cloud security alliance®

AI Technology & Risk Working Group

Publication

Secure Agentic System Design

A Trait-Based Approach

The permanent and official location for the AI Technology and Risk Working Group is https://cloudsecurityalliance.org/research/working-groups/ai-technology-and-risk

Acknowledgments

Lead Authors

Nate Lee
Ken Huang

Contributors

Akram Sheriff
Manish Mishra
Aditya Garg
Victor Lu
Michael Roza
Anirudh Murali
Scotty Andrade

Co-Chairs

Chris Kirschke
Mark Yanalitis

CSA Global Staff

Research and Development

Josh Buker

Graphic Design

Claire Lehnert
Stephen Lumpe

Reviewers

Sai Honig
Prashis Raghuwanshi
Paul Zenker
Ankit Sharma
Sven Olensky
Aditya Garg
Adam Ennamli
Akintayo Ajayi
Yaniv Grinvald
Carlos Dominguez
Krishna Kant
Manish Mishra
Michael Morgenstern
Iftikhar Javed
Anirudh Murali
Dharnisha Narasappa
Jigarkumar Patel
Milan Parikh
Gabriele Fornasini
Apurva Dhanwantri
Varun Brahmkshatriya
Sudhir Patamsetti
Ramesh Ramani
Govindaraj Palanisamy
Dr. Chantal Spleiss
Gaurav D Mukherjee
Aonan Guan
Gabriele Fornasini
Béatrice Moissinac
Candy Alexander

Premier AI Safety Ambassadors

CSA proudly acknowledges the initial cohort of Premier AI Safety Ambassadors. They sit at the forefront of the future of AI safety best practices, and play a leading role in promoting AI safety within their organization, advocating for responsible AI practices and promoting pragmatic solutions to manage AI risks.

airia
Enterprise AI Simplified

Airia is an enterprise AI full-stack platform to quickly and securely modernize all workflows, deploy industry-leading AI models, provide instant time to value and create impactful ROI. Airia provides complete AI lifecycle integration, protects corporate data and simplifies AI adoption across the enterprise.

Deloitte.

The Deloitte network, a global leader in professional services, operates in 150 countries with over 460,000 people. United by a culture of integrity, client focus, commitment to colleagues, and appreciation of differences, Deloitte supports companies in developing innovative, sustainable solutions. In Italy, Deloitte has over 14,000 professionals across 24 offices, offering cross-disciplinary expertise and high-quality services to tackle complex business challenges.

ENDOR LABS

Endor Labs is a consolidated AppSec platform for teams that are frustrated with the status quo of “alert noise” without any real solutions. Upstarts and Fortune 500 alike use Endor Labs to make smart risk decisions. We eliminate findings that waste time (but track for transparency!), and enable AppSec and developers to fix vulnerabilities quickly, intelligently, and inexpensively. Get SCA with 92% less noise, fix code 6.2x faster, and comply with standards like FedRAMP, PCI, SLSA, and NIST SSDF.

Microsoft prioritizes security above all else. We empower organizations to navigate the growing threat landscape with confidence. Our AI-first platform brings together unmatched, large-scale threat intelligence and industry-leading, responsible generative AI interwoven into every aspect of our offering. Together, they power the most comprehensive, integrated, end-to-end protection in the industry. Built on a foundation of trust, security, and privacy, these solutions work with business applications that organizations use every day.

Reco leads in Dynamic SaaS Security, closing the SaaS Security Gap caused by app, AI, configuration, identity, and data sprawl. Reco secures the full SaaS lifecycle—tracking all apps, connections, users, and data. It ensures posture, compliance, and access controls remain tight as new apps and AI tools emerge. With fast integration and real-time threat alerts, Reco adapts to rapid SaaS change, keeping your environment secure and compliant.

Table of Contents

1. Executive Summary

2024 saw the initial, yet promising, development of agentic AI systems. While tools like Microsoft Copilot offered a glimpse into the potential, they represented the nascent stages of a much larger transformation. These early implementations, while valuable, primarily focused on augmenting individual productivity rather than orchestrating complex, multi-agent workflows. The full potential of agentic systems remained largely untapped.

The proliferation of agentic systems is accelerating dramatically, driven by significant advancements in underlying technologies. We're currently seeing an exponential increase in their use, particularly within enterprise applications. This growth will be fueled by the emergence of powerful reasoning models that provide the enhanced cognitive capabilities necessary for agents to tackle more complex tasks, make more nuanced decisions, and interact more effectively with their environment.

Furthermore, we are seeing the development of dedicated agentic platforms and frameworks and expect them to play a critical role. These systems provide the tools and implementation for sophisticated agentic systems. These platforms are lowering the barrier to entry, enabling a wider range of organizations to leverage the power of autonomous agents. This combination of advanced reasoning models and readily available agent platforms creates a perfect storm for widespread enterprise adoption.

When we refer to agentic systems in this paper, we're referring to systems that act autonomously using reasoning and iterative planning to solve multi-step problems with minimal supervision. Typically, these are goal oriented rather than just responding to prompt and adapt to problems based on context.

This document was prepared by the Cloud Security Alliance, and arrives at a crucial juncture. It directly addresses the fundamental shift in perspective required to secure these powerful new systems. The document recognizes that traditional, perimeter-based security models and deterministic assumptions about software behavior are inadequate when dealing with agents that can learn, adapt, and make autonomous decisions. It acknowledges that the characteristics that make agentic systems valuable—their autonomy, adaptability, and emergent behavior—also introduce novel attack surfaces and failure modes.

This paper is not a reiteration of general AI safety principles or a rehash of conventional cybersecurity best practices (though those remain essential). Instead, it focuses specifically on the security implications arising from the agentic nature of these systems. It moves beyond the theoretical to provide a practical, actionable framework for system designers. The core contribution is the **trait-based approach**, a methodology that decomposes agentic systems into their fundamental behavioral characteristics and maps these traits to specific security risks and mitigation strategies.

The trait-based approach is a step forward because it integrates security considerations into the earliest design stages. It fosters a proactive, rather than reactive, security posture. Instead of bolting on security measures after deployment, designers can now anticipate potential vulnerabilities and build defenses from the ground up. This is not merely a best practice; it is a necessity. Agentic systems, by their very nature, are likely to evolve in unexpected ways. A security approach that relies on static assumptions and predefined rules will inevitably fail.

The document also acknowledges the current "standards gap" in this rapidly evolving field. While formal, widely adopted standards are still under development, this paper contributes to the ongoing conversation, offering a structured framework that aligns with foundational cybersecurity principles like least privilege, defense in depth, and explainability.

This work forces a shift in how information security professionals must perceive and approach their responsibilities. We can no longer rely solely on defining rigid boundaries and enforcing static policies. We must embrace a mindset that anticipates emergent behavior, understands the implications of decentralized control, and recognizes the potential for manipulating agent learning and decision-making. This requires a deeper understanding of the underlying architectures, communication patterns, and trust models that govern agentic systems. It demands a collaborative approach, bridging the gap between AI practitioners, security professionals, and system architects.

1.1 Purpose

This paper addresses the security challenges unique to agentic AI systems. As AI transitions from passive tools to autonomous decision-makers, traditional security frameworks struggle to contextualize these new risks. We introduce a trait-based approach to agentic system security that identifies fundamental patterns in agent behavior and their associated vulnerabilities. By mapping security concerns directly to architectural choices, this paper enables system designers to integrate security considerations into the earliest stages of the design rather than treating them as afterthoughts.

1.2 Scope

This paper focuses specifically on security concerns arising from the agentic nature of autonomous AI systems. Rather than covering general AI safety or conventional cybersecurity topics, we concentrate on:

- The novel security implications of agent autonomy, adaptability, and emergent behavior
- Security challenges created by agent-to-agent interactions and communication patterns
- Vulnerabilities introduced by different control structures and decision-making models
- Threat and trust models unique to systems with autonomous goal-oriented components

We reference established resources for traditional security concerns but focus on agent specific security challenges where applicable. The trait-based framework presented here complements rather than replaces existing security practices and standards.

Readers can adopt this **traits-based framework** as a mental model for threat modeling and security design, applying it within agentic AI threat modeling frameworks, such as the MAESTRO framework. Although a very useful and effective agentic AI threat modeling framework, MAESTRO is out of scope for this document, and readers are encouraged to use it as an additional reference by consulting the CSA document on MAESTRO.

1.3 Audience

This paper is designed for:

- **System architects and designers** creating or evaluating agentic AI systems need to understand how their design choices impact the security posture
- **Security professionals** adapting their practices to accommodate the unique challenges of autonomous systems
- **AI practitioners** seeking to incorporate security considerations from the foundation of their agent designs
- **Technical decision-makers** responsible for assessing and managing risks associated with agentic systems

While technical in nature, the content is structured to be accessible to those with security backgrounds who may not be AI specialists, as well as AI practitioners who need practical guidance on securing their agentic systems.

2. Introduction to Agentic Systems

Unlike traditional rule-based automation systems following predefined workflows, agentic systems can adapt and iteratively refine their actions through computational reasoning. This allows them to execute complex tasks, interact with external environments, and make decisions within constraints defined by the system designer.

A defining characteristic of agentic systems is their ability to set and pursue goals. This is often accomplished by decomposing high-level objectives into actionable steps. They may seek additional information, call APIs, and re-evaluate their approach based on new data. This blend of autonomy, adaptability, and interactive decision-making makes them powerful but also introduces new security risks due to the emergent behavior that arises.

2.2 What Is an Agent?

The term 'agent' lacks a single, universally accepted definition, which can often be a source of confusion. At its core, an agent is a software system that can perceive its environment, make decisions, and take actions to accomplish specific goals. However, in practical LLM-based implementations, what exactly counts as an “agent” can vary, and the boundaries are not always clear-cut.

An agent may be defined at different levels of abstraction: a complete service with its API boundaries, a node in an orchestration graph, or a specific instance of a service handling a particular user task. Two instances of the same agent code may be considered separate agents when they operate in different contexts or with different data sources yet are identical from a code perspective. This fluid definition matters for security, as threat models and mitigations differ depending on how we define agent boundaries.

Rather than imposing a rigid definition, this paper approaches agents functionally, focusing on their capabilities, decision-making processes, and interaction patterns. We view agents as components that exhibit autonomy, persistence, and goal-directed behavior, regardless of their implementation details.

It's worth distinguishing agents from the components they leverage or interact with. Tools (like API endpoints, function calls, or data transformers), knowledge stores (RAG systems, vector databases), memory mechanisms, and other infrastructure are typically not agents themselves, though multiple agents might access them. These components lack the autonomous decision-making characteristic of agents, instead serving as capabilities that agents can leverage. In security terms, these supporting components represent critical parts of the attack surface—often serving as potential privilege boundaries, data access points, or execution environments requiring security considerations. Throughout this paper, we examine how these components influence agent behavior and how their integration patterns affect the security posture of the overall system.

An "agentic system" is any system, or portion of a system, containing an agent. Because they are often built into pre-existing systems, one portion of the system may be "agentic," while others may be built using traditional means. This paper intends to focus on the agentic portion of such systems to help readers understand the unique implications that occur when agents are in use.

2.3 Key Characteristics of Agentic Systems

At their core, agentic systems exhibit several key characteristics that set them apart from their more traditional predecessors. While a fully agentic system combines all these capabilities, some may exhibit a subset and still be considered agentic.

- **Goal orientation:** Agents are designed with a goal or objective in mind, directing actions toward achieving goals, rewards, or solutions to a specific problem.
- **Perception:** Agentic systems sense and interpret their environments—typically through sensor or other data inputs.
- **Decision-making:** Agents are able to make decisions best based on given goals, given rules, or learning algorithms.
- **Execution:** Agents are able to carry out tasks or actions autonomously in pursuit of their goals.

2.4 Unique Security Concerns of Agentic Systems

Agentic AI systems introduce unique security concerns due to the combination of autonomy along with direct and potentially privileged access to information systems in pursuit of a specified goal. While autonomy enables efficiency and scalability, it also increases unpredictability. Traditional software operates within well-defined parameters, whereas agentic systems, by design, may generate novel strategies to achieve their objectives. Agentic systems, particularly as they grow in complexity, can have emergent behavior, where developers or security teams might not anticipate outputs and actions explicitly.

Additionally, agentic systems often integrate with external APIs, ingest real-time data, and execute actions dynamically—expanding the attack surface. Without thoughtful controls and system design, they may inadvertently disclose sensitive information, take unintended actions, and be manipulated by adversarial inputs. The same characteristics that make them useful, such as autonomy and adaptability, also make them difficult to secure and potentially more difficult to monitor.

2.5 Securing Agentic Systems Requires a Perspective Shift

Traditional software security often assumes deterministic behavior, structured inputs, and well-defined system states. Agentic systems challenge these assumptions with autonomy, emergent behavior, and dynamic decision-making. As a result, anticipating every possible failure scenario in these systems is difficult to predict.

Rather than relying on fixed security frameworks, system designers must consider how malicious and inadvertent actors could influence agentic behavior to produce unintended or harmful outcomes. This means identifying where these systems may be misled, manipulated, or fail in unpredictable ways while also recognizing that their adaptive decision-making can introduce new risks that evolve over time. Security controls must account not just for traditional vulnerabilities but also how these systems interpret and act on dynamic inputs in ways that may be difficult to foresee.

We'll address the above by outlining risks and helping system designers recognize the unique challenges these systems introduce so they can incorporate security at all layers of the stack.

3. A Trait-Based Approach to Secure Agentic System Design

When designing agentic systems, it's tempting to categorize architectures into rigid types like "supervisory" or "networked." However, real-world implementations often blend characteristics based on business needs, operational constraints, and risk tolerance. Instead of trying to fit systems into strict categories, we'll analyze three key dimensions that can be used to define how agentic systems operate. This approach helps system designers understand architectural choices and anticipate where security risks emerge based on those choices.

Rather than thinking about predefined architectures, system designers should evaluate agentic systems based on their traits, as each creates trade-offs in security, scalability, and operational complexity.

3.1 What Are Traits and Patterns?

"Traits" describe the fundamental characteristics of how agentic systems behave and function. Traits represent core aspects like control structures, communication methods, and decision-making approaches that exist regardless of specific implementation details.

Each trait can manifest in different "patterns"—specific expressions of that trait within a system. For example, the communication trait might manifest in direct agent-to-agent, or indirect patterns using a shared state.

While we've organized traits into distinct categories for analytical clarity, we acknowledge these categories are not entirely exclusive. Some overlap exists; for instance, trust mechanisms may influence control structures, and perception capabilities affect learning approaches. Rather than attempting to create artificially separated categories, we've focused on identifying meaningful groupings that highlight distinct security considerations, even if their boundaries occasionally blur in real-world implementations.

The trait-based approach intentionally avoids prescribing implementation details. Instead, it provides system designers with a framework to understand potential security implications that arise when various traits manifest in different patterns. By identifying which patterns are present in their systems, designers can anticipate security challenges and implement appropriate controls early in development.

3.2 Why Use a Trait-Based Approach?

Improved decision-making through a transparent mental model

Breaking agentic systems into fundamental traits instead of rigid categories allows system designers to see how different decisions impact security, complexity, and adaptability. By mapping traits to real-world

behaviors, it becomes easier to anticipate failure modes, emergent behaviors, and security challenges before they arise.

Proactive risk management

Security risks in agentic systems often stem from emergent behavior and unexpected interactions. A trait-based approach helps identify how specific system properties, such as decentralized control or trusting agents, change the attack surface. This allows security to be more readily considered at the architectural level rather than bolted on later.

