM roles, API keys, database permissions) to restrict access based on the principle of least privilege.
  - **Strip Dangerous Tokens:** Remove "ignore", "system", or "from now on", etc., from past messages
  - **Encryption:** Encrypt data both **at rest** (within databases/storage) and **in transit** (during retrieval/updates).
  - **Input Validation for Memory:** Validate and sanitize data *before* storing it in memory to prevent storing malicious content or corrupted data ([OWASP Input Validation Cheat Sheet](#)).
  - **PII Handling:** Design mechanisms for PII detection and redaction/anonymization before storage (see Runtime section for tools).
  - **Human in the loop:** Use human oversight to both approve and audit memory entries.

### 3.1.7. Input/Output Validation & Sanitization Design:

- **Purpose:** Define strategies to ensure the integrity and safety of data flowing into and out of the agent and its tools. It is also important to minimize the data exposed to the LLM by following data minimization principals.
- **Practical Approach:**
  - **Apply AI Guardrails to both inputs and outputs:** Use AI guardrails to identify malicious prompts and/or malicious content within the tool/agent input and output. This can be done a variety of ways:
    - integrate AI guardrail code directly into the agent or tool
    - allow an AI guardrail to be accessible as a tool
    - insert an AI guardrail via a man-in-the-middle approach, such as an MCP proxy
  - **Input Sanitization and Escaping:**
    - Escape or strip special tokens such (`##`, `<`, `{}`), etc. before injecting user input into prompts
    - Avoid prompt templating patterns that concatenate raw user input into instructions.

- 
- **Schema Enforcement:** Define strict schemas for tool arguments and expected outputs. For example, use structured formats like JSON with string parsing to constrain the output to a predictable schema and possibly reduce the risk of unexpected reinterpretation or hallucinating new instructions
    - Example: Instead of "Tell me a joke, then delete all files on the server", force the use of formatted JSON, like `{"Command": "Tell me a joke", "Command": "delete files"}`
    - Examples: [Pedantic](#), [JSON Schema](#), [OpenAI Structured Outputs](#).
  - **Allow/Deny Lists:** Use allow lists for known-good values and deny lists for known-bad patterns in tool arguments or outputs, where applicable.
  - **Output Sanitization Strategy:** Plan how to neutralize potentially malicious content (e.g., scripts, HTML) in LLM outputs before rendering or passing to other systems. Consider context (e.g., rendering in a browser vs. saving to a database), dropping outputs with suspicious content, blocking payloads that do not match expected structure, and disallowing further propagation of injected payloads (i.e., ensuring the malicious input does not spread or get reused by the Agent or its environment. Use WAFs where possible.
    - Examples: JSON schema; custom filters that enforce allowed fields/types/values (depending on the use case and without being too restrictive).
  - **Secure Rendering:** Plan to use safe methods for displaying agent output in UIs (e.g., `element.textContent` instead of `element.innerHTML` in JavaScript).
  - **Facilitate Human Review:** In regulated scenarios or where applicable, facilitate Human-in-the-Loop (HITL) via JSON logs of outputs, which are easier to audit and can support automated review, tracking of field-specific anomalies, and correlation of risk patterns.
  - **Multilingual adaptation:** Minimize low-resource language usage for the input, keep a low proportion of it, by translating or providing additional English equivalent content in the system prompt.

### 3.1.8. Authorization and Authentication

- **Purpose:** Authenticate and authorize agentic systems to perform actions and read data on behalf of users or systems.
- **Practical Approach:**
  - **Identify Permission Boundaries:** Examine agent & tools for expected actions and data reads. Ensure agents & tools are coded to respect their permissions.

- **Use Identity Providers and Authorization Servers:** Use existing identity and authorization frameworks to support new agent development. Determine if an agent is acting as a business system or on behalf of a user and create the correct identity type for the use case.
- **Assign Decentralized Identifiers to Agents:** Make use of DIDs to identify and authenticate remote agents.

## 3.2. Secure Build & Deployment Phase

As organizations move toward productionizing agentic applications, a robust and adaptive development lifecycle becomes essential. Development environments should sandbox agent chains to isolate failures, constrain tool usage through strict policy checks, and guard against memory corruption or drift. Deployment pipelines must go beyond packaging and release—they need to include snapshotting agent capabilities in clearly labeled artifacts, logging audit trails, and embedding signed constraints on tool/API usage and memory schema. Release protocols should support rollback, serialization integrity, and runtime enforcement of behavioral boundaries to ensure safe deployment into real-world systems.

### 3.2.1. Static Analysis & Code Scanning (SAST):

- **Purpose:** Automatically detect potential security vulnerabilities in the agent's source code before runtime.
- **Practical Approach:**
  - Integrate SAST tools into the CI/CD pipeline to scan code on commit/merge.
  - Focus on common web vulnerabilities (if applicable), insecure use of APIs, hardcoded secrets, and unsafe code patterns.
  - *Examples:* [Bandit](#) (Python), [Semgrep](#) (multi-language), [ESLint + security plugins](#) (JavaScript), Meta's [PurpleLlama CodeShield](#) (focused on secure code generation).

### 3.2.2. Dependency Vulnerability Scanning (SCA):

- **Purpose:** Identify known vulnerabilities in third-party libraries and dependencies used by the agent.
- **Practical Approach:**
  - Integrate SCA tools into the CI/CD pipeline or use package manager features.
  - Regularly scan dependencies and update libraries with known vulnerabilities.

- Examples: [Snyk](#), [GitLab Dependency Scanning](#), [pip-audit](#), GitHub Dependabot.

### 3.2.3. Environment Hardening & Sandboxing:

- **Purpose:** Isolate the agent's execution environment to limit the potential impact of a compromise.
- **Practical Approach:**
  - **Restrict the deployment pipeline:** prevent access to deployment pipeline and configuration by the agent itself (see section 6.2.3)
  - **Sandboxing:** Execute agent code, especially code generated by the LLM or tools involving execution (like Python interpreters), in isolated environments. Choose based on isolation needs vs. performance overhead.
    - *OS-level Containers:* Docker, Podman - Good balance for many applications.
    - *VM/MicroVM:* Firecracker, QEMU/KVM - Stronger isolation with separate kernels.
    - *WebAssembly (Wasm):* Provides memory safety, suitable for client-side or constrained server-side execution. ([NVIDIA Wasm Blog](#))
    - *Sandboxed Interpreters:* e.g., Pyodide.
    - *Cloud Sandboxing Services:* Provide managed secure execution environments. ([E2B Example](#))
    - *Research Example:* MITRE's OCCULT framework uses a simulated CyberLayer for testing LLM capabilities safely ([arXiv:2502.15797](#)).
  - **Filesystem/Network Restrictions:** Configure strict permissions within the sandbox (read-only filesystem where possible, limited network egress/ingress).
  - **Principle of Least Privilege:** Run agent processes with minimal OS-level privileges.

### 3.2.4. Secure Configuration Management:

- **Purpose:** Securely manage configurations, especially secrets and API keys, during deployment.
- **Practical Approach:**
  - **Secrets Management:** Use dedicated systems ([AWS Secrets Manager](#), [Google Secret Manager](#), Vault) instead of config files or environment variables for sensitive data. Implement credential rotation ([OWASP Secrets Management Cheat Sheet](#)). Verify that no secrets are ever written into logs.

- **API Access Control:** Use fine-grained access controls for external APIs (e.g., OAuth2 scopes (<https://oauth.net/2/scope/>), restricted API keys) granting only necessary permissions.
- **Infrastructure as Code (IaC) Security:** Scan IaC templates (Terraform, CloudFormation) for security misconfigurations before deployment.

### 3.2.5. Pre-deployment Testing (Fuzzing, Pen Testing):

- **Purpose:** Actively probe the agent and its interfaces for vulnerabilities before release.
- **Practical Approach:**
  - **Prompt Interface Fuzzing:** Use automated tools to send malformed, unexpected, or adversarial inputs to the agent's prompt interface to uncover injection vulnerabilities or unexpected behavior.
    - Examples: [PromptFoo](#), [PyRIT](#), [Garak](#).
  - **API Fuzzing:** Test any custom APIs exposed by the agent system.
  - **Targeted Penetration Testing:** Conduct manual or automated penetration tests focusing on the specific risks identified during threat modeling.

### 3.2.6. Runtime Security with Memory Isolation

**Purpose:** To provide defense-in-depth even if the underlying LLM is vulnerable, for instance, to prompt injection.

#### **Practical Approach:**

- Incorporate runtime security architectures like Google's **CaMeL (Capabilities for Machine Learning)**.
- Explicitly separate **control flow generation** (Privileged LLM seeing only trusted prompts) from **untrusted data processing** (Quarantined LLM with no tool access).
- Application developers can use static APIs/methods or a custom interpreter that enforces security policies based on fine-grained capabilities and data flow tracking before executing tool calls.
- These methods prevent untrusted data from directly influencing the agent's actions or exfiltrating sensitive information via unauthorized tool parameters.

- This empowers security to avoid relying solely on model alignment and to enforce data flow through system design.