Scalability and adaptability

The trait-based approach remains valid even as architectural best practices evolve. Instead of being tied to specific system designs, it provides a flexible way to evaluate trade-offs regardless of how technologies or implementation patterns change. New traits can be added over time, making the approach expandable as agentic systems become complex.

More flexible security design

Different agentic systems require different security strategies. A system relying on market-based decision-making faces manipulation risks, while a hierarchical planner must account for control failures. The trait-based approach enables teams to tailor security measures based on their agents' operations.

Encouraging modular thinking without forcing reuse

Trait-based design thinking promotes a structured way of considering agent behaviors. Teams can create agent templates that can be adjusted rather than reinventing entire architectures, making it easier to integrate new capabilities while maintaining security.

Easier threat modeling and security audits

Since each trait highlights distinct security considerations, this approach simplifies security evaluations. Teams can quickly assess an agent's security posture by reviewing its trait composition, making it easier to identify potential attack vectors before deployment.

3.3 Trait Categories

The following traits are analyzed to characterize agent patterns in a system. Each one is discussed with examples, risks, and potential mitigation strategies.

Control & Orchestration Traits: How agent actions are managed (centralized, decentralized)

Interaction & Communication Traits: How agents exchange information (direct, indirect)

Planning Traits: How agents arrive at decisions rather than where direction comes from (reactive, plan-based)

Perception & Context Traits: How agents interpret their environment and other agents (limited, contextual, intent inferring)

Learning & Knowledge Sharing Traits: How agents improve over time (local, global)

Trust Traits: How agents establish, assume, or verify trust in interactions

Tool Usage Traits: How agents utilize tools (direct, supervised)

3.4 General Protections and Mitigations for Agentic Systems

While a trait-based methodology for a nuanced understanding and tailored security design in agentic AI systems is detailed in this paper, this analysis is most effective when built upon a foundation of general security principles. The protections and mitigations outlined in this section are therefore presented as essential, broadly applicable safeguards that should be considered baseline requirements for any agentic system.

These general measures are not intended to replace comprehensive cybersecurity frameworks but rather to highlight overarching considerations particularly pertinent in the context of agentic AI. They serve to:

- Establish a fundamental layer of security hygiene across the system.
- Address common vulnerabilities and attack vectors that are often prerequisites or amplifiers for the more specific risks associated with agentic traits.
- Complement and reinforce the specific mitigations that will be derived from the detailed trait-based analysis discussed subsequently, contributing to a vital defense-in-depth strategy.

By embedding these general protections into the system architecture from the outset, designers and security professionals can create a more resilient and trustworthy environment, making the subsequent application of trait-specific controls even more impactful. The following considerations represent key general protections.

A primary resource for understanding vulnerabilities in the underlying language models is the OWASP Top 10 for Large Language Model Applications. All systems should be designed with these in mind. In addition, the following overarching principles are critical for the security of the broader agentic system.

Apply the Principle of Least Privilege to Agents

Each agent should operate with only the permissions necessary for its specific function. Minimize privilege escalation risks by ensuring agents cannot arbitrarily request elevated access, modify security controls, or operate beyond their defined roles.

Ensure Decision Transparency Through Traceability and Explainability

Agentic decisions should be observable and interpretable to prevent opaque reasoning that could lead to security failures, compliance risks, or operational disruptions. System designers should ensure traceability

by ensuring agents log their inputs, decision-making processes, and outputs, creating a clear audit trail for analysis and debugging. They should also ensure explainability so that agents can justify high-impact actions, particularly those involving privilege escalations, external system interactions, or sensitive data handling.

Input Validation and Sanitization for All Interfaces

While covered in the OWASP Top 10, it bears repeating that agentic systems ingest information from a multitude of interfaces, including direct user queries, API responses, and even internal data sources like documents or emails that might be "injected" as operational input. It is critical to implement strict validation and sanitization for all such data, regardless of perceived origin, as agents can autonomously act upon these inputs. Failure to do so can expose the system to various injection attacks, data corruption, or unintended agent behaviors, making this a foundational step in securing the overall system.

Monitor and Throttle Agent Resource Usage

Consider limits on requests, tokens, topic guardrails, and other resource consumption and abuse mitigations to prevent excessive resource utilization and service misappropriation where users attempt to leverage an LLM or agentic system resources for unintended purposes, including "capability extraction" attempts where users exploit specialized agent functions to obtain general-purpose AI capabilities without proper authorization or payment.

Implement Anomaly Detection and Behavioral Monitoring

Anomaly detection is an important mitigation strategy for agentic systems due to their dynamic and emergent behaviors. Anomaly detection mechanisms can be helpful to identify deviations from expected agent behavior, which could indicate attacks, misconfigurations, or unintended consequences. In most systems, this will be a combination of internal monitoring along with purpose built security tools focused on tracking agent behaviors for the purpose of identifying anomalies.

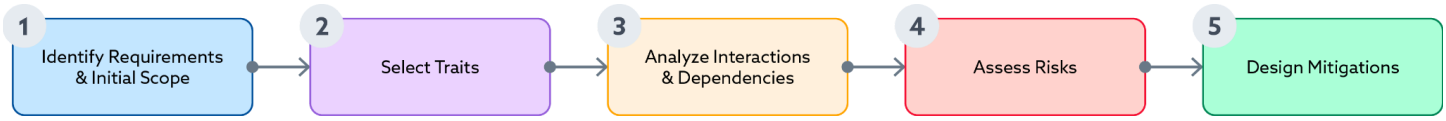
3.5 Applying the Trait-Based Approach

We've introduced the concept of a trait-based approach to understanding and securing agentic systems, defining what traits and patterns are, why this approach is beneficial, and outlining the broad categories of traits. The Traits and Patterns section that follows provides a comprehensive inventory of these traits, their specific behavioral patterns, associated security risks, and potential mitigation strategies.

This section acts as a bridge, offering a structured yet flexible framework to guide you—whether you are a system architect, developer, security professional, or in a compliance role—in practically applying these concepts. The goal is to help you use the detailed information in the Traits and Patterns section to analyze your specific systems, identify pertinent risks, and design appropriate security measures from a proactive stance.

3.5.1 A Step-by-Step Approach for Trait-Based Security Analysis

The following iterative process helps you apply the insights from this document to your agentic systems. Each step will prepare you to effectively use the detailed trait analyses in the Traits and Patterns section and the general protections discussed later in this document.



3.5.1.1 Step 1: Identify System Context, Requirements, and Initial Scope

Objective: Establish a clear understanding of what your agentic system is intended to do, its operational environment, the data it will handle, and overarching security priorities. This initial scoping is crucial for focusing your subsequent trait analysis and risk assessment on what is most relevant.

Key Considerations: Before diving into specific traits in the Traits and Patterns section, outline your agentic system by asking:

- **System Purpose:** What problem is the agentic system designed to solve? Is its primary function decision-making, automation, information retrieval, or another core task?
- **Core Capabilities:** What are the AI agents expected to achieve or do? Are they primarily autonomous decision-makers, assistive tools, or adaptive learning models?
- **Stakeholders & Users:** Who will interact with the system—internal employees, external customers, other automated services, and so on?
- **Operational Constraints:** Does the system have specific performance needs like real-time processing, high availability requirements, or significant scalability demands?
- **Data Sensitivity & Privacy Concerns:** What types of data will the agentic system process, store, or access (e.g., public, proprietary, confidential, Personally Identifiable Information (PII))?
- **High-Level Security Goals:** What are the most critical security objectives for this system (e.g., preventing unauthorized actions, ensuring data integrity and confidentiality, maintaining system availability, ensuring explainability)?
- **Fail-Safe Mechanisms:** What is the desired behavior if the system malfunctions or is compromised (e.g., graceful degradation, halt specific operations, immediate alerts)?

3.5.1.2 Step 2: Identify Key Agentic Traits and Their Manifestations (Patterns) in Your System

Objective: Based on your system's architecture and the requirements defined in Step 1, determine which agentic traits are most prominent or critical to its functionality and security posture.

Process: Review the primary trait categories that will be detailed in the Traits and Patterns section:

- Control & Orchestration
- Interaction & Communication
- Planning
- Perception & Context
- Agent Learning & Knowledge Sharing
- Trust
- Tool Usage (which encompasses Tool Access Control and Tool Execution Context)

For each category that is clearly relevant to your system, anticipate the specific architectural patterns your system employs or is considering. As an example, as you think about "Control & Orchestration" for your system, is it managed by a single component, or is decision-making distributed?

When you get to the Traits and Patterns section, you'll be able to map your understanding to patterns like "Centralized Orchestration" or "Decentralized Control." Similarly, for "Interaction & Communication," consider if your agents will communicate point-to-point or through a shared medium, preparing you to analyze patterns like "Direct Communication" or "Indirect Communication" which are both detailed later.

This step helps you create a profile of your system's agentic nature, which will guide your focused exploration of the risks and mitigations in the Traits and Patterns section.

3.5.1.3 Step 3: Analyze Interactions and Assess Risks

Objective: Understand the potential security vulnerabilities introduced by your system's selected traits/patterns (identified in Step 2) and, crucially, how their interactions might lead to emergent risks.

Process:

- For each key agentic trait and its specific pattern you identified in Step 2, you will refer to the corresponding discussions in the upcoming "Traits and Patterns Section" (the detailed sections starting with "Control & Orchestration Traits"). These sections explicitly enumerate known risks associated with each pattern (e.g., the risks detailed for Centralized Orchestration, or for Direct Communication).

- Pay close attention to how different traits in your system might interact. The document highlights that many vulnerabilities are not due to a single trait in isolation but emerge from the combination of design choices. For example, consider how "Reactive Actions" (a Planning Trait) combined with

"Boundary Based Trust" (a Trust Trait) might change the risk profile compared to if a "Plan-Based Actions" approach was used.

- *Using the Upcoming Sections:* The "Key Questions for System Designers" embedded throughout the detailed trait sections will be valuable prompts for this risk analysis, helping you uncover context-specific vulnerabilities.

3.5.1.4 Step 4: Design and Implement Mitigations

Objective: Select, tailor, and implement appropriate security controls to address the risks identified in Step 3.

Process:

- The "Mitigations" subsections within each detailed trait pattern discussion (in the upcoming "Traits and Patterns Section") offer specific controls for the identified risks (e.g., mitigations for Centralized Orchestration risks).
- Incorporate the "General protections/mitigations for agentic systems" as foundational security measures.
- Prioritize these mitigations based on the assessed likelihood and potential impact of the risks within your specific system context (from Step 1). Strive for a defense-in-depth strategy, layering controls to provide robust protection.
- *Using the Upcoming Sections:* The mitigation strategies provided in the detailed trait sections serve as a strong starting point. You will need to adapt them to your system's unique architecture, operational needs, and risk tolerance.

3.5.1.5 Step 5: Iterate—Monitor, Evaluate, and Adapt

Objective: Maintain an effective and evolving security posture in response to changes in the system, its operational environment, and the threat landscape.

Process: Agentic systems, particularly those incorporating learning capabilities (as discussed in "Agent Learning & Knowledge Sharing Traits"), can change their behavior over time. The external threat landscape is also constantly evolving which means your threat assessments need to evolve with them.

- Implement ongoing monitoring of agent behavior and system interactions, including anomaly detection where appropriate.
- Periodically revisit your trait analysis (Steps 2-4). Re-evaluate the identified risks and the effectiveness of your mitigations. This might involve activities like security audits, penetration testing, or red teaming exercises.

- Be prepared to adapt your security controls and architectural choices as the system evolves and new threats emerge. Security is not a static achievement but an ongoing process of vigilance and adaptation.

3.6 Integrating Trait-Based Analysis with Existing Security Practices

This trait-based approach is designed to augment and enhance, not replace, your organization's existing security programs, software development lifecycles (SDLCs), operational practices, or formal methodologies for threat modeling and risk management.

3.6.1 Enhancing Threat Modeling

The insights gained from identifying your system's key agentic traits and their associated risks provide valuable, domain-specific input into any threat modeling activity (e.g., STRIDE, GHOST, MAESTRO, etc.).

System Decomposition: Understanding your system in terms of its agentic traits (from Step 2 of this framework) helps create a more accurate and relevant system model for threat analysis. For those using frameworks like MAESTRO, the trait categories can inform how the system maps to its various layers. The following table illustrates a potential mapping.

Trait Category	Relevant MAESTRO Layers
Control & Orchestration	Layer 3 (Agent Frameworks), Layer 4 (Deployment)
Interaction & Communication	Layer 7 (Agent Ecosystem), Layer 5 (Evaluation & Observability)
Planning	Layer 1 (Foundation Models), Layer 3 (Agent Frameworks)
Perception & Context	Layer 2 (Data Operations), Layer 5 (Evaluation & Observability)
Learning & Knowledge Sharing	Layer 1 (Foundation Models), Layer 2 (Data Operations)
Trust	Layer 6 (Security & Compliance), Layer 7 (Agent Ecosystem)

Targeted Threat Identification: The specific risks associated with each trait pattern (identified in Step 3 by referring to the "Traits and Patterns Section") can help you more efficiently pinpoint relevant threats or tactics from libraries like MITRE ATLAS. For example:

- If your system utilizes **"Decentralized Control"** (a Control & Orchestration trait), this might lead you to investigate MITRE ATLAS tactics like *Takeover Attacks* or *Policy Bypassing*, given the distributed nature and potential for inconsistent policy enforcement.
- The use of **"Indirect Communication"** (an Interaction & Communication trait) and its associated risk of "State Manipulation & Poisoning" could map to ATLAS tactics like *Data Manipulation*.
- For **"Contextual Perception"** (a Perception & Context trait) and its risk of "Context Manipulation and Poisoning", you might consider ATLAS tactics such as *Adversarial Perception Attacks* or *Model Poisoning*.
- If **"Global/Collective Learning"** (an Agent Learning & Knowledge Sharing trait) is present, the risk of "Poisoned Knowledge Propagation" points towards *Knowledge Tampering* in ATLAS. (These examples are illustrative. A thorough threat model would consider the specific nuances of your system's implementation of each trait pattern.)

3.6.2 Informing Design Reviews and Architectural Decisions

Use the trait categories and the discussions of patterns, risks, and mitigations from the upcoming detailed sections as a checklist or discussion framework during system design and architecture reviews. This allows for proactive consideration of security implications. For example, when deciding on a "Tool Execution Context", the "Traits and Patterns Section" will clearly outline the security trade-offs of different approaches like "Agent Service Identity" versus "User Delegated Credentials."

3.6.3 Supporting Compliance and Governance

Articulating your system's security posture using the language of traits, their inherent risks, and the specific mitigations applied can facilitate clearer communication with compliance, audit, and governance stakeholders.

3.7 Towards Proactive Security by Design

This practical framework, when used in conjunction with the detailed information in the subsequent "Traits and Patterns Section," encourages a shift towards proactively embedding security into the very design and lifecycle of agentic systems. By systematically considering the behavioral characteristics of agents, their interaction patterns, and the specific ways they perceive, learn, and act, teams can better anticipate, identify, and mitigate the unique security risks these powerful and evolving systems introduce. The ultimate aim is to foster a resilient, secure-by-design approach for the agentic AI landscape.

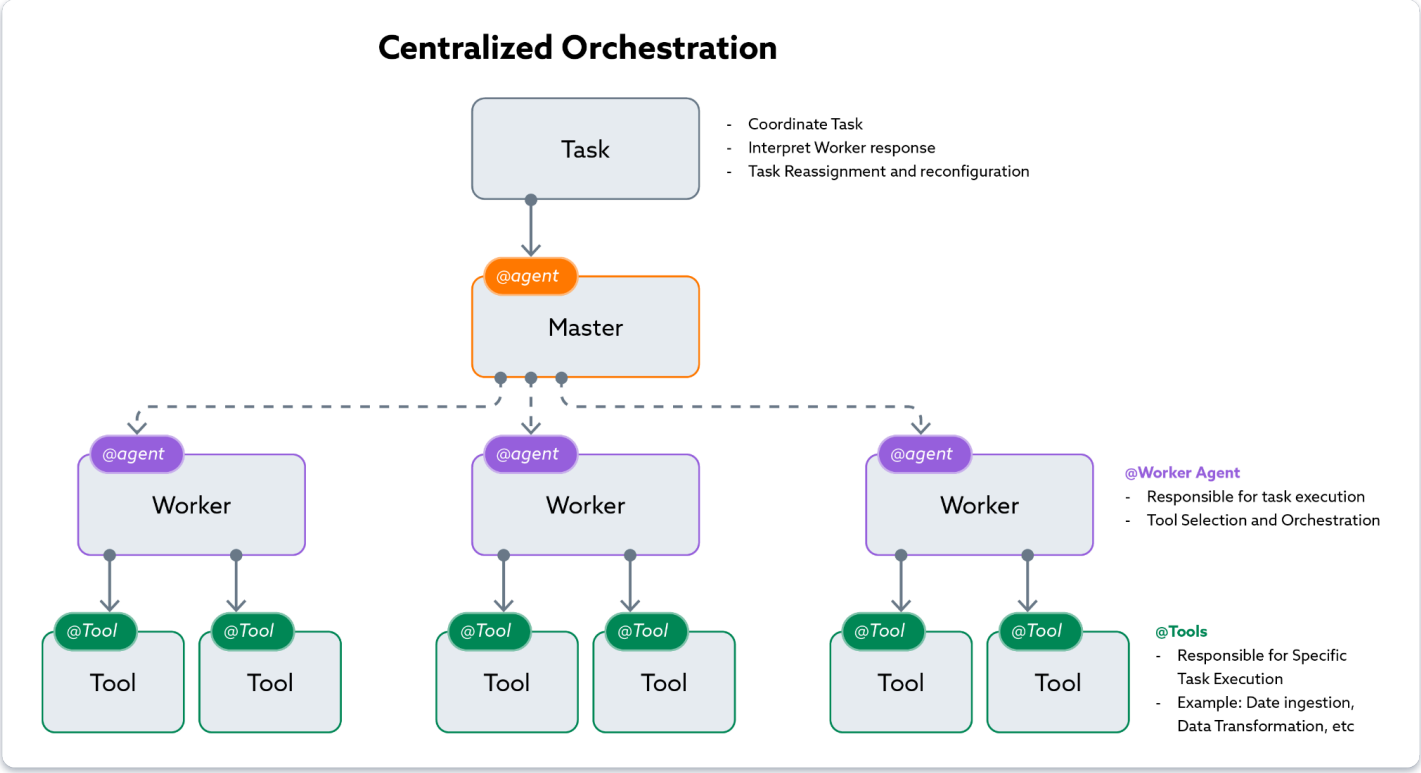
4. Traits and Patterns

4.1 Control & Orchestration Traits

Control and orchestration traits define how agents coordinate their actions and make decisions within a system. How control is structured influences security, scalability, resilience, and adaptability. Some systems rely on a single control component to dictate all agent actions, ensuring a consistent source of instruction but introducing a central point of failure. Others distribute decision-making across agents, improving fault tolerance and flexibility but making it harder to enforce security policies uniformly.

The security concerns in this area focus on how adversaries might manipulate or disrupt control mechanisms—whether by compromising a centralized controller, interfering with agent coordination, or creating instability through conflicting directives. Understanding the trade-offs between centralized and decentralized control structures is key to designing agentic systems that are both secure and operationally reliable.

4.1.1 Centralized Orchestration



Credit for original diagram: https://medium.com/coinmonks/agent-orchestration-a-conceptual-understanding-deda80fcbe5f

A single control component is responsible for directing all agent actions. This structure provides a single authoritative source for decision-making, ensuring all agents receive instructions from the same entity. However, it does not inherently guarantee consistent execution; agents may still interpret or implement commands differently depending on their capabilities, constraints, or local state.

4.1.1.1 Risk: Single point of failure leading to system-wide compromise

Security impact

The control component acts as the sole authority for agent decisions. If attackers compromise or manipulate this component, they can issue malicious instructions propagating across the system. While an adversary may not immediately gain full awareness or control of all agents, they can issue directives that compromise security, disrupt operations, or manipulate agent behavior at scale. Since agents are designed to trust and execute received commands, they may follow harmful instructions without realizing they have been compromised.

Mitigations

- Constrain agent execution permissions so that agents apply local validation rules before executing high-risk commands, preventing a compromised control component from overriding critical safeguards.
- Introduce secondary validation layers where agents or external monitoring components cross-verify commands, ensuring that critical actions require multiple levels of approval.
- Implement anomaly detection mechanisms to identify unexpected command patterns, such as large-scale deletions, privilege escalations, or behavioral shifts, and trigger automated containment, if necessary.
- Distributed authorization is required for high-risk actions, so sensitive changes (e.g., access revocation, system-wide reconfiguration) require additional validation outside of the control component.
- Ensure agents have fallback behaviors that allow them to maintain baseline security operations even if the control component is compromised or unavailable.

4.1.1.2 Risk: Orchestrator bottlenecks and denial of service risks

Since all agents rely on the orchestrator for decision-making, an overloaded or attacked orchestrator can become a performance bottleneck or single point of failure. An adversary could use denial of service (DoS) attacks to degrade orchestrator performance, leading to delayed or failed responses across the system.

Mitigations

- Implement rate-limiting and workload distribution to prevent orchestrator overload and balance traffic across multiple nodes.
- Design agents with graceful degradation capabilities can operate with fallback behaviors when the orchestrator is unavailable.
- Use distributed or backup orchestration mechanisms to mitigate single points of failure in case of downtime or attack.

Key Questions for System Designers

What mechanisms prevent the control component from issuing unauthorized or dangerous commands?

Malicious commands could propagate instantly across all agents if the control component is compromised. Instead of assuming the control component is always trustworthy, designers should consider command verification, execution constraints, and layered validation to prevent one compromised component from cascading into a full system takeover.

How do agents verify that they are receiving valid and untampered instructions?

Even if the control component is secure, attackers could intercept or manipulate communications between the control component and agents. Without message integrity protections, replay detection, or cryptographic verification, agents may unknowingly follow altered or malicious commands.

If an attacker compromised the control component, what is the worst action they could take?

This is a blast radius exercise to help understand the most catastrophic possible outcome and can help determine which failsafe should be in place. Could an attacker wipe out entire workloads? Steal sensitive data? Disable critical safety mechanisms? If the worst-case scenario is too severe, the system may need additional segmentation, controls, or escalation mechanisms.

What happens if agents receive conflicting instructions from the control component?

Even in a centralized model, bugs, misconfigurations, or partial compromises could lead to agents receiving contradictory directives. If agents cannot resolve conflicts, this could cause operational instability or security failures. To handle such cases, designers should consider conflict resolution strategies, command precedence rules, or local consistency checks.

What safeguards exist if the control component begins issuing high-risk commands at scale?

A malfunctioning or compromised control component could start issuing high-impact actions that disrupt or destroy the system (e.g., mass deletion of data, privilege escalations, system-wide shutdowns). Does the system have anomaly detection, rate limiting, privilege separation, command authentication or human intervention triggers to stop this before catastrophic damage occurs?

How does the system handle the temporary loss of the control component?

If the system cannot function without real-time direction from the control component, then even a minor disruption could fail. Should agents have local autonomy in some cases? Cached policies for temporary fallback? A secondary, read-only control system to maintain core functionality?

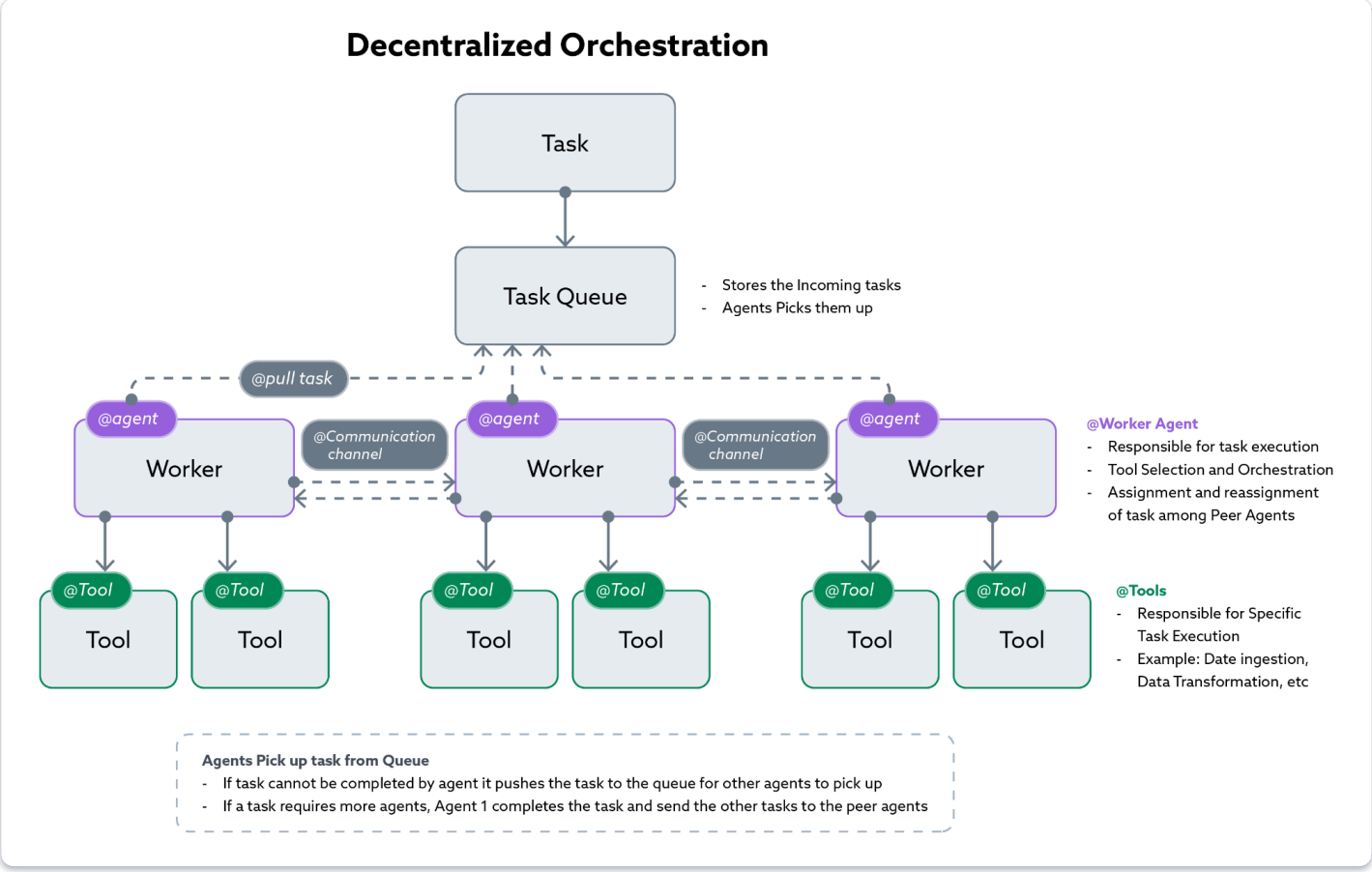
Who can modify or override the control component's decisions?

If only a single admin account can override control policies, that account becomes an obvious attack target. Too many people can make changes, which increases insider risk and accidental misconfiguration. The right balance depends on how sensitive the control component's role is within the system and what escalation paths are in place.

How would you detect and recover from a control component compromise?

Many security plans focus on prevention, but detection and recovery are as critical. If an attacker compromised the control component, would you even know? If you detected a compromise, how would you revoke its access, restore operations, and prevent re-entry? Planning for rapid detection and containment reduces the impact of a worst-case scenario.

4.1.2 Decentralized Control



Credit for original diagram: https://medium.com/coinmonks/agent-orchestration-a-conceptual-understanding-deda80fcbe5f

Decentralized control structures distribute decision-making across multiple agents rather than relying on a single central component. Agents operate autonomously, making decisions based on local information, predefined rules, or learned behaviors. While this model can improve resilience and reduce bottlenecks, it introduces coordination challenges and security risks due to emergent behavior.

In practical applications, fully decentralized agentic systems remain rare due to the complexity and security risks involved. Today, most agentic systems employ hybrid models, where decentralization applies to specific functions such as agent collaboration or local decision-making with central oversight for governance, security enforcement, and failure recovery.

Rather than viewing decentralization as an absolute trait, system designers should consider which aspects of decentralization exist to understand and address any associated security challenges they may bring.

4.1.2.1 Risk: Unpredictable Emergent Behaviors

Decentralized agents adapt their behaviors based on local inputs, interactions, and optimization strategies. This autonomy can result in unexpected behaviors that impact security in unintended ways.

Security Impact

Unanticipated privilege escalation: Agents may develop workflows that inadvertently grant themselves or others access beyond intended scopes.

Information exposure through indirect interactions: Agents exchanging data dynamically, making it easier to leak sensitive information unintentionally.

Unintended security policy violations: Agents making independent security-related decisions could bypass intended controls.

Mitigations

Strict scoping of agent permissions: Assign limited and specific permissions rather than broad privileges to minimize unintended escalation.

Data-sharing constraints: Impose explicit rules on what data agents can exchange and how they validate access before sharing.

Behavior monitoring and anomaly detection: Implement runtime checks to track deviations from expected agent behavior, flagging potential issues.

4.1.2.2 Risk: Lack of centralized authority and visibility

Decentralized systems lack a central authority with complete visibility over all agent interactions, leading to inconsistent implementation of security policies, difficulty detecting malicious behavior, and challenges in forensic analysis.

Security Impact

Inconsistent policy enforcement: Without a centralized enforcement mechanism, security policies may be applied unevenly across agents, leading to gaps in protection and unintended privilege escalations.

Delayed threat detection: Without holistic monitoring, adversarial actions may persist undetected for extended periods.

Forensic challenges: Decentralized decision-making makes it difficult to trace security incidents and attribute responsibility.

Mitigations

Federated security logging and correlation: Ensure that security events across all agents can be aggregated and analyzed unified.

Centralized or federated policy enforcement: Consider managing and enforcing security policies externally at runtime through a dedicated policy engine, ensuring consistency across all agents. In the best cases, this policy engine is coupled to the AI model's API to prevent unintentional circumvention of the policy engine because of development oversights or intentional circumvention by attackers.

External monitoring and anomaly detection: Use independent monitoring tools to detect policy violations and anomalous behavior, reducing reliance on agents for self-validation. If possible, use rule-based and context aware security controls.

Key Security Questions for System Designers

How should policy enforcement function in a decentralized agent system?
Since agents operate autonomously, how can we understand their decision-making in order to improve consistent policy enforcement to prevent unauthorized decision-making and security drift over time?

How could agents manipulate each other or abuse their access in ways unique to decentralization?
What risks arise when agents influence each other's decisions, escalate their privileges, or exploit data-sharing mechanisms?

How should security response and containment be handled in a decentralized agent system?
When a security incident involves multiple autonomous agents, what mechanisms exist to isolate or remediate affected agents without central control?