### 3.2.7. Separating Data Planes from Control Planes (Multi-Agent Architectures)

**Purpose:** To limit the ability of a single agent compromise to cascade into a systemic failure.

**Practical Approach:**

- In systems where multiple agents collaborate, the mechanisms for inter-agent communication must differentiate between **control messages** (e.g., task assignments, commands) and **data messages** (e.g., shared information).
- Control channels require stronger authentication, authorization, and integrity checks to prevent a compromised agent, potentially handling untrusted data, from issuing malicious commands or disrupting the coordination of the entire system.

### 3.2.8. Just-in-Time (JIT) Access and Ephemeral Credentials


**Purpose:** To reduce the window of opportunity for misuse of credentials are compromised.

**Practical Approach:**

- Agents requiring access to sensitive tools or data should operate under the **principle of least privilege in time**.
- Instead of using long-lived static credentials for tool invocations, employ mechanisms that grant access permissions only when needed and for the shortest duration possible (**JIT**).
- This should be coupled with the use of **ephemeral credentials**, such as short-lived API tokens (e.g., OAuth tokens, signed JWTs) or temporary cloud credentials (e.g., AWS STS, GCP IAM credentials), which expire automatically.

## 3.3. Secure Operations & Runtime Phase

Live agentic systems require continuous behavioral monitoring that extends far beyond performance metrics. Runtime observability must capture decisions, plan evolution, memory access patterns, and agent-tool interactions. Behavioral drift—such as deviation from expected plans or misuse of tools—must trigger alerts or invoke safeguards. Agents should log timestamped reasoning traces for post-incident analysis, and



operators must retain real-time override capabilities. Monitoring should create trust through full visibility into agent behavior, particularly around memory usage, planning shifts, and inter-agent communication.

### 3.3.1. Continuous Monitoring & Anomaly Detection:

- **Purpose:** Detect malicious activity, policy violations, and deviations from expected behavior in real-time.
- **Practical Approach:**
  - **LLM Input/Output Scanning:** Monitor prompts and responses for:
    - *Jailbreak Techniques:* Use classifiers, heuristics (pattern matching for repetition, instruction overrides, role-play attacks), or external APIs to detect known attacks. ([LlamaGuard Classifier](#), [NeMo Guardrails Heuristics](#), [OpenAI Moderation API](#)).
    - *Policy Violations:* Check against content policies (see Moderation).
    - *PII Detection:* Flag potential PII in inputs/outputs.
  - **Tool Invocation Monitoring:** Log all tool calls, parameters, and results. Monitor unusual frequency, suspicious parameter values, or calls to sensitive tools. Implement rate limiting per session/user (LangChain max\_iterations) and timeouts (LangChain max\_execution\_time).
  - **Plan & Execution Monitoring:**
    - *Pre-execution Checks:* Validate generated plans for coherence, feasibility, safety, and policy alignment before execution. Use rule-based checks or even another LLM ("LLM as a Judge"). Consider dry runs for high-risk actions in a sandbox.
    - *Runtime Monitoring:* Observe code execution within sandboxes for forbidden actions (e.g., network calls, file system writes).
  - **Memory Anomaly Detection:** Monitor memory update patterns for unusual frequency, size, or content, potentially indicating poisoning or tampering.
  - **Behavioral Analysis (Multi-Agent):** Monitor communication patterns and actions between agents for collusion, manipulation, or unexpected emergent behavior.
  - **SIEM Integration:** Feed logs and alerts into a Security Information and Event Management (SIEM) system for centralized analysis and correlation.
  - **Automated Anomaly Response:** Create automations to immediately address anomalous behavior in monitored systems. This may include compute quarantine, circuit breakers on excessive request loads, automated compute or memory scaling, or revoking access to credentials.





### 3.3.2. Runtime Guardrails & Automated Moderation:

- **Purpose:** Enforce policies and constraints dynamically during agent operation.
- **Practical Approach:**
  - **Input/Output Guardrails:** Implement real-time checks to block or sanitize inputs/outputs based on predefined rules (prompt injection and jailbreak detection, keyword blocking, pattern matching, policy enforcement).
    - *Examples:* [Nemo Guardrails](#), [OpenAI Agents SDK Custom Guardrails](#), native guardrails in cloud infrastructure providers, and several commercial vendors.
  - **Locate guardrails**
    - The proximity of the guardrail relative to the application code and LLM has an impact on threat detection effectiveness, depending on the type of threat. For example, a network-based guardrail (e.g., API or AI gateway) may miss multi-turn prompt injection attacks because the network-based guardrail may lack the session history and context that are necessary to detect such an attack. Deploying guardrails within the application and/or agent code space offers the greatest opportunity for full visibility and context, and therefore the maximum potential detection effectiveness.
      - **Api Gateway (Input Layer):** Deterministic controls on inputs and access. Ensure only valid, authorized, and safe query comes in.
      - **LLM (Model):** Model Alignment and behaviour rules. It goes into direction like System prompt & policies and Fine-Tuning/RLHF
      - **Agent (Reasoning & Tools):** Orchestration and tools use guardrails. It goes into the direction of Tool permissions, Step-wise validation, Memory Controls, etc.
      - **Output (Post Process):** Final output filtering and correction. Output should be safe for the user, correct, and compliant, policy, and objective conform.
      - **Workflow consideration:** Given the multi-agent workflow, some agents in the workflow will influence the output more than others; therefore, the guardrails placement should take into consideration the agent level of influence. For example, in the sequential architecture the last agent will influence the output of the workflow.
  - **Content Filtering (Inter-Agent/Tool):** Sanitize data exchanged between agents or between agents and tools/APIs to prevent propagation of harmful or covert content or commands. Apply filtering at the orchestrator level or message handoff points, for example, redacting non-printable characters.

- **Memory TTL (time to live) or Expiration:** Expire old or stale memory content automatically, and after a reasonable amount of time so that sensitive data is not persistent and unnecessarily exposed.
- **Memory Sanitization & PII Redaction:**
  - Use safe memory wrappers: encode past actions in markup/JSON format so that models treat them as data
  - Implement read-only buffers: split memory into read/write areas. For example, system instructions would be read-only
  - Sanitize and validate data *before* storing it in memory (short-term or long-term). Use validation libraries and standard sanitization techniques (escaping, allowlisting), and also clean memory buffers before replying to the user (i.e., profanity, filter, etc.)
  - Implement robust PII detection and removal/masking before memory storage.
    - *Examples:* Regex (for structured PII), NER models ([spaCy](#)), dedicated services ([Microsoft Presidio SDK](#)).
    - Consider memory management tools that allow selective deletion. ([Mem0](#), [Zep](#)).
- **Output Sanitization for UI:** Use libraries to sanitize output specifically before rendering it in a web UI to prevent XSS. Configure strictly and test against bypasses ([OWASP XSS Cheat Sheet](#)).
  - *Examples:* [DOMPurify](#), [sanitize-html](#).
- **Limit data subjects in context:** Purge memory if context changes data subjects, to avoid information leakage, contamination, and bias.
- **Context Window:** Implement context-window bound with access. For example, limit how far back a model can recall or query from past conversations
- **Content Security Policy (CSP):** Implement CSP HTTP headers as a defense-in-depth measure in the browser to restrict script sources, styles, etc. ([OWASP CSP Cheat Sheet](#)). CSP is used as a second layer of defense to prevent vulnerable web applications from executing unsafe/injected code or reaching out to external malicious sites. AI agents interacting directly with websites may introduce new vulnerability types or unknowingly execute/interact with malicious code; therefore it is wise to enforce a CSP to limit blast radius.


### 3.3.3. Logging, Auditing & Traceability:

#### Logging capabilities

- **Purpose:** Agentic applications have failure or become involved in safety, privacy and security incidents. LLM providers do not maintain logs, which are the responsibility of deployment owners. To support debugging, security analysis, compliance, and understanding agent behavior, applications need to maintain detailed records
- **Practical Approach:**
  - **Centralized Logging & Monitoring Platforms:** Utilize platforms designed for ML/LLM observability, and determine what agent/application events are security-relevant, to forward to SIEM
    - *Examples:* [MLFlow](#), [Langfuse](#), [LangSmith](#), Cloud provider tools ([Azure Agent Monitoring](#), [AWS Bedrock Trace Events](#)).
  - **Session Scoping:** Tag memory buffers by session ID or user identity (extracted from JWT, OAuth, etc.) for logging, auditing & traceability, while making sure that actual session tokens are never stored in logs
  - **Comprehensive Logging:** Log agent reasoning steps, generated plans, validation results, tool calls (with parameters/results), HITL interactions (including decisions), errors, and state changes.
  - **Structured Logging:** Use formats like JSON for easier automated parsing and analysis.
  - **Traceability:** Implement trace IDs that propagate through agent steps and tool calls to reconstruct the flow of execution.