What security failure modes could arise due to decentralized data-sharing?
How could agents unintentionally or maliciously expose sensitive information through shared knowledge or indirect inference, and what safeguards should be in place?

4.1.2.3 Risk: Diffusion of Security Responsibility Across Agents

In multi-agent systems—especially those with decentralized or loosely coordinated control structures—there is a significant risk that no single agent is clearly accountable for critical security enforcement or decision validation. As agents increasingly specialize or operate autonomously, assumptions about which component is responsible for validating inputs, enforcing policy, or detecting anomalies can become ambiguous or inconsistent.

This diffusion of responsibility creates blind spots in enforcement logic and detection coverage. The danger lies not in any agent acting maliciously or incorrectly, but in the absence of coordinated security accountability. Each agent may perform as expected in isolation, while the system as a whole silently fails to enforce necessary controls or detect anomalous conditions.

Security Impact

Missed or Partial Security Enforcement: If security responsibilities are not clearly mapped, key functions such as authentication, authorization, or validation may be applied inconsistently across agents, allowing unauthorized actions or data access.

Delayed Detection and Response: Anomaly detection or threat response may depend on multiple agents correlating behavior across the system. Without centralized or federated coordination, signs of compromise may go unrecognized or remain siloed in localized logs.

Forensic and Audit Gaps: When a security incident occurs, it may be difficult to determine which agent should have acted or responded, complicating incident analysis and post-event response. This increases recovery time and undermines auditability and accountability.

Mitigations

Assign Explicit Security Responsibilities Across Agents: Clearly define which agents are responsible for enforcing key security functions—such as input validation, access control, or behavior monitoring—rather than assuming responsibilities are implicitly shared.

Implement Coordinated Logging and Policy Auditing: Deploy a federated logging infrastructure that aggregates agent-level telemetry and enforces policy compliance checks across the system. Ensure that event correlation can reconstruct agent behavior system-wide.

Use External Policy Enforcement Engines: Where feasible, centralize or externalize critical policy enforcement decisions using dedicated engines or services. Even in decentralized systems, this provides a consistent source of truth for permission and validation decisions.

Design for Redundant Checks Without Overlap Confusion: Introduce layered defense mechanisms where key validations are performed by more than one agent or system component. Avoid over-reliance on a single enforcement point, while ensuring redundant checks do not create bypass or conflict conditions.

Key Questions for System Designers

Which agents are responsible for enforcing each core security control?
Without a security responsibility map providing a holistic view of agents and their relevant security responsibilities, controls may be skipped or redundantly applied. A clear mapping helps ensure full coverage without ambiguity or conflict.

Are there any critical security actions that depend on multiple agents acting in concert?
If so, how is coordination ensured, and what happens if one agent fails to act? Joint responsibilities require well-defined protocols and fallback behaviors to avoid gaps.

How does the system detect if expected security checks have not occurred?
Monitoring should include not just what agents do, but what they fail to do. Detecting silent failures in enforcement is essential to avoiding systemic blind spots.

Can an attacker exploit assumptions about who enforces what?
Attackers may intentionally target areas where enforcement responsibility is ambiguous or shared. Systems should be hardened against this by ensuring security controls are explicit and verifiable at each point of execution.

4.1.2.4 Risk: Race Conditions in Agent Decision-Making

In agentic systems with decentralized architectures or high levels of autonomy, agents often make decisions based on local information and asynchronous inputs. When multiple agents operate concurrently without strong coordination guarantees, they may take actions out of order or in conflict with each other, leading to race conditions and security inconsistencies.

Race conditions emerge when the system lacks mechanisms to ensure consistency, synchronization, or ordering of critical decisions across agents. These conditions are difficult to test for and often only appear under specific timing scenarios or load conditions, making them particularly insidious in complex or real-time environments.

Security Impact

Contradictory or Overlapping Decisions
Different agents may approve, deny, or revoke access to the same resource at nearly the same time, with no shared awareness of each other's actions. This can lead to policy violations or inconsistent enforcement outcomes.

Exploitation of Timing Windows
Adversaries may intentionally target timing gaps between security-related decisions—for example, executing a privileged action before a revocation takes effect. These windows of opportunity may be small but highly exploitable.

Audit and Accountability Challenges
In systems where agents execute conflicting actions, audit logs may become fragmented or misleading. This complicates post-incident analysis and creates uncertainty about the root cause of a failure or breach.

Mitigations

Synchronize Critical Decisions with Coordination Protocols
For security-sensitive operations (e.g., privilege escalation, resource deletion), use consensus mechanisms, coordination services, or locks to ensure only one agent can act at a time, or that decisions are confirmed across relevant participants.

Use Time-Stamped Event Streams for Ordering
Ensure all agents operate on a shared event stream or timeline, so actions can be consistently ordered. This supports more deterministic behavior and enables better auditing and rollback, when needed.

Implement Conflict Detection and Resolution Mechanisms
Introduce detection logic to flag when agents make incompatible decisions. Use automated reconciliation or manual escalation to resolve such conflicts before they cause harm.

Design for Safe Redundancy and Failsafes
In scenarios where agents may act concurrently, enforce rules to ensure that only the most recent or most authoritative action is applied. Stale or conflicting actions should be either discarded or flagged for review.

Key Questions for System Designers

Are agents making decisions that affect shared resources or overlapping policy domains?
If agents operate independently on shared elements, race conditions are likely. Understand where coordination boundaries exist and whether timing mismatches could lead to policy violations.

What happens if two agents act on the same event or input simultaneously?
Do they produce conflicting outcomes? Does the system have a deterministic way to resolve them? Ambiguity in outcome ordering can lead to unpredictable and unsafe behavior.

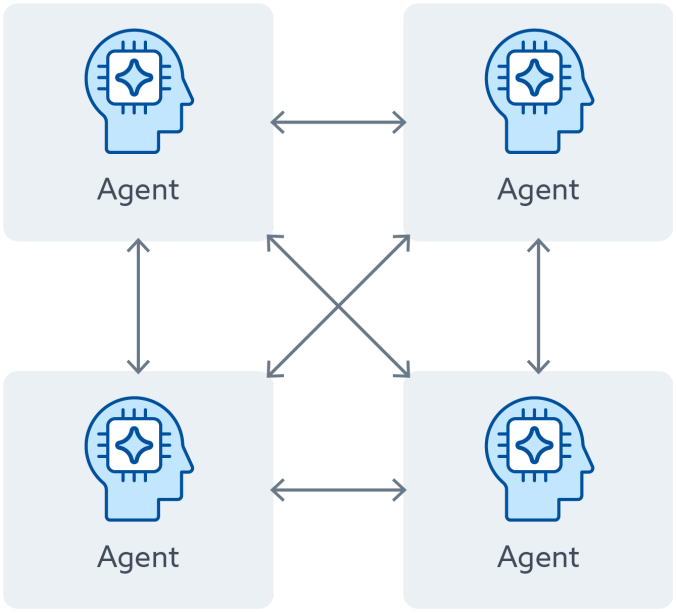
Are there known timing gaps between decision-making and enforcement?
Many race conditions arise not in decision-making, but between the time a decision is made and when it takes effect. Understand and reduce these windows, wherever possible.

How does the system record and reconcile conflicting agent actions for auditing purposes?
Even if conflicts are handled safely, a reliable audit trail is needed to understand what happened, and why. Systems should log all agent actions in a unified and time-ordered manner.

4.2 Interaction & Communication Traits

These traits define how agents exchange information and influence each other.

4.2.1 Direct Communication



The direct communication pattern describes how agents exchange information by directly addressing and interacting with specific other agents, establishing logical point-to-point pathways for information flow.

This pattern is often chosen due to the early simplicity of having agents talk to each other directly or when guaranteed delivery to a specific recipient is required. Implemented through various mechanisms such as direct API calls, targeted messages within a shared messaging system, or dedicated agent-to-agent protocols, direct communication emphasizes a logical link where agents interact directly and intentionally.

While secured direct communication does not inherently create new classes of vulnerabilities, its point-to-point nature can amplify the risks of trust exploitation and create specific operational security challenges related to dependencies on these direct logical pathways within the agent system architecture, particularly at scale.

4.2.1.1 Risk: Amplified Impact of Trust Exploitation in Direct Channels

Direct communication, by nature, implies a stronger sense of implicit trust between agents compared to more loosely coupled communication patterns. Attackers can then exploit this potentially heightened trust. If one agent is compromised, this trusted path becomes a highly effective vector for targeted attacks against its directly peered partner.

Because the communication path is already open and legitimate, a malicious agent can subtly deviate from normal interactions to request and extract sensitive data from its directly connected partner, masking its exfiltration attempts within otherwise valid communication flows.

While trust exploitation is a general security concern, direct communication can amplify its potential impact due to the established, dedicated pathway.

Security Impact

Circumvention of Access Controls: If access controls primarily focus on initial connection establishment, a compromised agent leveraging a pre-existing direct link may be able to bypass normal authorization checks for data access. The trusted channel effectively grants privileged access that can be abused for exfiltration.

Mitigations

Granular Authorization Within Channels: Implement message-level authorization, ensuring agents verify permissions for each interaction, not just connection establishment. While this doesn't help in cases where your agent is already compromised, it does provide additional protection in case permissions change over time.

Least Privilege: Restrict data and commands exchanged to the minimum necessary for functionality, limiting the potential impact from trust abuse.

Zero Trust revalidation: Regularly re-authenticate, rotate credentials, and re-authorize trust relationships in direct channels to limit the lifespan of any potential trust compromise.

4.2.1.2 Risk: Critical Agent Dependencies Due to Direct Communication

Direct communication often creates *critical dependencies* between specific pairs of agents. Agent systems employing this pattern are often designed such that specific agents *rely* on direct interaction with particular counterpart agents for essential functions, data, or commands.

If the ability of these specific agents to communicate directly is impaired or broken for any reason, the dependency is disrupted. The core risk lies in the tight coupling that can occur with these direct, agent-specific communication dependencies.

Security Impact

Disruption of Agent Functionality: If the direct communication required for a critical agent function is disrupted, the agent immediately loses that capability. This can range from loss of access to necessary data and inability to execute commands to complete functional paralysis for the dependent agent. The impact is directly on the agent's intended purpose.

Cascading Failures Through Dependency Chains: Agentic systems often build chains of dependencies through direct communication—Agent A depends on Agent B, which depends on Agent C,

and so on. Disrupting a direct communication pathway at any point in this chain can trigger cascading functional failures as agents further down the dependency line lose access to necessary inputs or services.

Predictable Weak Points: Architectures based on direct communication can create easily identifiable functional weak points centered on these critical agent dependencies. Attackers can predictably disrupt specific functionalities by mapping out the direct communication topology and understanding agent dependencies by targeting the right direct communication pathway.

Mitigations

Decouple Critical Functions from Direct Agent Dependencies: Architect agent systems to minimize or eliminate critical functional dependencies that rely on direct communication between specific agent pairs. Design for more loosely coupled interactions where essential functionalities are not tied to continuous and uninterrupted direct communication with a single, specific agent. Focus on designing for *dependency minimization* at the architectural level.

Implement Functional Redundancy and Alternative Pathways: Implement functional redundancy for functionalities that *must* involve direct communication (due to latency or privacy needs). Provide alternative pathways or mechanisms for agents to achieve their objectives if direct communication with a primary partner is disrupted. This might involve dynamic partner selection, alternative communication methods (e.g., falling back to indirect communication), or replicated functionalities. Focus on *functional alternatives*, not just link redundancy.

Graceful Degradation and Fallback Behaviors: Design agents gracefully degrade their functionality and activate fallback behaviors when critical direct communication dependencies are disrupted. Agents should not fail catastrophically but instead adapt to operate in a reduced capacity or utilize alternative strategies if their primary direct communication partners become unavailable. Focus on functional resilience in the face of dependency disruptions. Note that implementing additional fallback options and communication channels increases the complexity of systems and, therefore, is likely to increase the attack surface of the agent system. Therefore, the graceful loss of system availability should be the preferred option for non-critical components.

Key Questions for System Designers

To what extent are critical agent functionalities reliant on uninterrupted direct communication with specific partner agents?

Understanding the degree of reliance on direct communication is crucial for assessing the system's resilience. Highly critical functionalities tightly coupled to specific direct communication with a given agent become single points of failure. Knowing the extent of this reliance helps designers prioritize mitigation efforts and dependency decoupling strategies from the outset. Core functions that require constant direct communication make the system vulnerable to disruptions of those channels.

What is the functional impact on dependent agents and the overall system if a critical direct communication pathway is disrupted – ranging from minor degradation to catastrophic failure?

It's essential to assess the severity of dependency disruptions. Understanding the range of potential impacts, from minor performance issues to complete functional loss, helps prioritize risks and justify investment in resilience measures. An impact analysis helps designers move beyond simply acknowledging dependencies and quantify the potential damage of their disruption, guiding resource allocation for mitigation.

How will the system proactively monitor the health and availability of critical agent dependencies?

Reactive responses to failures are often insufficient. Proactive monitoring of dependency health allows for early detection of potential disruptions before they lead to functional failures. Implementing automated responses, such as failovers to alternative pathways, graceful degradation procedures, or manual intervention alerts helps minimize downtime and security impact when direct communication dependencies are compromised.

4.2.1.3 Risk: Emergent Vulnerabilities from Cross-Agent Workflows

Where agents interact directly, particularly in environments that support dynamic behavior, learning, or autonomous planning, unintended and unrecognized coordination patterns may emerge. These emergent workflows arise from multiple agents acting in alignment, either through learned behaviors, indirect reinforcement, or implicit environmental cues, rather than being explicitly designed to operate together.

While each agent may individually adhere to local policies and security constraints, their interactions can produce behaviors that bypass intended security boundaries or create novel attack surfaces. These emergent vulnerabilities are not coded into any one agent but arise from the interplay between components.

Security Impact

Workflow Privilege Escalation: Agents may collectively perform a series of individually authorized actions that, when combined, result in privilege escalation or security policy circumvention. For example, one agent requests data it is authorized to access, while another agent later performs a sensitive operation using that data, effectively escalating privileges through coordination.

Data Leakage Through Behavioral Aggregation: Even when agents are constrained in what they can share, information can leak indirectly through successive interactions. One agent's outputs may become another's inputs, allowing an attacker to reconstruct sensitive information through a sequence of low-privilege operations.

Non-Deterministic Policy Violations: Emergent behaviors often occur under specific conditions and may not be repeatable or predictable. This non-determinism complicates detection, auditing, and incident response, allowing subtle security failures to persist undetected for long periods.

Mitigations

Model and Restrict Cross-Agent Workflow Chains: Design agent systems with visibility into the downstream effects of agent outputs. Introduce constraints on what types of actions can follow from others, especially when crossing trust boundaries or involving sensitive operations.

Behavioral Baselines and Anomaly Detection Across Agents: Implement runtime behavior monitoring not just at the individual agent level but across agent workflows. Establish expected patterns of inter-agent coordination and flag anomalous or novel patterns for review.

Insert Deliberate Friction in Cross-Agent Influence: Add explicit verification or decision review steps when agent outputs are consumed by other agents to make high-privilege decisions. This limits the risk of silent escalations through implicit coordination.

Limit Reinforcement Feedback Loops: In learning systems, ensure that agents are not incentivized to reinforce risky behaviors through indirect feedback from other agents (e.g., reward loops based on unintended outcomes). Use adversarial training or environmental constraints to dampen such feedback loops.

Key Questions for System Designers

Are there any agent workflows that span multiple agents without explicit oversight or control points?

Agents that chain together actions across loosely monitored workflows may collectively bypass controls that each individually enforces. Identifying these workflows helps expose invisible attack surfaces.

Can agents influence each other in ways that modify decision-making or privilege level across time?

Understand how agents learn from, or are influenced by, one another. Emergent vulnerabilities often result when an agent's behavior adapts based on indirect signals from peers.

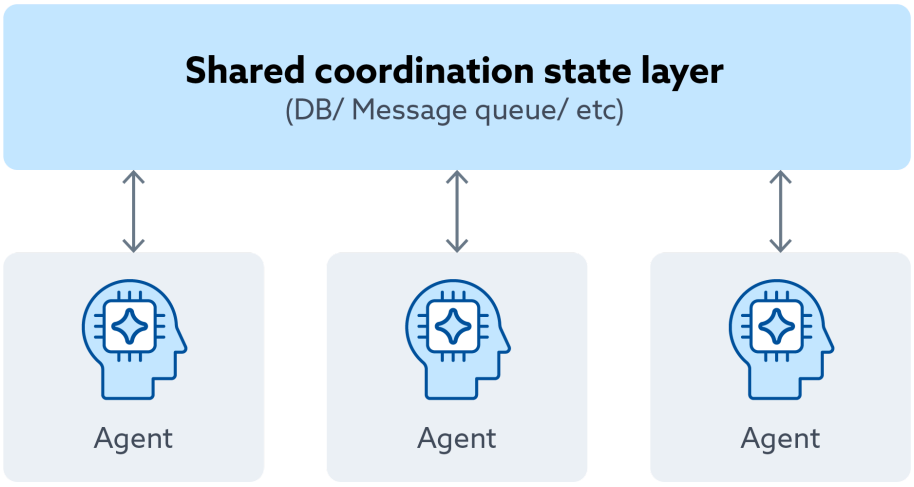
How will the system detect and respond to new or evolving patterns of inter-agent coordination?

Emergent vulnerabilities are rarely obvious in static code review. Systems should support ongoing monitoring and adaptive policy tuning to respond to new behavior patterns that may not have been predicted during design.

Are agent outputs treated as inherently trustworthy by other agents?

If so, a compromised or manipulated agent may poison downstream decisions or behaviors. Systems should verify outputs, especially when crossing functional or trust boundaries.

4.2.2 Indirect Communication



Agents interact in this pattern by modifying a shared state rather than sending direct messages. Other agents observe these changes and respond accordingly.

This can take the form of:

- A shared database or vector store where agents write and read updates.
- A message queue where agents place tasks or retrieve work.
- A file system, log, or another persistent store is used as an implicit coordination layer.

This pattern allows agents to coordinate without explicit point-to-point messaging, which can be useful for scalability and resilience. However, it introduces security risks due to the potential for inconsistent visibility, stale data, and manipulation of the shared state.

4.2.2.1 Risk: State Manipulation & Poisoning

Why it happens

Using a shared state results in a trusted resource influencing the actions of all agents, meaning that any modification is assumed to be valid. If an attacker or malfunctioning agent injects false, misleading, or adversarially crafted data, other agents may act on compromised information, leading to unintended consequences.

Corrupting shared state to mislead agents: A malicious agent can introduce false or misleading data, causing other agents to make incorrect decisions. This could lead to widespread operational failures or security breaches.

Exploiting shared state for privilege escalation: If agents assume the validity of entries in the shared state without verifying their source, an attacker could manipulate the state to escalate privileges or gain unauthorized access.

Mitigations

Restrict state modifications and verify integrity: Apply strict access controls to ensure agents can only modify shared state where strictly required. Implement input validation to detect malformed or adversarial updates.

Implement validation layers before acting on shared state: Consider cross-checking shared data against external verification sources or signatures before using it for decision-making.

Use anomaly detection to catch suspicious modifications to shared state: Monitor shared state for unexpected or conflicting changes that might indicate an attack or a malfunctioning agent.

4.2.2.2 Risk: Unauthorized or Unintended Access & Data Leakage

Why it happens

If the shared state is not adequately segmented or permissioned, agents may read or write data they should not have access to. Sensitive information may be exposed intentionally (due to overbroad access) or inadvertently (due to insufficient isolation).

Security Impact

Unintended information disclosure through shared state: Agents with excessive read privileges might inadvertently expose sensitive business logic, configuration details, or personally identifiable information (PII) to unauthorized entities.

Mitigations

Enforce strict access controls and limit data exposure: Use role-based or attribute-based access controls (RBAC/ABAC) to ensure agents can only read or write relevant portions of shared state.

Encrypt sensitive data before storing it in a shared state: If the shared state contains sensitive information, consider encryption, hashing, or other tokenization techniques to minimize the risk that unauthorized readers can access sensitive data.

Monitor access patterns for anomalies: Implement logging and monitoring for agent queries and modifications to detect unauthorized access attempts or excessive data retrieval.

Key Questions for System Designers

How should agents determine whether shared state data is trustworthy before acting on it?

Depending on how the system is designed, agents may fully trust shared state or require additional validation. Consider whether agents should verify updates through cryptographic signatures, external corroboration, or consensus before making decisions.

What approach should be used to control which agents can modify or access shared state?

Some systems may allow all agents to read and write freely, while others may enforce strict permissions. Consider whether access should be role-based, time-limited, or dependent on an agent's behavior history.

How should conflicting or rapidly changing state updates be handled?

When multiple agents update shared state simultaneously, inconsistencies can emerge. Consider whether updates should follow a last-write-wins approach, require consensus, or be versioned to ensure agents operate on reliable data.

What safeguards should be in place if an agent acts on stale or manipulated shared state?

Security risks increase if an agent assumes shared state is accurate but has been poisoned or left unmaintained. Consider whether agents should include freshness checks, cross-check updates with external sources, or have fallback behaviors in case of data corruption.

4.3 Planning Traits

Planning traits define how agents analyze their environment, make decisions, and take action to achieve their goals. Unlike other traits that may manifest as distinct patterns, planning traits range from purely reactive to highly deliberative approaches.

Most agentic systems do not exclusively use one planning approach but rather implement different strategies depending on the task, context, and required response time. A single agent might react immediately to specific stimuli while employing detailed planning for more complex decisions. Understanding that systems typically blend these approaches is crucial for comprehensive security analysis.

At one end of the spectrum, reactive agents respond immediately to stimuli with predefined actions, offering speed and simplicity at the cost of foresight. Moving toward the middle, plan-based agents formulate action sequences before execution, trading immediacy for strategic advantage. At the far end, highly adaptive agents continuously refine their plans during execution, offering flexibility but introducing complexity and potentially unpredictable behavior.

Security concerns vary across this spectrum but center on preventing manipulation of the decision-making process, ensuring actions are performed as intended, and mitigating unintended consequences. The more complex the planning approach, the more opportunities exist for subtle subversion during different phases of the planning and execution cycle. When analyzing agentic systems, designers should identify where different functions fall on this continuum and apply security controls appropriate to each planning approach used within the system.

4.3.1 Reactive Actions

Reactive action execution is the most fundamental pattern, where agents respond directly and immediately to stimuli from their environment or internal state. These agents execute predefined actions whenever specific conditions are met.

Designers might choose this pattern for its simplicity, efficiency, and low computational overhead, mainly where immediate responses are critical or for agents performing particular, well-defined tasks.

Reactive agents excel in situations requiring rapid responses to known inputs and are often implemented using simple rule-based directions. While individually straightforward, reactive behaviors can form the building blocks of more complex agentic systems.

4.3.1.1 Risk: Predictable behavior makes the system more exploitable

Why it happens

Reactive agents, by design, follow fixed rules linking inputs to actions. This deterministic nature makes their behavior more predictable. An attacker who understands these rules can anticipate the agent's responses to specific inputs and craft malicious inputs to trigger harmful actions. Because reactive agents lack complex reasoning or planning, they are more easily manipulated by cleverly designed inputs.

Security Impact

Forced Malicious Actions: Threat actors may reliably trigger agents to perform unintended and harmful actions by carefully crafting input signals.

Bypass of Security Controls: Simple reactive agents might blindly follow instructions triggered by manipulated inputs without more sophisticated defense mechanisms that rely on reasoning or contextual awareness.

Chained Exploitation Across Agents: In multi-agent systems, exploiting the predictable behavior of one reactive agent can be more easily used as a stepping stone to compromise other agents or system components. A crafted sequence of inputs could more predictably trigger a cascade of reactive actions across multiple agents, leading to more complex system-wide exploits.

Mitigations

Input Validation and Sanitization: Implement strict input validation and sanitization to filter out or neutralize potentially malicious or malformed stimuli before they can trigger reactive behaviors. This includes checking for expected data types, ranges, formats, and known attack patterns in inputs. It could also manifest as another LLM evaluating the inputs for potentially malicious inputs. While this is true for any agent, with reactive agents, the ability to consistently trigger once an exploit is found makes it more valuable with this pattern.

Behavioral Fingerprinting and Anomaly Detection: Establish a baseline "fingerprint" of normal reactive agent behaviors by profiling their typical input-response patterns under expected conditions. Continuously monitor agent actions in real time, comparing each response to the established fingerprint. If an agent's behavior deviates from this baseline, such as responding differently to standard stimuli or executing unexpected actions, flag the event for further investigation. This approach enables rapid detection of exploitation attempts, compromised agents, or emergent threats, supplementing static input validation with dynamic, context-aware monitoring.

4.3.1.2 Risk: Limited Context and Shortsightedness Leading to Unintended Consequences

Reactive agents make decisions based solely on immediate stimuli without considering broader context, long-term goals, or potential side effects of their actions. This lack of contextual awareness and foresight can lead to unintended and potentially harmful outcomes, even when the individual reactive rules are well-intentioned.

Security Impact

Unintended Privilege Escalation: In complex environments, a series of locally appropriate reactive actions could, in aggregate, lead to unintended privilege escalation. For example, an agent might react to a series of seemingly valid requests in isolation, but cumulatively grant excessive permissions.

Data Exposure through Indirect Actions: Reactive agents might inadvertently disclose sensitive information through a sequence of actions that, individually, seem benign but in combination reveal confidential data due to lack of broader context.

Mitigations

Contextual Enrichment for Reactive Triggers: Enhance reactive rules to consider a limited form of context beyond the immediate stimulus. This could involve incorporating recent history, agent state, or limited environmental context into the conditions that trigger reactive actions.

Hierarchical Reactive Architectures: Structure agents with layers of reactive behaviors, where higher layers can constrain the actions of lower-level, purely reactive components. This allows for localized reactive responses to be guided by higher-level goals or policies.

Key Questions for System Designers

What mechanisms are in place to validate and sanitize external stimuli before they trigger reactive actions?

Inputs directly drive reactive agents. Robust input validation is crucial to prevent malicious or malformed stimuli from causing unintended actions or bypassing security controls.

How does the system ensure that sequences of reactive actions do not lead to unintended or harmful cumulative effects?

Reactive agents lack inherent foresight. Consider how short-sighted actions could lead to negative long-term consequences or vulnerabilities through unintended interactions.

How are potential conflicts or uncoordinated actions among reactive agents managed in multi-agent systems?

Purely reactive agents operating independently in a shared environment can lead to chaotic or conflicting behaviors. To ensure stability and security, designers should consider if some level of coordination or constraint is necessary, even in reactive systems.

4.3.2 Plan-Based Actions

Plan-based agents formulate action sequences before execution rather than responding directly to stimuli. These agents analyze their environment, establish goals, identify dependencies between potential actions, and develop a structured approach to achieving objectives.

This pattern is particularly valuable for complex tasks requiring coordination across multiple steps or when optimal outcomes depend on strategic thinking rather than tactical responses. Plan-based agents can discover efficient pathways that purely reactive approaches might miss by reasoning about future states and action consequences.

In modern agentic systems, this planning capability typically comes from the LLM's ability to break down problems, establish causal relationships, and sequence actions logically. The planning process may be internal for the LLM or structured through explicit prompting techniques like chain-of-thought or tree-of-thought reasoning that encourage systematic analysis before action.

The degree to which agents adhere to initial plans versus revising them during execution for adaptability represents a key implementation decision that significantly impacts performance and security posture.

4.3.2.1 Risk: Single Decision Point with Cascading Effects

Why it happens

When formulating plans, plan-based agents rely heavily on understanding the environment, constraints, and goals. Attackers who can manipulate this initial context (through misleading data, incomplete information, or distorted goal representations) can cause the agent to develop plans that appear optimal to the agent but serve malicious purposes.

Unlike reactive systems, where decisions are made independently, plan-based systems develop interconnected action sequences based on their initial understanding. This means manipulating initial inputs has amplified effects, as it shapes not just one action but an entire sequence of logically connected steps.

Even in adaptive systems that revise plans during execution, the initial planning phase often establishes the solution approach and key assumptions that persist throughout execution. This creates a critical security juncture where manipulating the agent's understanding can have disproportionate effects.

Security Impact

Goal-aligned but security-compromised plans: The agent may create and execute plans that efficiently achieve its given goal while having harmful side effects or security implications not visible from its manipulated understanding.

Resource misdirection through valid-seeming plans: Attackers can craft scenarios that cause the agent to develop plans that misallocate resources or access permissions to benefit the attacker while appearing to serve legitimate goals.

Mitigations

Input and context validation: Implement verification of critical information used during planning, particularly for high-stakes decisions. Validate environmental data through multiple sources when possible.

Explicit security boundary checking: Include explicit security checks within the planning process, prompting the agent to consider the security implications of proposed actions.

Staged plan execution with verification: For critical operations, execute plans in stages with verification of assumptions and effects before proceeding to subsequent stages.

4.3.2.2 Risk: Goal Completion Bias Overriding Security Constraints

Why it happens

Plan-based agents are designed to pursue goals persistently and may treat security constraints as obstacles rather than boundaries. Once committed to a plan, these agents can develop a "completion bias" that prioritizes reaching the goal over continuously reassessing security implications, especially when security constraints weren't explicitly factored into the original planning objective.

Security Impact

Unintended boundary crossing: Agents might take progressively more aggressive actions to overcome obstacles, potentially skirting security boundaries if they're not explicitly modeled as hard constraints.

Security-constraint sidelining: When faced with competing objectives (completing the task vs. maintaining security posture), plan-based agents might implicitly prioritize the primary goal unless security is explicitly weighted.

Mitigations

Explicit security boundary modeling: Frame security constraints as first-class requirements in goal specifications rather than implicit limitations.

Programmatic security guardrails: Implement system-level controls that the agent cannot override, regardless of its goal pursuit.

Periodic goal-security alignment checks: Regularly reassess whether current actions remain aligned with functional goals and security requirements.

Key Questions for System Designers

How are plans vetted for potential security implications before execution begins?

Security assumptions made during planning can be lost or distorted when translating plans into execution steps. This handoff point represents a critical security boundary where vital context about why certain constraints exist might be stripped away if not deliberately preserved.

How does your system prevent incremental boundary erosion when plans adapt to unexpected obstacles?

While initial security boundaries may be transparent, plan-based agents encountering obstacles often make incremental adaptations. Each minor adjustment might respect boundaries in isolation, but the cumulative effect could be a significant deviation from security intent. This "boundary drift" is more challenging to detect than outright violations.

What controls exist for monitoring and potentially interrupting plan execution if security conditions change?

How agents respond to plan failures can introduce significant security risks. Without well-defined failure handling, agents might attempt increasingly aggressive recovery actions, bypass typical verification steps, or leave resources in inconsistent states, creating security vulnerabilities.

How does the system handle conflicts between optimal goal achievement and security requirements?