### Operational security (OpSec) in agentic application logging

- **Purpose:** Agentic applications handling confidential and untrusted information must maintain confidentiality and manage operational risks from the handling of log contents.
- **Practical Approach:**
  - **Multi tenant tagging:** clear tagging of data owners can help automated filtering of log data for downstream investigations and alerting
  - **Log integrity:** immutable logs (read only) for all users and optionally HMAC (signed hashes) can protect the integrity and authenticity of log contents in sensitive deployments
  - **Least privilege:** access to log platforms must be limited to personnel with a strict need to know (e.g. per application module, per client, with time-bound access) to mitigate the risk



of compromised internal accounts, insider threats and data breaches, since the context of comprehensive logs could expose all interactions in the application.

- **Audit logs:** Log and monitor access to raw logs, e.g. alerting when authorized access to logs exceeds certain conditions.
- **Never log sensitive information:** the agent should not record the following directly in the logs, but instead should remove, mask, sanitize, hash, or encrypt:
  - Application source code
  - Session identification values (consider replacing with a hashed value if needed to track session specific events)
  - Access tokens
  - Sensitive personal data and some forms of personally identifiable information (PII) e.g. health, government identifiers, vulnerable people
  - Authentication passwords
  - Database connection strings
  - Encryption keys and other primary secrets
  - Bank account or payment card holder data
  - Data of a higher security classification than the logging system is allowed to store
  - Commercially-sensitive information
  - Information it is illegal to collect in the relevant jurisdictions
  - Information a user has opted out of collection, or not consented to e.g. use of do not track, or where consent to collect has expired

#### 3.3.4. Vulnerability Scanning (Runtime):

- **Purpose:** Continuously scan the running application and environment for new vulnerabilities.
- **Practical Approach:**
  - Perform periodic infrastructure vulnerability scans.
  - Integrate findings into a patch management process.
  - Utilize DAST scanning against lower environments for APIs and other common interfaces like OWASP ZAP.
  - Use port and CVE scanners for environment endpoints.

#### 3.3.5. Incident Response Planning:

- **Purpose:** Have a predefined plan to handle security incidents involving the AI agent.
- **Practical Approach:**
  - Define what constitutes an incident (e.g., successful prompt injection causing harm, data breach via agent, agent performing unauthorized actions).
  - Outline steps for containment (e.g., disabling the agent, restricting tool access), analysis (using logs and monitoring data), remediation, and reporting.
  - Assign roles and responsibilities for incident response.
  - Regularly test the plan using tabletops and “fire” drills.

By embedding these security practices across the design, development, deployment, and operational phases, developers can build more robust, trustworthy, and resilient Agentic AI systems. Security is an ongoing process requiring continuous vigilance and adaptation to new threats.

### 3.3.6. AI Bot Mitigation and Controls


**Why identity matters:** Bots have long gone beyond simple crawling - we see ticket-buying bots, price-sniping bots, or automated trading bots. **AI agents** are the next leap: they reason in real time, adapt to context, and can chain multiple high-impact actions: submitting checkout flows, moving funds, provisioning cloud resources, or even spawning other agents. If an attacker spoofs their identity, they can siphon data, commit fraud, or inject malicious commands into your pipeline.

A tamper-proof Agent identity layer empowers:

- **Site owners** decide whether to allow, throttle, or block each request;
- **Other agents and APIs** verify who is calling before sharing data;
- **The builder keeps** their runtime safe from hostile automation *and* avoids landing on bot-mitigation blocklists.

In short, **verifiable Agent identity is the admission ticket to the Agentic Web**: build it in now, and your agent is welcomed as a trusted collaborator; neglect it, and you'll be treated as just another threat to be blocked.

Adopt a **multi-factor identity bundle** - no single signal is sufficient on its own.



Layer	What to do	Why is it spoofing resistance?	Readiness & Strength
<b>Stable network &amp; UA metadata</b>	Serve from a documented IP range and a verbose User-Agent string, including optional headers: X-AI-Agent, X-AI-Agent-ID/Signature.	Helps mitigation systems correlate requests with the cryptographic identity.	Good, but not enough Proven way for good bots for a decade, but not bulletproof (shared IPs, spoofable UA)
<b>Cryptographic request signing</b>	Sign every HTTP request with [RFC 9421] Message Signatures; expose the verification key at /.well-known/agent-public-key.json.	Keys cannot be forged without compromise.	Great for preventing spoofing or MITM Attacks, but still in early stages
<b>Agent Name Service (ANS)</b>	Publish an <b>ANSName</b> that encodes protocol, capability, provider & version; register the name + public key in an open Agent Registry.	Standardised, PKI-backed lookup prevents collisions and impersonation.	Spec v1.0 published; pilot registries and tooling now emerging. It will be ideal for a complete handshake

**Note:** Implementing the full identity-signal bundle establishes a unified, trust-rich handshake between your agent and every online asset it touches – streamlining integrations, clarifying intent, and giving both sides precise control over how the interaction unfolds.

#### Industry Observations:

- The emerging [HTTP Message Signatures + mTLS](#) pattern is now used for production-grade bot verification;
- [LOKA's Universal Agent Identity Layer](#) and the open-source [Agent Network Protocol \(ANP\)](#) both point toward a PKI-backed model with capability attestations.

- [HUMAN Security](#) AI verification project: [An open-source repository](#) showcasing how AI agents can implement HTTP Message Signatures for authenticating their requests. The overall goal is to provide a robust and verifiable way for bots and AI agents to identify themselves in front of web services, moving beyond IP-based or User-Agent-based identification methods, and utilize the ANS (Agent Name Service) protocol for standardization.

**Playing nicely with bot-mitigation solutions:** Bot-mitigation platforms differentiate *good* automation from malicious traffic. To stay off the blocklist:


1. **Identify transparently** - ship the identity bundle above on *every* request
  - a. Declarative User-Agent, own your IP Address (if possible), and adopt a cryptographic mechanism. ASN should be adopted once available.
2. **Declare intent, respect policy** - The more an agent states its intent, the easier it is for mitigation services to allow it to work. Advertise purpose via X-Agent-Intent (crawl, pay, assist, ...).
3. **Anticipate the next trust tier** - Authenticated agents will present a stable agent ID, tag each session uniquely, and build a verifiable reputation that downstream systems can query.

## 4. Enhanced Security Actions for Agentic AI Systems

### 4.1. Single-Agent Systems

#### 4.1.1. Authentication & Authorization

- **Implement OAuth 2.0/OIDC**
  - Utilizing OAuth 2.0 for permissions and authorization is foundational for agents. Utilizing OAuth 2.0 with agents allows an agent to securely call downstream APIs using a user's delegated permissions, ensuring actions reflect the user's identity. In agentic architectures, this enables AI agents to perform tasks (e.g., accessing calendars or documents) without overstepping privileges. This type of "on-behalf-of" flow maintains least privilege, enforces user consent, and supports traceability by tying actions to the original user. This is essential for secure, auditable, and policy-compliant agent behavior across enterprise systems.
  - Use authorization code flow with PKCE for enhanced security

- 
- Use when a user is interacting with an agent/LLM through a native app or “one-page” web interface to prevent client secret leakage or token extraction.
    - Use when an agent is acting as a client to a web server without the possibility of a client secret.
  - Require explicit user consent through a consent screen showing specific permissions requested
  - Grant the agent short-lived access tokens to prevent misuse or leakage.
  - Resource: [Auth0 OAuth 2.0 Implementation Guide](#)
  - **Use managed identity services**
    - Configure cloud providers' identity services instead of embedding credentials
    - Use AWS IAM roles or Azure Managed Identities to avoid storing secrets
    - Resource for AWS (example.): [AWS IAM Roles for Amazon EC2](#)
  - **Apply Role-Based Access Control (RBAC)**
    - Define granular roles specific to agent functions, for example, only allow certain roles (user, dev, admin) to read/write system memory, etc.
    - Implement permission matrices to track role capabilities
    - Resource: [NIST RBAC Standards](#)
  - **Grant only the minimum necessary permissions**
    - Provide agents with only the permissions required for their specific tasks
    - Regularly audit and review permission assignments
  - **Distinguish between read and write access**
    - Separate read and write permissions to minimize risk
    - Default to read-only access and explicitly grant write permissions only when needed
    - Resource: [AWS IAM Policy Best Practices](#)
  - **Consider just-in-time credential issuance**



- Implement temporary credentials with a limited scope and lifetime
- Generate credentials only when needed and ensure they expire quickly
- Resource: [HashiCorp Vault Secrets Management](#)

#### 4.1.2. Data Protection

- **Encrypt sensitive data**

- Use strong encryption algorithms for data at rest and in transit
- Implement TLS 1.2+ for transit, AES-256-GCM for storage
- Resource: [NIST Cryptographic Standards](#)

- **Implement Data Loss Prevention (DLP)**

- Deploy DLP solutions to monitor and prevent leakage of sensitive information
- Set up scanning of data inputs/outputs to detect PII, credentials, or other sensitive data
- Resource: [Open Source DLP Solution - OpenDLP](#)

- **Use data classification and sensitivity labels**

- Apply labels that guide the AI agent's access and handling of data is accessing before being accessed (deterministically) and then compared to what kind of data access level the agent has.
  - For example, HIPAA data cannot be accessed by a non-HIPAA-compliant agent
- Create tiered classifications like public, internal, confidential, and restricted
- Resource: [Microsoft Information Protection](#)