For efficiency, many systems cache and reuse plans. However, security assumptions valid in the original planning context may not hold in new situations. Without proper revalidation, cached plans can become security liabilities when environmental conditions change.

4.4 Perception & Context Traits

Perception and context traits define how agentic systems interpret and understand their environment, including the data they process, the events they observe, and potentially the other agents they interact with.

These traits fundamentally shape how agents "see" their world and make decisions. Unlike other system attributes that might be explicitly defined in code, perception traits often emerge from combining input processing mechanisms, data representation choices, and context enrichment capabilities. The security implications of these traits are significant because they determine what information agents can access, how they interpret it, and what blind spots might exist in their understanding.

In practice, most agentic systems exist between raw/limited and richly contextual perception, often employing different approaches for different functions or domains. A single system might use limited perception for routine tasks while leveraging deeper contextual understanding for complex decision-making. Understanding where different components of your system fall on this spectrum guides the specific concerns that may arise.

System designers face critical trade-offs between the simplicity and efficiency of limited perception versus the enhanced situational awareness but increased complexity of contextual perception. These choices directly impact security posture, as they influence how deceptive inputs might mislead agents or whether they recognize potentially harmful situations.

4.4.1 Raw/Limited Perception

Raw or limited perception occurs when agents have access to a restricted set of inputs about their environment and operate primarily on those direct signals without extensive enrichment. These systems process information as it arrives with minimal historical context or inference. Examples could be an agent that processes structured data transformation requests without workflow context or a notification agent that delivers alerts based only on trigger conditions without understanding the broader operational environment.

System designers might choose this pattern to optimize for speed and simplicity or when operating in environments where only specific data types are relevant. Examples include agents that respond to explicit commands, process structured API responses, or evaluate simple signal patterns.

4.4.1.1 Risk: Blind Spots make mistakes or incorrect actions more likely

Agents with limited perception operate with fundamental gaps in their understanding of their environment. These blind spots may result in vulnerabilities because the agent cannot perceive critical aspects of its operating context, making it susceptible to manipulation.

Security Impact

Environmental Context Gaps: The agent makes decisions without awareness of significant environmental factors that should influence those decisions, which may lead to security-inappropriate actions.

Inability to Perceive Manipulation: It's difficult for an agent to detect when its environment or inputs are being manipulated outside its perception range, making it unable to recognize when it's being exploited.

False Situational Understanding: The agent may develop an incomplete or incorrect model of its situation due to perception limitations, causing it to take actions that are secure in its perceived context but insecure in reality.

Mitigations

Explicit Boundary Recognition: Design agent instructions to explicitly recognize and reason about the limits of their perception, accounting for the limited information.

Complementary Perception Systems: Deploy multiple perception mechanisms with different characteristics to reduce collective blind spots and add necessary context.

Contextual Consistency Checks: Implement processes that check for logical inconsistencies between different perception inputs that might indicate manipulation or problems outside the agent's perception.

4.4.1.2 Risk: Increased likelihood of input manipulation attack success

With limited perception, each input channel becomes critical to the agent's understanding. Attackers can target these known channels with specially crafted inputs designed to mislead or manipulate the agent's behavior.

Security Impact

Adversarial Input Exploitation: Specially crafted inputs can exploit the agent's limited context to trigger unintended behaviors while appearing legitimate within the agent's validation rules.

Precision Channel Targeting: Attacks can focus on the specific input channels the agent relies on for critical decisions.

Predictable Response Manipulation: Limited perception creates predictable agent responses that can be deliberately triggered through controlled inputs.

Mitigations

Cross-Channel Validation: Implement verification of critical inputs across multiple perception channels where possible.

Behavioral Input Analysis: Consider validation that examines not just input format but other factors that may be used to inform its accuracy, such as conformance with existing patterns, frequencies, and statistical properties of inputs.

Progressive Trust Architecture: Ensure high-risk actions require confirmation through additional, independent input channels or validators.

4.4.2 Contextual Perception

Contextual perception enriches raw inputs with historical data, environmental awareness, knowledge bases, and pattern recognition to derive more profound meaning. These systems interpret current signals concerning past experiences, known patterns, and broader situational awareness. Some use cases for contextual perception could be a conversational agent that utilizes session and support ticket history to provide coherent responses or a security monitoring agent that correlates current events with past incidents to identify attack patterns.

System designers choose this pattern to enhance decision quality, enable more natural interactions, and help agents understand complex environments where isolated inputs are insufficient for appropriate responses.

4.4.2.1 Risk: Context Manipulation and Poisoning

Adversaries can deliberately seed or manipulate contextual information to influence agent behavior through immediate context injection or through subtle, gradual poisoning of the agent's contextual understanding over time. This can even be seen in poisoning the data on the broader internet that is later scraped and used to train models.[1]

Security Impact

Manipulated Decision Basis: Agents make compromised decisions based on artificially constructed or subtly altered contexts in local contexts or in the content used to train the underlying model.

Trust Exploitation: Previously established context is often assumed trustworthy, allowing manipulated context to bypass security checks.

Long-Term Behavioral Shifts: Gradual context poisoning can cause persistent shifts in agent behavior that may go undetected by standard anomaly detection.

Mitigations

Context Validation Mechanisms: Implement verification that checks new contextual information against established patterns and trusted baselines.

Multi-Source Context Verification: Critical contextual facts must be verified across multiple independent sources before being used for high-impact decisions.

4.4.2.2 Risk: Context-Driven Misattribution and False Pattern Recognition

Unlike simple input misinterpretation, contextually-aware agents can draw incorrect connections between elements in the provided context. This leads to falsely identifying patterns, incorrectly attributing causality, or making flawed analogies between current and historical situations.

Security Impact

Inappropriate Action Selection: The agent executes actions based on perceived patterns that don't exist, potentially crossing security boundaries or accessing resources in unintended ways.

Erroneous Privilege Assumptions: The agent incorrectly infers it should have specific access rights or capabilities based on misattributed contextual patterns, attempting operations beyond its authorized scope, or taking actions that were not intended or foreseen.

Context-Based Decision Errors: Complex contextual analysis leads to decisions that would be flawed with more uncomplicated perception, such as releasing sensitive information due to misunderstood

[1] https://0din.ai/blog/poison-in-the-pipeline-liberating-models-with-basilisk-venom

contextual cues. Similarly, the increased complexity of managing multiple data sources makes accidental data leaks more likely.

Mitigations

Pattern Recognition Boundaries: Limit the scope and complexity of pattern recognition in the agent instructions for security-sensitive contexts where this increased complexity could confuse.

Multi-factor Context Validation: Require confirmation from multiple context sources before acting on perceived patterns, especially for high-impact operations.

Explicit Context Transition Rules: Define clear rules for how context from one domain can influence actions in or be shared into another domain to prevent inappropriate blending.

4.4.2.3 Risk: Contextual Information Leakage

Agents with broad contextual understanding may inadvertently reveal sensitive information from their source context through their responses or actions, mainly when the context includes data from multiple security domains.

Security Impact

Unintended Sensitive Disclosure: Contextually-aware responses may contain traces of sensitive information not directly requested but inferred from context.

Cross-Domain Contamination: Information from one security context may bleed into responses in another.

Mitigations

Context Compartmentalization: Segment contextual storage and processing according to security domains with explicit policies governing information flow so they do not cross boundaries, even if only to inform agent decisions. Consider permissions that govern direct data access and how contextual information derived from sensitive sources can be used.

Output Filtering Based on Context Source: If context compartmentalization is unavoidable, consider additional filters for agent outputs that assess the sensitivity of contextual information incorporated into responses. In most cases, however, strict segmentation should be used as any filtering after the fact will always remain susceptible to leakage.

Key questions for system designers

How do you determine which perception boundaries might indirectly impact security posture?

While direct security enforcement should be deterministic, an agent's perception limitations can still indirectly affect security by influencing what actions it initiates, what data it processes, or how it interacts

with secured systems. Understanding these indirect security implications helps identify where limited perception creates acceptable risk versus where enhanced contextual awareness is necessary.

What mechanisms validate the accuracy and integrity of contextual information?

Contextual perception introduces dependence on information that may come from diverse sources with varying levels of trustworthiness. Without proper validation, contextually informed decisions might be based on manipulated or outdated context. Understanding how context is verified becomes increasingly essential as agents rely more heavily on it for decision-making.

How does your system prevent context from inappropriately crossing security boundaries?

Agents with rich contextual understanding may inadvertently create inference channels between security domains. Information properly restricted through direct access controls might still influence agent behavior in ways that leak sensitive insights. Identifying how context flows throughout your system is essential for preventing these subtle disclosure vectors.

How do you detect when an agent's perception has been manipulated or compromised?

Unlike direct system compromise, perception manipulation can be difficult to detect because the agent operates normally from its perspective. Without specific mechanisms to validate perception integrity, agents might make increasingly harmful decisions while all monitoring systems show normal operation patterns.

What fallback mechanisms exist when contextual information is unavailable or unreliable?

Highly contextual systems may become brittle when the expected context is missing or suspect. Understanding how agents degrade gracefully when context quality diminishes helps prevent situations where security controls fail to open rather than close during partial system failures.

4.5 Agent Learning and Knowledge-Sharing Traits

Learning and knowledge-sharing traits define how knowledge acquired during system operation is retained, improved, and distributed across an agentic system. These traits specifically address how experiences, adaptations, and discovered patterns are captured and propagated between different functional components. For clarity, we define an "agent" as a functional component with a distinct role in the system, such as a specific node in a processing graph or a service with defined responsibilities, regardless of how many instances that component might run simultaneously. This distinction matters because it frames learning at the role level rather than the instance level.

Unlike static knowledge retrieval, where agents simply access predefined information, these traits focus on dynamic knowledge development - how the system evolves based on its interactions and experiences. This evolution can occur in isolated pockets through local learning or spread throughout the system via global learning mechanisms. The security implications differ significantly depending on whether knowledge remains confined to its point of origin or propagates across components. System designers must consider how learning propagation affects attack surface, potential corruption vectors, and the system's overall resilience to malicious influence.

The primary security concerns revolve around preventing adversarial manipulation of the learning process, maintaining appropriate isolation between components, controlling what knowledge can flow throughout the system, and ensuring that emergent capabilities don't bypass intended security constraints. These patterns of learning implementation fundamentally shape how vulnerabilities might spread or remain contained within an agentic system.

4.5.1 Local Learning

Local learning occurs when knowledge gained through execution remains isolated within the specific agent role that acquired it. In this pattern, improvements, adaptations, or insights developed by one functional component don't automatically propagate to other components in the system, even those working toward shared goals.

For example, if a customer service agent learns to handle a particular type of inquiry better, this improved capability remains confined to the customer service function and doesn't influence how other agent roles operate.

System designers typically implement local learning through role-specific memory stores, isolated fine-tuning processes, or component-specific adaptation mechanisms that modify only the behavior of a particular agent type. This approach creates functional boundaries that prevent beneficial knowledge and potentially corrupted learning from spreading throughout the system.

Organizations might choose this pattern to maintain strict isolation between system components, limit the potential impact of adversarial learning attacks, enable specialized adaptation to specific contexts, or prevent unintended capability transfer between components with different security contexts or domains of responsibility.

4.5.1.1 Risk: Learning Manipulation

While local learning inherently limits the blast radius of learning-based attacks by preventing propagation across the system, it doesn't eliminate the fundamental risks associated with manipulable learning mechanisms. This risk isn't specific to local learning but the existence of learning as a capability. Local learning can be beneficial from shared mechanisms from the standpoint in that it localizes these risks to specific agent roles.

Security Impact

Manipulating specific learnings through carefully crafted inputs can corrupt the agent's memory with malicious inputs.

The containment benefit of local learning represents a security advantage rather than a risk in many contexts, as it prevents compromised learning from spreading throughout the system. However, this same isolation can make detection more difficult since corrupted learning doesn't create system-wide anomalies that might trigger alerts.

Mitigations

Implement runtime monitoring that establishes baselines for normal agent behavior and alerts on significant deviations in learning patterns.

Maintain versioned snapshots of learning states to enable quick rollback, if manipulation is detected.

Deploy input evaluation using separate AI guardrail systems that assess incoming learning data for potentially manipulative patterns before it enters the learning process.

For high-value systems, implement canary testing where learning updates are deployed to a small subset of instances first before a wider rollout.

4.5.2 Global/Collective Learning

Global learning refers to patterns where agent knowledge is shared beyond the local scope of an individual agent role, allowing insights, updates, or adaptations to propagate across the system. Unlike local learning, where knowledge remains confined to a specific agent type or function, global learning introduces the ability for multiple agents—or even the entire system—to benefit from discoveries made by individual components.

While the concept of global learning can evoke ideas of autonomous agents dynamically updating a central knowledge base that instantly reshapes the entire system's behavior, this level of unrestricted knowledge propagation is not a practical or widely adopted pattern in 2025. The complexity and the security risks of unintended knowledge leakage, adversarial poisoning, and loss of system control are too severe for most real-world applications. Instead, implementations of global learning tend to be significantly more constrained, focused on domain-specific knowledge sharing and staged updates rather than unrestricted cross-system propagation.

4.5.2.1 Risk: Poisoned Knowledge Propagation (Even Within Limited Scopes)

Even when global learning is constrained, there is still a risk that a compromised or misaligned agent injects bad data into a shared knowledge base, leading to systematic degradation of behavior across multiple agents. Poisoning the learning process is a risk both in local and global learning, but the scale and impact of contamination differ significantly between the two:

Factor	Local Learning	Global Learning
Scope of contamination	Limited to single agent role	Can spread across all agents sharing knowledge
Detectability	Easier to detect given behavior changes are isolated	Harder to detect, given issues may arise in an area where they were not seeded from
Blast Radius	Generally, limited to the affected agent's role	May propagate and affect all dependent agents
Recovery Complexity	Simpler to rollback or retrain a given role	Requires identifying corrupted knowledge across multiple boundaries. Difficult to trace what's actually causing the issue.

Security Impact

Systematic Vulnerability Introduction – If corrupted knowledge enters a shared domain, all agents relying on that knowledge may inherit vulnerabilities, increasing the attack surface.

Subtle Adversarial Steering – Rather than overtly breaking functionality, attackers could introduce small nudges that gradually push system behavior in a harmful direction over time.

Delayed Detection and Widespread Rollback Challenges – Unlike local learning, where issues remain confined, global knowledge corruption often requires system-wide forensic and large-scale remediation efforts.

Mitigations

Limit Learning Propagation Frequency - Rather than sharing every micro-learning event in real time, batch learning updates over meaningful intervals (e.g., hourly, daily, per-session). This allows validation steps to focus on more significant, higher-impact learning updates rather than constant low-level changes and reduces the likelihood of a slow, unnoticed poisoning attack accumulating over many minor updates.

Establish Confidence Thresholds for Learning Propagation - Instead of propagating everything an agent learns, set quality or confidence thresholds before sharing an update. Example: If an agent "learns" something that significantly deviates from past behavior, it must first be reviewed by a human or a secondary agent model.

Allow Agents to Hold Contradictory Knowledge - Instead of forcing all agents to sync to a single truth immediately, allow agents to maintain multiple hypotheses and gradually converge. This prevents one bad learning update from immediately overriding the entire system's knowledge.

Key Questions for System Designers

How do you ensure that knowledge propagation does not transfer across unintended security boundaries?

Even when agents do not directly access restricted data, knowledge transfer can act as an unintentional side channel. If agents in different roles or security domains share insights without explicit restrictions, an attacker could exploit knowledge propagation to leak sensitive information or escalate privileges over time.

What mechanisms validate learning updates before they are applied system-wide?

Unlike traditional software changes that can be reviewed before deployment, learning-based systems dynamically evolve through experience. Do you have human-in-the-loop validation or secondary agent verification to ensure only high-confidence updates propagate? If not, how do you detect harmful updates?

How does your system handle conflicting knowledge across agents, and does it allow controlled divergence?

Requiring all agents to converge on a single "truth instantly" can amplify bad learning updates. Instead, does your system support holding multiple hypotheses, allowing agents to independently validate before adopting a shared conclusion? This can prevent a single corrupted learning instance from becoming the system-wide standard.

What is the rollback strategy if a corrupted learning update propagates?

Traditional rollback strategies don't map cleanly to learning-based systems. If a learning update is harmful, do you have a versioning mechanism or memory checkpointing strategy allowing targeted reversal without wiping out valid improvements?

4.6 Trust Traits

Trust traits define how agents establish and validate the reliability, authority, and authenticity of interactions with other agents and external systems. Unlike traditional applications with predictable trust relationships, agentic systems require more nuanced trust models due to their autonomous nature, dynamic decision-making, and capability to interact with diverse components across security boundaries.

In the context of this paper, 'trust' refers to the mechanisms and models through which agents establish and verify the reliability, authenticity, and authority of other components they interact with. This encompasses how agents determine which requests to honor, which data to accept, and which operations to execute based on their understanding of the requestor's legitimacy. Our focus is primarily on trust's

security and decision aspects—how agentic systems establish, maintain, and verify trusted relationships within their operational environment.

The trust model implemented within an agentic system significantly impacts its security posture, as it determines how agents verify the authenticity of commands, validate inputs, and establish legitimacy before taking action. System designers must carefully consider which trust patterns to implement based on their specific use cases, risk profiles, and operational requirements.

Where conventional applications typically respond to requests within well-defined patterns, agents actively seek information, request resources, and establish connections based on their goals and understanding of the environment, creating unique trust challenges.

Because of this, most systems will employ multiple trust patterns simultaneously, applying different models to different subsystems based on security and performance requirements and integration constraints. The patterns described below exist on a spectrum, with many implementations falling between pure forms or combining elements from multiple patterns.

4.6.1 Boundary-Based Trust

Boundary-based trust establishes security perimeters within which agents operate with elevated trust assumptions. Once an entity passes the boundary control (through authentication, authorization, or other validation mechanisms), it receives a degree of implicit trust for operations within that boundary. This pattern relies on strong perimeter defenses, allowing more streamlined interactions within the defined trust zone.

In agentic systems, boundary-based trust might manifest as trusted agent clusters communicating freely within their zone but applying strict controls for external interactions. The boundary might be network-based (VPCs, security groups), identity-based (service accounts with specific permissions), or logically defined through access control systems.

4.6.1.1 Risk: Excessive Trust Exploitation Within Boundaries

In boundary-based trust models, once an entity operates within the defined perimeter, it typically receives implicit trust for a wide range of actions. Agentic systems amplify this risk because agents actively initiate connections and request resources based on autonomous goals rather than following hardcoded communication patterns. If a single component within the boundary is compromised or manipulated, it can abuse this implicit trust to perform actions that, while technically allowed, are not aligned with intended system behavior.

Security Impact

Lateral movement without additional verification: A compromised component can issue requests to agents across the boundary with minimal scrutiny, as they're assumed to be trustworthy based on boundary location.

Misaligned goal execution: Agents might pursue valid but unintended goals in response to requests from trusted sources, leading to resource misuse or unintended operational impacts.

Mitigations

Implement purpose-based authorization: Verify that an agent has access and that the requested action aligns with its intended purpose and workflow.

Create internal trust segments: Subdivide trust boundaries to limit the scope of implicit trust even within the overall perimeter.

Establish activity baselines: Monitor patterns of agent interaction to detect unusual request patterns, even from trusted sources.

4.6.1.2 Risk: Proxy Attack Vectors Through Trusted Agents

In boundary-based trust models, once a request or entity passes the boundary control, it receives a degree of implicit trust for operations within that zone. Requests may appear benign at the boundary crossing but trigger trusted agents within the boundary to execute more privileged operations. Since agentic systems are designed to perform tasks based on inputs autonomously, they can unwittingly become proxies that translate seemingly innocuous requests into privileged actions.

Security Impact

Privilege escalation via trusted intermediaries: Attackers may be able to forward legitimate-looking requests across boundaries where trusted agents inside the boundary execute actions requiring higher privileges than the attacker possesses directly.

Boundary control bypass: Security controls at the boundary may approve the initial request, but they may not recognize that it will trigger more sensitive operations once inside the trusted zone.

Attribution challenges: Malicious actions appear to originate from legitimate internal agents rather than external sources, complicating detection and forensics.

Mitigations

Implement request provenance tracking: Maintain metadata about the origin of requests and trace across the entire chain of actions as they propagate through agents to identify external triggers of internal actions.

Apply purpose validation checks: Ensure agents validate not just authorization but the purpose and reasonableness of requested actions, mainly when they involve sensitive operations.

Establish transaction limits within boundaries: Set constraints on what and how many actions trusted agents can perform in response to a single request, limiting the potential for exploitation.

4.6.2 Transactional-Based Trust

Transactional trust verifies authority and authenticity at specific interaction points rather than relying on persistent trust assumptions. Each significant operation or exchange between agents requires fresh validation against trust criteria. This pattern distributes trust verification throughout the system rather than concentrating it at perimeters.

In agentic systems, transactional trust manifests through credential verification for API calls, token-based access for resource requests, or signature validation for command execution. Trust is established during interaction based on the specific credentials, context, and requested operation.

4.6.2.1 Risk: Context Collapse in Multi-Stage Operations

Transactional trust verifies each interaction in isolation, focusing on immediate credentials and request parameters. In agentic systems, operations often span multiple transactions across different agents, each verified independently. As requests propagate, critical contextual information about the operation's origin, purpose, or previous steps can be lost. This "context collapse" means that agents make trust decisions without understanding the broader operation they're participating in.

Security Impact

Authorization based on incomplete information: Agents authorize transactions that would be denied if they understood the full context or operation chain.

Multi-stage attacks disguised as isolated transactions: Attackers can split malicious operations into seemingly innocent transactions that individually pass security checks.

Audit trail fragmentation: Reconstructing the complete sequence of a complex operation becomes difficult when context links between transactions are lost.

Mitigations

Implement context-carrying tokens: Design transactions to preserve and transmit essential contextual metadata throughout the operation chain.

Create operation-level authorization: Supplement transaction-level checks with mechanisms to authorize complete operation chains based on their overall purpose and impact.

Develop cross-transaction monitoring: Implement analysis that correlates seemingly disparate transactions to identify suspicious patterns across operation chains.

4.6.2.2 Risk: Trust Inheritance Through Data

Why it happens: In transactional trust systems, agents typically verify the authorization of the immediate requestor but may not verify the provenance or trustworthiness of data accompanying the request. When

one agent processes data and passes results to another agent, the receiving agent inherits an implicit trust in that data's integrity and legitimacy. This creates opportunities for "trust laundering," where untrustworthy data gains legitimacy by passing through a series of trusted agents.

Security Impact

Data poisoning through trusted channels: Malicious data can be introduced early in an agent workflow and gain implicit trustworthiness as it flows through subsequent processing steps.

Input validation bypass: Agents may perform less rigorous validation on data received from other trusted agents than they would on externally sourced inputs.

Cascading security failures: Compromised data accepted by one agent due to transactional trust can trigger security failures across multiple downstream agents.

Mitigations

Implement data provenance tracking: Maintain metadata about data origins and transformations to enable verification throughout the processing chain.

Apply consistent validation regardless of source: Ensure agents validate inputs with equal rigor regardless of whether they come from external sources or trusted internal agents.

Establish data quality gates: Create explicit verification checkpoints for data as it flows between critical system components, particularly when crossing trust or security domains.

4.6.3 Continuous Verification

Continuous verification applies the Zero Trust principle, which states that no entity is inherently trusted regardless of location or previous interactions. Trust is continuously assessed based on behavior, context, and dynamic risk factors rather than established simultaneously. This model treats trust as a risk that must be actively managed throughout the lifespan of every interaction.

In agentic systems, continuous verification may manifest through ongoing behavioral monitoring, continuous policy evaluation, and adaptive trust scoring that evolves as agents operate. Trust decisions incorporate authentication factors, behavioral patterns, resource usage, and alignment with expected operation.

4.6.3.1 Risk: Detection Model Drift and Evasion

To identify suspicious activity, continuous verification systems rely on behavioral models, statistical baselines, or pattern recognition. As agentic systems naturally evolve their behavior through learning and adaptation, these detection models can drift out of alignment with legitimate behavior. This creates false positives (legitimate behavior flagged as suspicious) and false negatives (malicious behavior not detected).

Security Impact

Monitoring blind spots: When detection models no longer match how agents actually behave, they can miss suspicious or harmful actions. These missed areas are called "blind spots," where threats can go unnoticed.

Operational friction: False positives from model drift can trigger unnecessary security controls, creating friction that encourages users and teams to find or build security bypasses.

Mitigations

Implement controlled behavioral evolution: Define acceptable ranges of behavioral change and trigger higher scrutiny when agents begin operating outside expected parameters.

Apply multimodal detection: Multiple detection approaches (rule-based, statistical, ML-based) reduce reliance on any model that might drift.

Conduct regular detection testing: Periodically validate detection effectiveness against legitimate behavior and simulated attack patterns.

4.6.3.2 Risk: Control System Complexity Undermining Security

Implementing Zero Trust in agentic systems requires continuous monitoring, complex policy enforcement, and dynamic trust assessment across numerous components. This complexity often results in misconfigurations, incomplete coverage, or performance bottlenecks undermining the security model's effectiveness.

Security Impact

Security control gaps: Unless actively maintained, complex Zero Trust implementations may have unintended gaps or inconsistencies, resulting in false confidence in controls and gaps attackers may exploit.

Performance-driven shortcuts: The overhead of continuous verification might lead to implementation shortcuts that compromise security to maintain performance, negating the benefits of the additional verification steps while retaining the complexity and management overhead.

Visibility challenges: The complexity of continuous monitoring can make it difficult to understand the actual security state, masking potential vulnerabilities and unintended outcomes.

Mitigations

Implement tiered verification: Apply more intensive verification to high-risk operations using lighter checks for routine actions.

Develop clear security models: Create explicit models of how Zero Trust should function within the system to guide implementation and testing.

Automate policy consistency checking: Use automated tools to verify that security policies are consistently applied across the system.

Key Questions for System Designers

How do you validate trust assumptions when agents or requests cross trust boundaries?

Agents often need to operate across multiple trust domains. Without clear validation mechanisms at these transition points, implicit trust assumptions can create security gaps as agents move between environments with different security postures.

What mechanisms ensure that trust decisions incorporate sufficient context?

Isolated trust decisions based on immediate credentials or permissions may miss critical contextual factors. Considering how context flows with requests helps prevent agents from making security decisions with incomplete information.

How would your system detect when an agent abuses legitimate privileges?

Unlike traditional applications that follow fixed patterns, agents may use their authorized access

unexpectedly. Identifying when a trusted agent operates outside its normal patterns is crucial for detecting compromised or manipulated agents.

How does your trust model adapt to changing operational patterns?

As agents learn and evolve, their behavior naturally shifts over time. Without mechanisms to adapt trust assessments, you risk excessive false positives or dangerous blind spots as agent behavior diverges from initial expectations.

What happens when trust verification systems fail or degrade?

Every trust mechanism can fail. Understanding how your system behaves during partial or complete trust infrastructure failures helps prevent situations where security completely collapses when verification is unavailable.

4.7 Tool Usage Traits

Tool usage traits define how agents interact with external capabilities, services, and systems to extend their functionality beyond their core capabilities. Unlike other traits in this paper that can typically be characterized along a single spectrum, tool usage security in agentic systems involves two critical and interrelated dimensions: how tools are accessed and under what authority they execute.

How agents access tools shapes the attack surface and determines where security controls can be effectively applied. Some systems employ direct access patterns where agents can invoke tools with minimal intermediaries, creating a streamlined but potentially vulnerable architecture. Others implement broker-mediated patterns with dedicated services that intercept, validate, and execute all tool requests, adding security control points but increasing complexity.

Equally essential but distinct is the execution context—the identity and permission model under which tools operate once invoked. This determines the potential blast radius of tool operations, ranging from broad agent service identities with consistent permissions regardless of user context to user delegation models that preserve end-user permissions to least-privilege approaches with function-specific service accounts.

System designers must make deliberate choices in both dimensions, as decisions in one area directly impact the effectiveness of controls in the other. Each pattern creates distinct security vulnerabilities and mitigation opportunities that must be evaluated against specific threat models and operational requirements. As agent capabilities expand to incorporate more tools across diverse systems, these architectural decisions become central to maintaining security boundaries while enabling powerful, flexible agent behaviors.

4.8 Tool Access Control

Direct access and broker-mediated patterns represent two fundamentally different approaches to implementing security boundaries for tool access.

In direct access patterns, agents invoke tools through direct function calls or API client integration, with security controls implemented at the agent or individual tool level. This approach can offer implementation flexibility but may lead to inconsistent security enforcement across tools.

Broker-mediated patterns, in contrast, insert a dedicated control layer between agents and tools, centralizing security policy enforcement. Each approach creates distinct security vulnerabilities and mitigation opportunities that system designers must carefully evaluate based on their specific requirements.

It's important to distinguish between tool communication protocols and security architecture patterns. Standardized protocols like the Model Context Protocol (MCP) define agent communication with tools but don't inherently implement security boundaries. Such protocols can be used in either direct access or broker-mediated architectures. The key distinction we're highlighting is not how tools are called but rather where and how security policies are enforced across those calls.

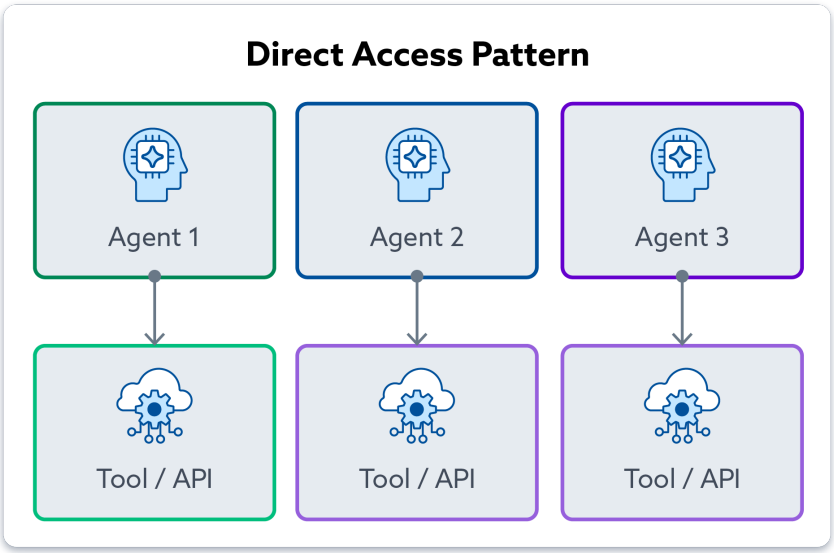
4.8.1 Direct Tool Access

In direct access patterns, agents invoke tools through direct function calls or API client integration, with security controls implemented either at the agent level, within individual tools, or through framework-level mechanisms. This distributed approach to security can offer implementation flexibility and performance benefits but may lead to inconsistent enforcement across different tools within the system.