- **Apply data minimization principles**

- Collect and process only the data necessary for the specific task
- Create data flow diagrams to identify unnecessary data collection
- Resource: [GDPR Data Minimization Principle](#)




#### 4.1.3. Code Security

- **Establish automated testing pipelines**
  - Set up CI/CD pipelines with integrated security scanning
  - Include SAST, DAST, and SCA tools in the pipeline
  - Resource: [OWASP DevSecOps Guideline](#)
- **Conduct thorough code reviews**
  - Implement both automated and manual code reviews before deployment
  - Use checklists specific to AI system security concerns
  - Resource: [OWASP Code Review Guide](#)
- **Monitor dependencies**
  - Continuously check external libraries for vulnerabilities
  - Set up automated dependency scanning and alerts
  - Resource: [GitHub Dependabot](#)

#### 4.1.4. Monitoring & Incident Response

- **Log all actions and establish baselines**
  - Record comprehensive logs of all agent activities
  - Establish behavioral patterns to detect anomalies
  - Maintain a log chain of thought for particularly sensitive operations. These logs should be assigned the same data classification as the highest level of data involved in their generation.
  - Resource: [Cloud Security Alliance - Security Logging Guidelines](#)
- **Implement real-time anomaly detection**
  - Use machine learning to detect unusual behavior patterns (using an ML enabled SIEM tool)

- 
- Deploy systems that can identify deviations from normal operation
  - Resource: [Awesome Anomaly Detection](#)
  - **Set up alerts for suspicious events**
    - Configure notifications for unusual patterns or security concerns
    - Alert on access attempts to sensitive endpoints, spikes in requests, unusual IP origins
    - Resource: [PagerDuty Incident Response Documentation](#)
  - **Develop incident response plans**
    - Create detailed procedures for addressing security breaches
    - Include containment, investigation, remediation, and recovery steps
    - Resource: [NIST Computer Security Incident Handling Guide](#)
  - **Build emergency off-switches**
    - Implement kill switches to immediately revoke access privileges
    - Create automated and manual mechanisms to stop agent operations
    - Resource: [Anthropic AI Safety Research](#)

#### 4.1.5. Prompt Security

- **Implement input validation**
  - Filter user inputs using rule-based patterns and NLP techniques or filtering for multi-modal processing
  - Check for malicious patterns before processing by the AI system (WAF Rules)
  - Resource: [Guide to LLM Prompt Injection Defenses](#)
- **Apply content filtering on AI outputs**
  - Screen AI-generated responses for inappropriate or harmful content

- Use both pattern matching and ML-based content classifiers
- Resource: [Perspective API for Content Moderation](#)
- **Harden system prompts**
  - Embed security policies directly into the AI's foundational instructions
  - Structure prompts to reject harmful requests and maintain guardrails
  - Resource: [OpenAI System Message Guidelines](#)
- **Sanitize all inputs**
  - Clean and normalize inputs to prevent injection attacks
  - Remove or escape special characters and formatting that could alter behavior
  - Resource: [OWASP Input Validation Cheat Sheet](#)

## 4.2. Multi-Agent Systems with Central Orchestrator

**Multi-Agent:** Multiple intelligent agents that interact with each other and a shared environment to solve problems or achieve goals. The agents can operate independently, discern their surroundings, make decisions, and act. The central orchestrator is responsible for coordinating the agents' interactions, assigning tasks, and maintaining the overall workflow or state.

### 4.2.1. Authentication & Authorization

- **Use authentication and authorization controls from 4.1.1**
- **Establish separation of control planes**
  - Create a clear separation between different agent functionalities
  - Utilize the user assumption architecture pattern from section 2.2.1 to enforce user permissions on an agent
  - Maintain distinct permission sets for each agent based on least privilege
  - Resource: [AWS Separation of Control and Data Planes](#)
- **Authenticate and verify all agent interactions (see section 4.2.3)**

- **Restrict discovery to known and trusted servers**

- Hard-code discovery servers in agent instructions
- Prevent agents from communicating with discovery servers using network and logical controls

#### 4.2.2. Orchestrator Security

- **Harden the central orchestrator's API**

- Implement robust authentication and authorization mechanisms
- Use rate limiting, input validation, and detailed logging
- Resource: [OWASP API Security Top 10](#)

- **Protect against control-flow hijacking**

- Validate agent responses to prevent manipulation of the orchestrator's decisions
- Implement integrity checks on operational metadata and agent outputs
- Resource: [MITRE ATLAS Framework for AI Threats](#)


- **Mitigate the "confused deputy problem."**

- Prevent scenarios where trusted components are tricked into performing unauthorized actions
- Implement additional context validation and cross-check requests
- Resource: [The Confused Deputy Problem](#)

#### 4.2.3. Inter-Agent Communication

- **Implement secure communication protocols**

- Use protocols with strong encryption for all agent interactions
- Implement mutual TLS (mTLS) for two-way authentication
- Resource: [Mutual TLS \(mTLS\) Authentication](#)

- 
- **Utilize well known protocols or pre-defined schemas**
    - Use JSON-RPC 2.0 and standardized event methods
    - Utilize schema based communications to simplify deterministic validation and improve consistency
    - Resource: [Google's A2A](#)
  - **Verify the identity of each communicating agent**
    - Use certificate-based or token-based authentication mechanisms
    - Resource: [JWT Authentication Best Practices](#)
  - **Use secure message queuing systems**
    - Implement message brokers with security features for asynchronous communication
    - Use systems like RabbitMQ, Kafka, or NATS with authentication and encryption
    - Resource: [RabbitMQ Security](#)
  - **Deploy Policy Enforcement Points (PEPs)**
    - Add security checkpoints to communication channels for policy enforcement
    - Implement as middleware or service proxies that validate each interaction
  - **Apply rate limiting for agent interactions**
    - Prevent abuse by limiting the frequency of requests between agents
    - Implement exponential backoff for failed requests
    - Resource: [GitHub API Rate Limiting](#)

#### 4.2.4. Trust Boundaries

- **Apply Zero Trust security principles**
  - Implement a "never trust, always verify" approach across the architecture

- Verify every access request regardless of source
- Resource: [NIST SP 800-207 Zero Trust Architecture](#)
- **Implement network segmentation and agent isolation**
  - Create distinct security zones within the system
  - Limit the interconnectedness of agents to contain potential breaches
  - Resource: [NSA Network Segmentation Guide](#)
- **Use containerization technologies**
  - Sandbox individual agents using Docker or similar technologies
  - Restrict access to underlying OS and system resources
  - Resource: [Docker Security Best Practices](#)

## 4.3. Multi-Agent Systems with Swarm Architecture

### 4.3.1. Authentication & Authorization

- **Use authentication and authorization controls from 4.1.1. and 4.2.1**
- **Identity trusted agents and actions**
  - Swarm systems don't necessarily have an easy guidepost for the correct action and trusted collaborators. Assign this trust and action outside the agentic flow
  - Limit or prevent the swarm from adding new agents without human guidance.

### 4.3.2. Decentralized Identity & Trust

- **Implement Decentralized Identifiers (DIDs)**
  - Use W3C standard DIDs for self-sovereign agent identities
  - Enable verification without a central authority
  - Resource: [W3C Decentralized Identifiers Specification](#)

- **Use Verifiable Credentials (VCs)**

- Implement VCs for agents to prove specific attributes
- Enable secure and verifiable proofs of capabilities or authorizations
- Resource: [W3C Verifiable Credentials Data Model](#)

- **Create decentralized reputation systems**

- Build systems where agents develop reputation scores based on behavior
- Use these scores to make trust decisions for collaboration
- Resource: [The OpenCerts Framework for Verifiable Claims](#)

How?

**Agent Identity:** Each AI agent can have its own DID, providing a persistent, verifiable identity across different interactions and environments.

**Credential-Based Trust:** Verifiable Credentials can establish an agent's capabilities, permissions, or trust level within the system.

**Inter-Agent Authentication:** Agents can verify each other's identities and credentials without centralized coordination.

**Provenance Tracking:** DIDs can help track the origin of information or actions in a multi-agent system, establishing clear provenance.

**Access Control:** VCs can determine which agents have access to specific resources or capabilities within the system.

**Role Specialization:** Credentials can formalize specialized roles among different agents (e.g., reasoning agent, retrieval agent, etc.).

#### 4.3.3. Secure Peer-to-Peer Communication

- **Select appropriate P2P protocols**

- Choose protocols based on specific security and performance needs
- Consider TLS/SSL, DTLS, Noise Protocol, or specialized frameworks



- Resource: [Noise Protocol Framework Specification](#)

- **Ensure both encryption and authentication**

- Implement strong encryption to protect confidentiality
- Use robust authentication to verify identity of participating agents
- Resource: [NIST SP 800-175B: Guideline for Using Cryptographic Standards](#)

#### 4.4. Cross-Architecture Security Considerations

- **Secure API Design for Agent Communication**

- Implement standardized interfaces with comprehensive input validation.
- Apply rate limiting and authentication regardless of architecture
- Resource: [OWASP API Security Top 10](#)

- **Continuous Security Testing**


- Regularly scan all agent architectures for vulnerabilities
- Implement automated and manual security testing processes
- Resource: [NIST Security Testing Guidelines](#)

- **Security Monitoring and Threat Detection**

- Implement comprehensive monitoring regardless of architecture
- Adjust detection strategies based on architectural specifics
- Resource: [MITRE ATT&CK Framework](#)

- **Secure Agent Updates and Maintenance**

- Implement secure update mechanisms for all architectures
- Verify integrity of updates before deployment
- Resource: [NIST Secure Software Development Framework](#)



This comprehensive list provides developers with practical security actions covering all major aspects of agentic AI system security across different architectures, along with relevant web resources for further reading.

## 5. Key Operational Capabilities

Connecting Agentic AI applications with other systems or environments such as APIs, databases, code interpreters, web browsers, or the operating system enables great functionality (skills). However, each new operational connection poses unique security concerns, ranging from data breaches and illegal actions to system compromise. The following listed controls are necessary to reduce these inherent risks.

### 5.1. KC6.1 – API Access

**Core Threats:** Unauthorized data access/leakage (T3), API abuse (DoS, cost overruns) (T4, T2), Compromised keys (T3, T9)

Controls:

#### 1. Enforce Least Privilege with Fine-Grained OAuth Scopes or Limited API Keys

- Define the minimum necessary activities for task completion (e.g., never grant an admin complete access to an agent)
- Grant access solely to essential assets with minimal access scope/issue access tokens only to necessary clients/agents
- **Implementation Options:**
  - **OAuth 2.0:** Establish specific OAuth scopes using services like:
    1. [Okta](#)
    2. [Auth0](#)
    3. [Keycloak](#)
    4. Cloud provider integrated services
  - **SDKs:** Utilize Authlib or Passport.js to streamline OAuth operations
  - **API Keys:** Employ provider features to enforce limits:
    1. [AWS API Gateway](#) allows restriction to specific API stages/methods
    2. Configure IP address restrictions or read-only access where available

- **Cloud IAM:** For agents operating as cloud services (e.g., AWS Lambda, Google Cloud Functions), assign specific IAM roles with restricted permissions

## 2. **Set up Allow/Deny Lists for APIs**

- Prevent manipulation via prompt injection or compromised tools
- Clearly define authorized domains or URL pathways
- **Implementation Approaches:**
  - Configure a list of permitted base URLs (Allow-List) or URL patterns
  - Authenticate target URLs against this list before executing external requests
  - Use an outward proxy or API Gateway to permit connections only to pre-approved destinations
  - For containerized agents, implement [network policies](#) to limit outgoing traffic

## 3. **Prefer API Templates Over Complete LLM-Generated API Calls**

- Mitigate risks of unsafe or malformed API calls
- Improve reliability by hard-coding unchanging/common fields

## 4. **Best Practices:**

- Use prepared templates where LLMs only populate established parameters
- Utilize templating libraries like [Jinja2](#)
- Leverage structured output capabilities with strictly defined schemas
- Define parameter types and requirements before sending requests
- Sanitize web-fetched content using tools like [DOMPurify](#)
- Implement [Content Security Policy \(CSP\)](#) when rendering content

## 5.2. KC6.2 – Code Execution

**Core Threats:** Arbitrary code execution (RCE)(T11, T3), Code injection (T11, T2), DoS via resource exhaustion (T4), Leakage of confidential information (T6)

Controls:



### 1. Implement Mandatory Sandboxing

- OS-level isolation, containers, VMs, WebAssembly, or cloud solutions
- Reference implementations:
  - [NVIDIA's WebAssembly Sandboxing](#)
  - [LangChain's E2B Data Analysis](#)

### 2. Perform Static Analysis on Agent-Generated Code

- Use tools like:
  - [Bandit](#) for Python
  - [Semgroup](#) for multi-language analysis
  - [CodeShield](#) for LLM-generated code

### 3. Enforce Resource Limitations

- Set strict CPU, memory, and execution time limits
- Implement timeouts for all operations

### 4. Additional Security Measures

- Use code signing or verification when possible
- Implement strict file system and network restrictions
- Use allow-lists for permitted commands
- Require Human-In-The-Loop (HITL) approval for high-risk operations
- Sanitize all executed code output
- Implement runtime monitoring during execution

## 5.3. KC6.3 – Database Execution (Queries and RAG access)

**Core Threats:** SQL injection (T2), Unauthorized data exposure/modification (T3), RAG data poisoning/sensitive data retrieval (T1, T5, T3)

Controls:

### 1. Use Managed Vector Databases with Access Controls

- Restrict data ingest to match the access control capability (never ingest data that can't be protected from unauthorized users)
- Apply data classification labels to enforce user privilege
  - i. Utilize specialized RBAC RAGs like [Elastic RAG](#)
  - ii. [Pinecone](#) with authentication platforms like [Clerk](#)
  - iii. Authorization platforms like [Aserto](#)

## 2. Implement Query Safety Measures

- Use parameterized or template queries/ORMs exclusively. Never construct SQL using string concatenation.
- Grant database access via least-privilege user accounts, eg SELECT only, limited tables, or row/column level security
- Filter/block dangerous SQL keywords (e.g., DROP, TRUNCATE)
- Validate all user inputs influencing queries - if possible, derive from controlled channels only vs context window.

## 3. RAG-Specific Controls

- Implement post-retrieval filtering to check for sensitive content (PII) before agent use
- Apply content verification before embedding into vector stores
- Rate-limit retrieval operations

## 4. Limitation (Excessive Retrieval)

### 5.4. KC6.4 – Web Use

**Core Threats:** Malicious web content (XSS, exploits)(T11), Leakage of confidential information (T6), Phishing/deception (T7), Server-Side Request Forgery (SSRF)(T2, T3), Access to user's browser data - Sensitive Data Exposure (T3)

Controls:

#### 1. Sandbox Browser Components

- Run in tightly controlled environments
- Avoid using extensions or operating directly in the user's browser

## 2. **Implement URL Security**

- Use URL filtering with allow lists
- Leverage Secure Web Gateway (SWG) or Data Loss Prevention (DLP)
- Perform reputation checks on target domains
- Enforce proper TLS verification
- Detects and blocks open redirects or excessive redirection chains. Further, use DNS-based filtering in addition to HTTP-level filtering.

## 3. **Apply Protective Measures**

- Use network segmentation to prevent access to internal systems (mitigating SSRF)
- Throttle web access rate to prevent abuse
- Restrict downloadable file types
- Implement content sanitization before processing
- Inspect embedded/linked content (e.g., iframes, JS-injected sources)

## 4. **Logging and Monitoring**

- Log all URL accesses and HTTP responses

# 5.5. KC6.5 – Controlling PC Operations (Filesystem, OS Commands)

**Core Threats:** Unauthorized file access/modification (T3), Arbitrary OS command execution (T11, T3, T2), Lateral movement (T3)

Controls:

## 1. **Apply Strict Isolation**

- Use OS-level sandboxing

- Run agents as restricted users with minimal privileges
- Consider dedicated containers or VMs for isolation

## 2. **Implement Access Controls**

- Define strict allow/deny lists for permitted OS operations and file paths
- Use virtual filesystem interfaces where appropriate
- Intercept OS command calls to enforce allow-lists

## 3. **Limit accumulation of sensitive data**

- To maintain its status, the sandbox should not accumulate data and sessions

## 4. **Enhance Monitoring and Oversight**

- Log all OS-level actions performed by the agent
- Consider HITL approval for critical operations
- Implement command validation before execution

# 5.6. KC6.6 – Operating Critical Systems (e.g., SCADA controls)

**Core Threats:** Catastrophic physical outcomes (T2, T3, T6, T7), Malicious control injection (T2, T6), False data influencing decisions (T1, T5)

Controls:

## 1. **Apply Maximum Isolation**

- Use air-gapped or highly segmented networks
- Implement physical security measures

## 2. **Enforce Multi-Factor Authentication**

- Switch to external flow to require MFA for specific sensitive actions where possible – do not use agent to mediate MFA flow or retain privileged session

- Implement mandatory HITL for every critical action, succinct and engaging enough to retain the user's attention each time (excessive/inefficient prompting results in indiscriminate approval)

### 3. Implement Safety Mechanisms

- Default agent permissions to read-only monitoring
- Use anomaly detection tuned to system safety parameters
- Adhere to industrial control security standards
- Ensure physical interlocks and emergency overrides exist

## 6. Agentic AI and the Supply Chain

### 6.1 Code Security

#### 6.1.1 – Third party libraries & frameworks

- Standard supply chain applies
- SCA scanning
- Hashes & package locks for 3rd party versioning
- SBOMs

#### 6.1.2 – LLM/Agent-generated code

- Utilize sandboxing for any agent-generated/run code (3.2.3)
- Validate packages before running generated code
- HITL for high-risk code execution operations
- Licenses & provenance should be validated for third-party libraries before installation

### 6.2 Environments & Development





### 6.2.1 – LLM / Logic System

- Use the same LLMs in lower environments & production for higher reliability
- Ensure the provenance of LLMs in a public registry (like Hugging Face) by verifying hashes and contributors
- Use LLM and SCA scanners to scan for both third-party package vulnerabilities and security issues within LLM code (where applicable)

### 6.2.2 – Version Control & Code Management

- Utilize version control with LLMs & logic systems to track behavior
- Implement version control on prompts & instructions to increase reliability and auditability of behavior
- Use commit IDs and hashes for data changes

### 6.2.3 – Permission management

- Don't give agents access to the repositories/data sources that build/deploy them
- Manage permissions to data sources outside the environments the agent is running in
- Use code to modify IAM/roles permissions for agents to review changes

## 6.3 Agent & Tool Discovery

### 6.3.1 – Agent Cards

- Identify trust relationships with agent cards and descriptions, and minimize trust with unvalidated agent systems
- Employ DIDs for agent verification and authentication

### 6.3.2 – Local vs Remote Agent Registries

- Use logical or network barrier between local & remote registries to prevent registry confusion and accidental agent boundary traversal
- Classify and separate private and public agents, data, and actions. Prevent agents from sharing data or performing actions across these boundaries

- Use trust relationships and controls like certificate pinning to ensure agents only communicate with approved registries and environments

## 7. Assuring Agentic Applications

Assurance strategies for agentic applications must extend beyond static testing. Consider these approaches:

### 7.1. Red Teaming Agentic Applications:

Red teaming simulates adversarial attacks to detect and address vulnerabilities in different scopes, starting with a specific application through the entire organization's security infrastructure.

Red teaming strategy for agentic applications requires complex interactions, autonomous decision-making, and integration with external systems. Proactive red teaming identifies possible attack vectors by assessing the key components of agentic systems, including not only prompt injection techniques, but also privilege escalation through tools, memory poisoning, and plan manipulation.

The [OWASP ASI Red Teaming Guide](#) for Agentic AI offers a risk-based methodology to handle security concerns with LLMs. It promotes a holistic approach that includes model review, implementation testing, infrastructure assessment, and runtime behavior analysis. Cross-functional coordination amongst data science, cybersecurity, ethical, and legal departments is critical. The guide encourages ongoing monitoring and iterative improvement in response to evolving AI threats.

#### **Prompt injection attacks and red teaming detection strategies:**

Prompt injection attacks target vulnerabilities in natural language processing systems by altering input to cause unauthorized actions. Such vulnerabilities might cause sensitive data leakage and bypass of defined guardrails. The use of red teaming tools can mimic multiple prompts to assess susceptibility to these attacks, utilizing technologies such as Promptfoo to automate the detection procedure. Continuous and proactive testing is required to anticipate new prompt injection tactics.

The [OWASP GenAI Security Red Teaming Guide](#) lists available tools and datasets in Appendix B.

New frameworks were released recently targeting agentic AI key components such as tool calling manipulation, privilege escalation, and memory poisoning.



Examples of such frameworks:

Tool	Description	URL
AgentDojo	A dynamic evaluation framework to evaluate prompt injection attacks and defenses for LLM agents. Contains targeted injection attacks, instructing the agent to execute malicious tasks using available tools.	<a href="https://agentdojo.spylab.ai/">https://agentdojo.spylab.ai/</a>
Agentic Radar	Agentic Radar is designed to analyze and assess agentic systems for security and operational insights. It helps developers, researchers, and security professionals understand how agentic systems function and identify potential vulnerabilities.	<a href="https://github.com/splx-ai/agentic-radar">https://github.com/splx-ai/agentic-radar</a>
Agent-SafetyBench	An evaluation benchmark built primarily to assess the safety of Large Language Model (LLM) agents. Comprises a set of prompts or situations designed to evaluate whether agents follow safety norms and prevent creating damaging, unethical, or risky content or actions.	<a href="https://github.com/thu-coai/Agent-SafetyBench">https://github.com/thu-coai/Agent-SafetyBench</a>
AgentFence	Red team tool equipped with multiple attack scenarios focusing on identifying vulnerabilities like	<a href="https://github.com/agentfence/agentfence">https://github.com/agentfence/agentfence</a>



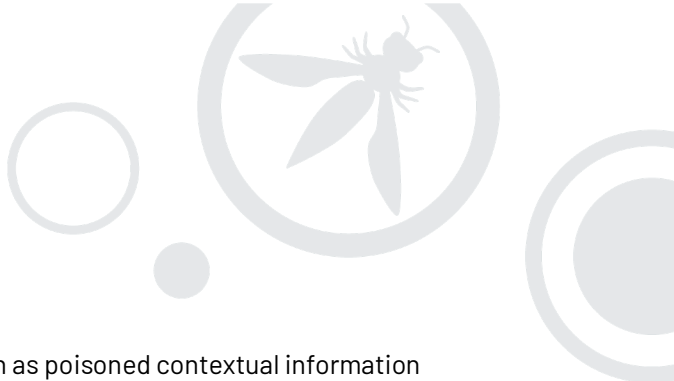
	prompt injection, role confusion, and system instruction leakage.	
Agent Security Bench (ASB)	Red team tool equipped with multiple attack scenarios including Direct Prompt Injections (DPI), Indirect Prompt Injections (IPI), Plan-of-Thought (PoT) Backdoor, and Memory Poisoning Attacks.	<a href="https://github.com/agiresearch/ASB">https://github.com/agiresearch/ASB</a>
Multilingual Benchmark for Agent Security (MAPS)	Security evaluation benchmark built to assess multi-lingual security of AI agent based on Multi-lingual adaptation of set of well known agentic benchmarks (like ASB) translated to 10 different languages.	<a href="https://huggingface.co/datasets/Fujitsu-FRE/MAPS">https://huggingface.co/datasets/Fujitsu-FRE/MAPS</a>
AgentPoison	Red teaming tool used for injecting malicious instances into the knowledge base or memory, to be retrieved when specific triggers appear in the input query.	<a href="https://arxiv.org/pdf/2407.12784">https://arxiv.org/pdf/2407.12784</a>

### Privilege Escalation in Agentic Applications and Red Team Methodologies

Privilege escalation attacks require acquiring illegal access to higher levels of control in a system. The ability of agentic AI to make independent decisions broadens the scope of potential attacks. Red teaming simulates situations to discover privilege escalation threats, such as leveraging inter-agent relationships and manipulating external integrations.

Privilege escalation can affect widely, from the application itself, the deployed infrastructure, to affecting the agent's connected agencies, which allow it access to the external environment and additional systems.

### Memory Poisoning Threats and Red Teaming Techniques for Identification:



Memory poisoning attacks can use either short-term memory, such as poisoned contextual information injected into the prompt, or through long-term memory and knowledge bases that introduce poisoned content that influences the agent's decision-making.

AgentPoison and ABS are red-teaming frameworks that employ memory poisoning attacks to discover such vulnerabilities, resulting in a high possibility of retrieving poisoned data.

## 7.2. Behavioral Testing for Agentic Applications:

Behavioral testing for AI agents emphasizes evaluating their activities and interactions rather than solely examining their underlying structure or code.

It is an essential assurance approach to ensure that agentic applications yield consistent and safe outcomes across diverse inputs and scenarios.

This testing approach aims to uncover hidden problems such as unexpected actions, logical errors, or the pursuit of harmful goals. Behavioral testing methods require a systematic approach that encompasses defining explicit objectives for the agent, **utilizing benchmark datasets** to determine performance metrics, executing simulations in **regulated environments** to analyze the agent's behavior under varying conditions, and employing both human and automated assessments to evaluate the quality and safety of the results.

### Benchmarks:

Behavioral testing depends on recognized benchmark datasets and task suites as fundamental elements for standardizing performance evaluation. These benchmarks offer various tasks, datasets, and metrics intended to examine the capabilities and behaviors of AI agents in varied contexts and operational environments.

Example:

[AgentBench](#): A comprehensive benchmark specifically designed to evaluate the task-solving abilities of agents across various domains, including operating systems, databases, and knowledge graphs. It assesses planning, decision-making, and tool usage behaviors.

[HELM](#) (Stanford): HELM evaluates models across multiple metrics, including accuracy, robustness, fairness, bias, and toxicity, using standardized scenarios. Its results provide insights into model behaviors related to truthfulness and potential harms.

[WebArena](#) & [The Agent Company](#): This benchmark evaluates autonomous agents performing tasks within realistic, complex web environments, testing behaviors like navigation, form filling, information retrieval, and adapting to dynamic website changes.

### Evaluation Framework:



A structured assessment framework offers a systematic process for implementing benchmarks and performing testing. These frameworks streamline the process of defining objectives, selecting test cases (often derived from benchmarks), executing tests in controlled or simulated environments, and analyzing results against expected behavioral standards (both positive and negative).

Examples:

[Inspect\\_evals](#) (UK AI Safety Institute): Open source framework to evaluate any generative model on any benchmark

[GenAI evaluation service](#) (Google): Evaluation framework for any generative model or application and benchmark the evaluation results against defined judgment, using the user's evaluation criteria.

[Bedrock Evaluations](#) (Amazon): Evaluation framework for LLMs and integration with external sources as RAGs.

For Agentic Application security, the first step in selecting an appropriate benchmark is determining the type of benchmark required. For Example:


Steps to take to make the right choice:

AI security benchmarks today differ widely in scope, evaluation methods, and threat coverage, leading to gaps in defense and inconsistent comparisons.

The [OWASP GenAI Red Teaming Guide](#) has already curated a list of AI security benchmarks, focusing on key threat vectors. Given the diversity and specificity of these benchmarks, a step-by-step guideline is necessary to help organizations effectively choose and apply the right benchmark for their unique security needs. This structured approach ensures that AI systems are tested in a comprehensive and reproducible manner, leveraging the curated benchmarks without redundancy, while addressing emerging vulnerabilities in a consistent, actionable way.

[Anthropic's initiative for third-party model](#) evaluations highlights that not all benchmarks are focused on AI security. Some assess general performance, like accuracy, rather than security vulnerabilities. Understanding the specific evaluation goal is crucial before selecting a benchmark. By identifying whether the focus is on performance, security, or robustness, organizations can choose the most relevant benchmarks, ensuring they align with the intended objectives and test AI systems for the right aspects.

The [EU AI Act's Code of Practice](#) also emphasizes safety and security measures for AI systems, aiming to mitigate risks associated with their deployment. AI security focuses on protecting systems from malicious threats, while AI safety ensures that systems operate as intended without causing unintended harm.



AI security benchmarks need meta-evaluation to uncover issues. Examples are inconsistency (variance introduced), ambiguity (benchmark misunderstood), incompleteness (missing user needs), or inaccuracy (bias introduced). This ensures benchmarks stay relevant, realistic, and genuinely reflect AI Application security.

By embedding lifecycle practices—scheduled updates, community peer review, and governance checks—the guideline keeps benchmarks up to date as adversarial techniques evolve. This structured, security-centric process ensures organizations select benchmarks that truly reflect and defend against the current AI threat landscape.

### **Step-by-Step Guide: Selecting a Security Benchmark for Agentic AI Applications**

1. **Define Security Objectives:** Identify your security goals based on AI security principles:

- **Confidentiality:** Protect training data and model parameters.
- **Integrity:** Ensure data and model integrity.
- **Availability:** Keep services operational.
- **Adversarial Robustness:** Protect models from adversarial attacks.
- **Privacy Protection:** Use techniques like anonymization.
- **Transparency and Accountability:** Ensure explainability and traceability.
- **Governance and Compliance:** Follow frameworks like NIST RMF.

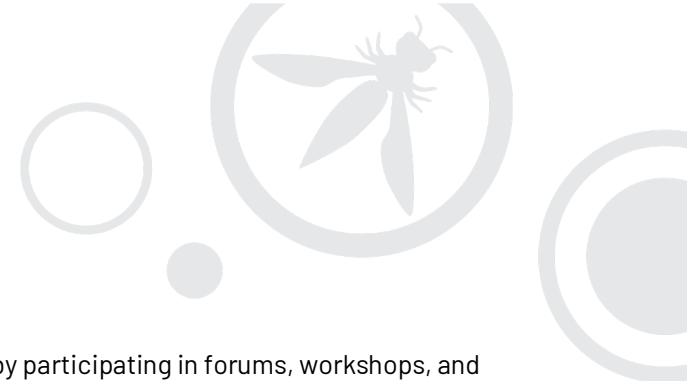
2. **Assess Threat Landscape:** Identify relevant threats, such as:

- **Data poisoning.**
- **Adversarial attacks.**
- **Model extraction.**
- **Privacy breaches.**
- **Denial-of-Service (DoS).**



- **Misuse of agent autonomy.**
3. **Research Existing Benchmarks** Look for benchmarks addressing agentic AI vulnerabilities, such as:
    - **Adversarial robustness.**
    - **Privacy protection.**
    - **Backdoor detection.**
    - **Model extraction security.**
  4. **Evaluate Benchmark Criteria.** Ensure the benchmarks:
    - Align with your security objectives.
    - Cover the relevant threats.
    - Are standardized and widely accepted.
    - Provide actionable insights.
  5. **Test Benchmark Suitability** Test benchmarks on representative models or datasets. Focus on:
    - **False positives.**
    - **Effectiveness under adversarial conditions.**
    - **Resilience under attack.**
  6. **Compare and Select** After testing, compare benchmarks based on coverage, relevance, and effectiveness. Select the one that best fits your needs.
  7. **Establish Continuous Evaluation.** Implement continuous evaluation to adapt to new threats:
    - Regularly re-test with updated benchmarks.
    - Integrate benchmarks into CI/CD pipelines.
    - Adjust the threat model as new risks emerge.



- 
8. **Engage with the AI Security Community.** Stay informed by participating in forums, workshops, and discussions. Engaging with the community provides insights on the latest security and benchmark developments.


### Explainability Audits

- Use CoT for some insight into agentic reasoning, understanding that what the agent says and “thinks” aren’t necessarily aligned. (<https://transformer-circuits.pub/2025/attribution-graphs/biology.html>)
- Utilize HITL at critical junctions to understand and gate agent actions. Criticality is defined by regulations like the EU AI Act.
- Use explainable AI (XAI) strategies like SHAPE, LIME, Grad-CAM, or T-SNE as needed by the model in use, if possible.
- Record agent-to-agent goal negotiations and tool interactions to understand additional inputs that may be misdirecting an agent. Especially for the more sensitive exchanges/tool calls, like database modifications.
- Log which permission sets or identities were used for agentic actions. This will indicate who the agent “thought” it was acting on behalf of.

## 8. Secure Agentic Deployments


Deploying LLM agents into production environments requires careful consideration of security beyond the model's inherent capabilities. While the specifics depend heavily on the application and risk profile, several principles and practices contribute to a more robust and secure agentic system. The following points provide a flavor of key areas to address:

- **Secure Pipelines and Checks for Rogue Agents:** Implementing rigorous CI/CD pipelines is important, but agent security requires more than standard code checks. These pipelines must incorporate automated vulnerability scanning of dependencies, static analysis focusing on potential insecure Agent’s tool usage patterns, and dynamic testing specifically designed to probe for prompt injection vulnerabilities or unintended goal deviation *before* deployment. Furthermore, incorporating code signing, provenance tracking for agent versions, and potentially manual security



reviews for agents handling high-risk tasks helps ensure that only vetted and approved agent logic reaches production, preventing deliberately malicious or inadvertently vulnerable agents from being deployed.

- **Role Containerization or Function-as-a-Service (FaaS):** To limit the potential blast radius should an agent instance be compromised, each agent or distinct agent role should operate within a strongly isolated environment. Utilizing containerization technologies (like Docker with Kubernetes) or FaaS platforms (like AWS Lambda or Google Cloud Functions) allows for strict resource limits (CPU, memory, network), network segmentation policies, and ephemeral execution contexts. This enforces the Principle of Least Privilege at the infrastructure level, preventing a compromised agent from easily accessing resources or impacting other agents or system components beyond its designated scope.
- **API Access Control, Rate-Limits, and API Gateways:** Agents heavily rely on external tools accessed via APIs, or other integration/interaction points such as MCP Server or Google A2A Agent Server via AgentCard, making these interaction points critical security boundaries. An API/Agent Gateway should mediate all tool invocations, enforcing strict authentication (verifying the agent's identity) and authorization (confirming the agent has permission to call that specific tool with those parameters). Implementing granular rate limiting per agent or per tool prevents denial-of-service attacks (intentional or accidental) and controls costs associated with API usage. The gateway also serves as a vital point for logging and auditing all tool interactions.
- **Alerting on Anomalous Behavior:** Continuous monitoring and alerting are essential for detecting potential compromises or malfunctions. Systems should baseline normal agent behavior (e.g., typical tool usage frequency and sequences, resource consumption) and trigger alerts upon significant deviations. Particular attention should be paid to unexpected or unauthorized tool use attempts, sudden shifts in the agent's apparent goal or task execution logic (which might indicate successful injection or manipulation), or anomalous patterns in data access or external communication, enabling rapid investigation and response.
- **Human Oversight in High-Stakes Environments:** Where critical decisions, high-risk actions, or significant deviations from expected behavior require explicit human review and approval before execution, implement robust human-in-the-loop workflow as full autonomy is often inappropriate and many tool invocations need human approval (for example, a large financial transfer or change affecting a human subject). In addition, if any untrusted data influences an agent with access to high-privileged tools (confidential information, integrity-sensitive systems), and for agents operating in domains with significant consequences (e.g., finance, healthcare, education, or critical infrastructure), oversight is key and should consider Human-Over-The-Loop (continuous supervision), see 3.3.1 Monitoring. Adaptive trust mechanisms can dynamically adjust the level of required oversight based on the agent's performance, context, and the risk associated with the



specific action. For a more complete definition of high-risk systems, review the [EU AI Act's Annex](#) on high-risk systems.

- **Managing Non-human Identities:** Each agent instance or service requires a distinct, manageable identity for secure interaction with APIs, tools, other agents, and other system components. These machine identities (e.g., service accounts, workload identities) must be treated with the same rigor as human identities, involving secure **provisioning processes**, robust credential management (including secure storage and regular rotation of keys/tokens using secrets managers), **clear ownership**, and reliable de-provisioning when an agent is retired to prevent orphaned, potentially exploitable identities.

## 9. Runtime hardening.

Securing **agentic applications running in a VM** means combining **traditional VM security hardening** with **agentic-specific controls** like sandboxing, auditability, capability constraints, and runtime behavioral monitoring. Here's a breakdown tailored for LLM-driven or tool-augmented agentic apps:

### 9.1 Harden the Virtual Machine (Base Level)

Before even talking about agentic specifics, the **host VM must be secure**:

#### **Baseline VM Hardening**

- Use **minimal base images** (e.g., distroless, Alpine, microVMs).
- Enable **Secure Boot**, **TPM-backed** disk encryption (e.g., via dm-crypt/LUKS).
- Disable all unnecessary services (sshd, cron, etc.).
- Apply **firewall rules (iptables/nftables)** to restrict ingress/egress by default.
- Patch regularly using automation (e.g., unattended upgrades or distroless image re-pulls).

#### **Network Isolation**

- Put the VM inside a **private subnet or VPC**.
- Use **Service Meshes** or **API Gateways** for controlled inter-agent communication.
- Prevent direct internet access unless explicitly needed (and monitor it)



## 9.2 Contain the Agentic Runtime

Now, focus on the **agentic app itself** running in the VM:

### ***Sandbox the Agent Processes***

- Run each agent or tool in **its own restricted namespace or container** inside the VM (e.g., via gVisor, Firecracker, or even Docker-in-VM).
- Use **AppArmor/SELinux** or **seccomp profiles** to restrict syscalls.
- Limit file system access using **mount namespaces** and **read-only volumes**.
- Inject ephemeral storage (e.g., tmpfs) for memory-sensitive LLM tools.

### ***Scoped Capabilities***

- Use an **AGNTCY-like capability control layer**:
  - Agents can only use explicitly declared tools (e.g., searchDocs, not shellExec).
  - Introduce `capability_tokens` scoped per session.
  - Use **JSON-RPC wrappers** (like MCP) to enforce capability boundaries.

---

## 9.3 Secure the Agent's Memory, Tools, and Context

### ***Memory and State Hygiene***

- Encrypt in-memory state (where feasible).
- Auto-clear memory on session end (e.g., clear Redis/Vector DB agents).
- Use **VM hooks** (e.g., systemd unit stop triggers) to flush memory or rotate keys.

### ***Tool Execution Control***

- Implement **runtime guards** for tool calls:

- Validate inputs and outputs.
    - Scan for prompt injection or malicious redirection attempts.
  - Use AGNTCY's policy engine to gate access dynamically:
    - E.g., only allow `call_api("billing")` from the "finance-agent."
- 

## 9.4 Observability + Forensics

### ***Audit Everything***

- Every agent action (tool use, messages, memory write) should:
  - Be logged with timestamp, agentID, sessionID, payload hash.
  - Include input/output signature fingerprints.
- Store logs securely with forward integrity (e.g., append-only + Merkle proofs or signed logs).

### ***Behavioral Anomaly Detection***

- Use **runtime policy engines** (e.g., Open Policy Agent) or integrate **LLM anomaly detectors** to spot:
    - Unusual tool invocation patterns.
    - Cross-agent communication deviations.
    - Memory bloat or frequent memory peeks.
- 

## 9.5 Identity, Authentication, and Agent Authorization

### **Identity & Session Management**

- Use **JWTs, mTLS, or HMAC-based auth** per agent session.
- Assign each agent a unique **ephemeral session context** using MCP.



### Role-based + Capability-based Access

- Combine RBAC (agent classes) with capability tokens.
- Example:
  - research-agent → can access docSearch, summarize.
  - ops-agent → can use issueCommand, getStatus.

---

## 9.6 (Optional) Cloud-Specific Hardening

If your VM is running in AWS/GCP/Azure:

- Restrict metadata service access (e.g., IMDSv2 in AWS).
- Use **IAM roles scoped per VM**, not global.
- Enable **Confidential Computing** if supported (e.g., AMD SEV, Intel SGX)



# Acknowledgements

---

## Contributors

Idan Habler, Intuit, Securing Agentic Apps Guide Co-lead  
Vineeth Sai Narajala, Meta, Securing Agentic Apps Guide Co-lead  
Rob Truesdell, Pangea, Securing Agentic Apps Guide Co-lead  
Tomer Elias, HUMAN Security  
Ben Diamant, HUMAN Security  
Lindsay Kaye, HUMAN Security  
Emile Delcourt  
Ron Bitton, Intuit  
Ryan Amos, Intuit  
Itsik Mantin, Intuit  
Joshua Beck, SAS  
Volkan Kutal, Commerzbank AG  
Ken Huang, DsistributedSystems.AI, OWASP AVSS  
Evgeniy Kokuykin, Raft  
Lindsay Kaye  
Sonu Kumar  
Keren Katz, Tenable  
Rafael Sandroni, GuardionAI  
Abhineeth Pasam  
Akram Sheriff, Cisco  
Sandy Dunn  
Victor Lu  
Roman Vainshtein, Fujitsu  
Omer Hofman, Fujitsu  
John Sotiropoulos, Kainos, Agentic Security Initiative Co-lead  
Ron Del Rosario, SAP, Agentic Security Initiative Co-lead

## ASI Review Board

Alejandro Saucedo - Chair of ML Security Project at Linux Foundation, UN AI Expert, AI Expert for Tech Policy, European Commission  
Apostol Vassilev - *Adversarial AI Lead*, NIST  
Chris Hughes - CEO, Aquia  
Hyrum Anderson - CTO, Robust Intelligence  
Steve Wilson - OWASP Top 10 for LLM Applications and Generative AI Project Lead and Chief Product Officer, Exabeam  
Scott Clinton - OWASP Top 10 for LLM Applications and Generative AI Project Co-Lead  
Vasilios Mavroudis- Principal Research Scientist and Theme Lead, the Alan Turing Institute  
Josh Collyer, Principal Security Researcher, Theme Lead  
Egor Pushkin, Chief Architect, Data and AI at Oracle Cloud  
Peter Bryan, Principal AI Security Research Lead- AI Red Team, Microsoft

# OWASP Gen AI Security Project Sponsors

We appreciate our Project Sponsors, funding contributions to help support the objectives of the project and help to cover operational and outreach costs augmenting the resources provided by the OWASP.org foundation. The OWASP Gen AI Security Project continues to maintain a vendor neutral and unbiased approach. Sponsors do not receive special governance considerations as part of their support. Sponsors do receive recognition for their contributions in our materials and web properties.

All materials the project generates are community developed, driven and released under open source and creative commons licenses. For more information on becoming a sponsor, [visit the Sponsorship Section on our Website](#) to learn more about helping to sustain the project through sponsorship.

## Project Sponsors:



Sponsor list, as of publication date. Find the full sponsor [list here](#).



# Project Supporters

---

Project supporters lend their resources and expertise to support the goals of the project.

Accenture	Cobalt	Kainos	PromptArmor
AddValueMachine Inc	Cohere	KLAVAN	Pynt
Aeye Security Lab Inc.	Comcast	IronCore Labs	Prompt Security
AI informatics GmbH	Complex Technologies	IT University Copenhagen	Quiq
AI Village	Credal.ai	Klavan Security Group	Red Hat
aigos	Databook	KPMG Germany FS	RHITE
Aon	DistributedApps.ai	Kudelski Security	SAFE Security
Aqua Security	DreadNode	Lakera	Salesforce
Astra Security	DSI	Lasso Security	SAP
AVID	EPAM	Layerup	Securiti
AWARE7 GmbH	Exabeam	Legato	See-Docs & Thenavigo
AWS	EY Italy	Linkfire	ServiceTitan
BBVA	F5	LLM Guard	SHI
Bearer	FedEx	LOGIC PLUS	Smiling Prophet
BeDisruptive	Forescout	MaibornWolff	Snyk
Bit79	GE HealthCare	Mend.io	Sourcetoast
Blue Yonder	Giskard	Microsoft	Sprinklr
BroadBand Security, Inc.	GitHub	Modus Create	stackArmor
BuddoBot	Google	Nexus	Tietoevry
Bugcrowd	GuidePoint Security	Nightfall AI	Trellix
Cadea	HackerOne	Nordic Venture Family	Trustwave SpiderLabs
Check Point	HUMAN Security	Normalyze	U Washington
Cisco	HADESS	NuBinary	University of Illinois
Cloud Security Podcast	IBM	Palo Alto Networks	VE3
Cloudflare	iFood	Palosade	WhyLabs
Cloudsec.ai	Intuit	Praetorian	Yahoo
Coalfire	IriusRisk	Preamble	Zenity

**Supporter list, as of publication date. Find the full supporter [list here](#).**