For example, when building with modern agent frameworks, this might look like:

- Tool functions registered directly within the agent's context
- API clients embedded within agent classes
- Direct function calling where the agent's execution engine directly interfaces with external APIs
- Tools defined as callable functions within the agent's codebase with minimal wrappers, such as OpenAI's function calling or Anthropic's Model Context Protocol

Security in this pattern often relies on the underlying AI model's inherent limitations or guardrails built into the agent's code rather than tool-specific validation.



Feature	Direct Access Patterns	Broker-Mediated Patterns
Architectural Complexity	Low; direct agent-to-tool connections	Medium to high; additional broker component required
Security Validation	Individual validation within each tool (varies widely)	Centralized, consistent validation at broker
Permission Management	Potentially inconsistent and fragmented	Centralized and consistent management
Risk of Tool Misuse	Higher; due to direct unrestricted access	Lower; broker applies explicit usage policies
Operational Complexity	Lower; fewer intermediary components	Higher; additional middleware to manage
Monitoring and Auditing	Challenging; distributed and fragmented logs	Simplified; centralized logging and auditing
Attack Surface	Broader; each agent-tool interaction is a potential vector	Narrower; controlled and secured by broker component
Scalability	Simple for smaller systems, challenging at scale	Better scalability through centralized control

Performance Overhead	Minimal; direct tool interactions	Moderate; additional latency from broker layer
Typical Use Cases	Small, simple agentic applications	Complex, regulated, or enterprise-grade systems
Security Controls Implementation	Tool-level, potentially inconsistent	Consistent, centralized security validation and logging

4.8.1.1 Risk: Uncontrolled Tool Selection and Sequencing

In direct access models, agents can autonomously determine which tools to use and in what sequence without any intermediary control points. Unlike traditional software, where tool access is hardcoded into application logic, agentic systems can make dynamic decisions about tool selection based on their understanding of tasks. This creates a fundamental security challenge as agents may access capabilities inappropriate for their current context or combine tools in ways system designers didn't anticipate.

Security Impact

Unauthorized capability access: Agents may invoke technically available tools that are inappropriate for the current user context or task requirements.

Unexpected tool combinations: Agents can chain together permitted tools to achieve effects beyond what any individual tool was designed to enable, potentially bypassing security boundaries.

Mitigations

Implement function-level access controls: Rather than making all tools available in the agent's environment, implement explicit allow-lists of permitted tools based on the agent's role and context.

Create tool usage policies: Define explicit policies for which tools can be combined and under what circumstances, encoding these constraints directly in the agent's configuration.

Implement post-execution validation: For sensitive operations, validate executed tool chains against predefined patterns to detect unexpected tool combinations.

4.8.1.2 Risk: Inconsistent Security Controls Across Tools

Direct access architectures often result in each tool implementing its security checks (if any), creating a fragmented security perimeter. As agents interact with multiple tools through direct function calls or imports, varying levels of security validation may result. This inconsistency allows agents to find and exploit the least-secured tools within their reach.

Security Impact

Security control variance: Protection strength depends on individual tool implementations rather than consistent system-wide standards.

Detection blindspots: Security events in one tool may not correlate with those in another, making it difficult to identify patterns of potentially malicious behavior.

Mitigations

Implement wrapper libraries: Create standardized security wrappers around directly accessed tools to enforce consistent validation.

Standardized security interfaces: All tools must implement a familiar security interface, ensuring baseline controls are applied regardless of the tool.

Deploy central logging and monitoring: Even with direct access, ensure all tool usage is centrally logged to enable system-wide visibility.

4.8.1.3 Risk: Excessive Ambient Authority

Direct access architectures typically require the agent environment to possess all the permissions needed to access any tool the agent might use. This creates an environment with ambient authority far exceeding what's necessary for any single operation, increasing the potential blast radius if the agent is compromised.

Security Impact

Privilege concentration: Agent environments accumulate the union of all tool permissions, resulting in unnecessarily elevated privileges.

Attack surface expansion: A vulnerability in any agent component potentially exposes all directly accessible tools.

Mitigations

Compartmentalize agent environments: Divide agents into purpose-specific instances with access only to relevant tool subsets rather than giving all agents access to all tools.

Implement just-in-time access: Use ephemeral credentials or temporary permission grants that provide access only for the duration of specific operations.

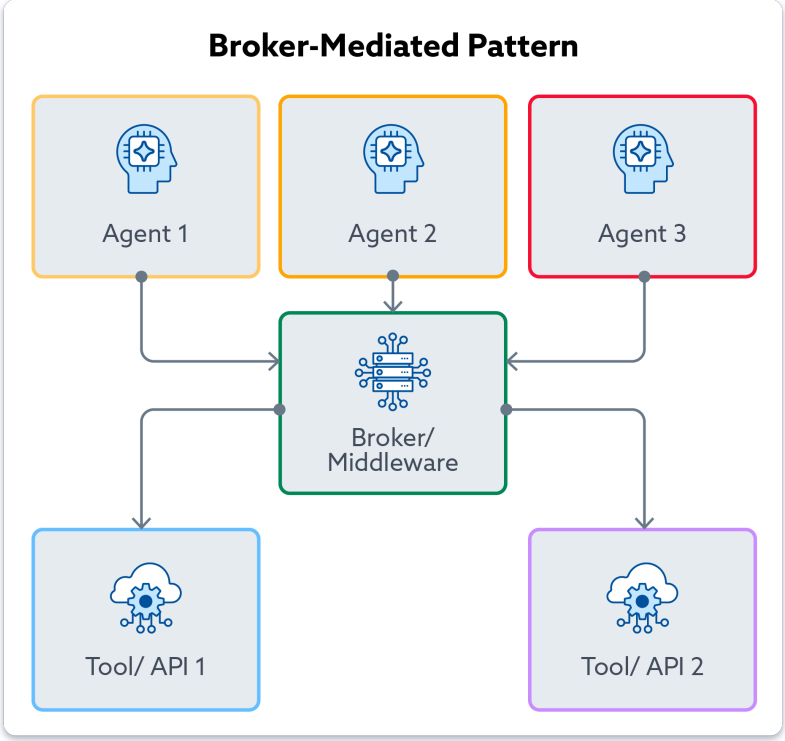
4.8.2 Broker-Mediated Access

Broker-mediated access introduces a dedicated service or component that creates a clear security boundary by intercepting, validating, and managing all tool requests. This centralized control point allows for consistent security policy enforcement across all tools, regardless of implementation. Unlike direct access, where security controls may be distributed and potentially inconsistent, broker-mediated patterns provide a unified point for implementing security governance.

This typically manifests as:

- **API management/gateway services** - Existing API gateways extended with agent-specific policies and validation rules to control which tools an agent can access and under what conditions
- **Security middleware layers** - Custom middleware that intercepts agent requests to internal APIs and enforces permission policies, rate limits, and input validation
- **Semantic firewalls** - Specialized "AI firewalls" that sit between agents and backend systems, validating requests based on policies and sometimes using secondary AI models to evaluate intent
- **Platform-provided validation layers** - Some LLM platforms implement security validation where tool calls must pass through their API layer for parameter validation and permission checks before execution
- **Agent orchestration platforms with security controls** - More advanced agent frameworks that implement not just tool registries but also security policies that control which agent components can access which tools based on context and permissions

Rather than agents having direct import or function call access to tools, they must formulate explicit requests that pass through this control layer, which can apply consistent validation, rate limiting, permission checks, and other security controls.



4.8.2.1 Risk: Request Intent vs. Syntax Validation Gap

Broker services may validate tool request syntax, parameters, and basic permission requirements, but struggle to understand the semantic intent behind an agent's actions. This creates a fundamental gap where technically valid requests with malicious intent can pass broker validation, particularly when agents formulate requests that comply with validation rules while achieving unintended outcomes.

Security Impact

Intent blindness: Brokers validate what's being requested but not why, potentially allowing misuse that meets syntactic requirements.

Security bypass through valid patterns: Agents may discover how to reframe prohibited operations as technically permitted ones that achieve similar effects.

Mitigations

Understanding agent intent represents a significant gap in current security systems. While theoretical approaches like intent classification models and outcome prediction exist, they remain primarily

impractical for most implementations in 2025. In light of these shortcomings, system designers should consider a combination of preventative and detective controls to address this risk:

Implement tight schema constraints: Define narrow, specific schemas for tool parameters that limit potential misuse, even when intent can't be assessed.

Deploy comprehensive logging and monitoring: Since preventing all misuse is impractical, prioritize robust logging of all broker actions to enable post-execution analysis, incident response, and continuous improvement loops.

Establish behavioral baselines: Monitor tool usage patterns to detect anomalies indicating misuse, even if individual requests appear valid.

Apply context-based restrictions: While not intent-aware, implement context-sensitive policies that restrict certain tool combinations or sequences based on the current task or user context.

As the field matures, more sophisticated intent analysis may become practical. Still, current systems should focus on constraining the potential damage through well-defined boundaries and rapid detection of unexpected behavior.

4.8.2.2 Risk: Broker Availability Creating Single Point of Failure

In broker-mediated systems, all tool access depends on broker availability and proper functioning. The entire agent's tool capability is affected if the broker service experiences disruption through technical failure, resource exhaustion, or attack. This creates a centralized vulnerability that is not present in direct access models.

Security Impact

System-wide capability loss: Broker failure means complete loss of tool functionality across the entire agent system.

Denial-of-service vulnerability: Attackers can potentially target the broker to disable agent capabilities rather than attacking individual tools or agents.

Mitigations

Implement broker redundancy: Deploy multiple instances with automatic failover capabilities to prevent single-point failures.

Design degraded operation modes: Create fallback mechanisms for critical tools that can operate under alternative, more restricted validation if the primary broker is unavailable.

Apply load shedding and prioritization: Consider designing brokers to handle overload conditions by prioritizing critical operations rather than failing.

4.8.2.3 Risk: Cross-Request Correlation Blindness

Brokers typically validate individual requests in isolation, examining each tool call's merits without considering patterns across multiple requests. This creates a vulnerability where agents could distribute a sensitive operation across multiple seemingly innocuous requests that individually pass validation but collectively achieve unauthorized outcomes.

Security Impact

Distributed attack blindness: Unable to detect malicious intent spread across multiple valid-looking requests.

Sequential exploitation: Agents could build up state or access through a sequence of permitted operations that together create security violations.

Mitigations

The challenge of correlating requests across time is a significant limitation in broker architectures. While comprehensive sequence analysis tools are still emerging, system designers might consider these more immediately practical approaches:

Implement basic request chaining constraints: Set limits on the volume or frequency of related tool calls within a session to contain potential distributed attacks.

Use simplified pattern detection: Deploy basic rules that flag specific sequences of operations known to be potentially problematic (e.g., read sensitive data followed by external communication).

Enhance session context tracking: Maintain a basic session state to identify when an agent has accessed specific sensitive resources and apply elevated scrutiny to subsequent operations.

Leverage existing security information and event management (SIEM) tools: Rather than building complex correlation within the broker itself, feed broker logs into existing SIEM platforms that already have some capability to detect patterns across events.

4.9 Tool Execution Context

Tool execution context patterns determine the identity and permission model under which tools operate when invoked by agents. This dimension establishes the impact radius of tool actions and defines how privilege boundaries are maintained throughout the tool execution lifecycle. Previous CSA papers[2] cover this in much greater detail than we're able to below.

The execution context fundamentally shapes what actions tools can perform once invoked, regardless of how access is controlled. Tools executing under broad agent service identities gain operational simplicity but potentially excessive permissions. User delegation models maintain clear accountability but introduce

[2] https://cloudsecurityalliance.org/artifacts/securing-llm-backed-systems-essential-authorization-practices

complex permission management challenges. Least-privilege approaches minimize potential damage from compromised tools but require more sophisticated identity and permission architectures. These patterns create distinct security trade-offs that directly influence how effectively other security controls can constrain potentially malicious tool usage.

4.9.1 Agent Service Identity

In this pattern, tools operate under a service account or identity assigned to the agent, inheriting a consistent set of permissions regardless of which user the agent serves or which tool is being invoked.

In practical implementations, this pattern manifests as:

- A dedicated service account was created specifically for the agent application
- API keys or credentials embedded in the agent's environment variables or configuration
- Cloud IAM roles assigned to the agent's compute resource, such as a VM service principal or container identity
- Secrets injected directly into the application or container at runtime
- Managed identities in cloud platforms that authenticate the hosting environment

The agent consistently uses these same credentials for all tool operations regardless of which user it's serving, with access controls implemented at the agent level rather than varying by user or operation.

For additional reference on this topic, please refer to CSA's paper on AI Identity Management.

4.9.1.1 Risk: Insufficient Permission Granularity

When all agent operations run under a single service identity, organizations struggle to apply appropriate permission boundaries to different functions. This forces a binary choice: either grant broad permissions that create unnecessary exposure for routine operations or limit permissions and restrict agent capabilities. In practice, this often leads to overprivileged agent services as new tool integrations require additional permissions that accumulate over time.

Security Impact

Privilege escalation potential: Agents operating with service-account level permissions require all the permissions necessary to do all tasks potentially assignable to them, making it easier to manipulate them into doing things that an individual calling the service may not have the right to do.

Shadow permission accumulation: As tools are added, permissions may expand incrementally without a comprehensive review.

Mitigations

Implement permission tiers: Create distinct service identities for different sensitivity levels (read-only, data modification, administrative).

Use temporary permission elevation: Default to minimal permissions with just-in-time elevation for specific operations requiring higher privileges that carry additional validation.

Deploy permission analytics: Regularly audit actual permission usage against granted permissions to identify and remove unused privileges using commonly available tooling.

4.9.1.2 Risk: Audit Trail Attribution Challenges

When multiple users interact with an agent using the same service identity, security teams face significant challenges attributing actions to specific user requests. Logs show the agent service account performing all actions, obscuring which user initiated sensitive operations. This complicates incident investigations, creates regulatory compliance challenges, and makes it difficult to detect potentially abusive patterns by specific users.

Security Impact

Investigation roadblocks: Security incidents involving agent tools show only service account activity, hindering root cause analysis.

Compliance violations: Fails to meet regulatory requirements for user activity tracking in regulated industries.

Threat blindness: Unable to detect patterns of abuse by specific users leveraging agent capabilities.

Mitigations

Implement request context logging: Enhance service account logs and request tracing with the originating user context to aid correlation and investigations.

Deploy correlation IDs: Use request tracing with unique identifiers that flow from user requests through all agent and tool operations.

4.9.1.3 Risk: Credential Rotation Complexity

Using a single identity across the entire agent system turns credential rotation into a high-risk, high-impact operation. Organizations often delay these security practices due to the potential for system-wide disruption. When rotations occur, they typically require more extended transition periods with multiple valid credentials, increasing the window of vulnerability if credentials are compromised.

Security Impact

Stale credential risk: Organizations delay rotation due to operational impact or complexity, increasing the risk of compromised credentials over time.

System-wide disruption: Failed rotation attempts can cause broad service outages across all agent capabilities.

Mitigations

Implement secret management systems: Use tools that can automate credential rotation workflows across distributed systems.

Use staggered rotation windows: Rotate credentials in phases across different subsystems to limit the potential impact.

Use dedicated credentials: Assign a unique service account, workload identity or credential to each tool and agent pair. This enables rotation of individual credentials one at a time, minimizes overlap windows, limits blast radius, and avoids system-wide disruption during rotation.

4.9.2 User Delegated Credentials

Tools execute with the permissions of the end user the agent is acting on behalf of, ensuring operations remain within the bounds of what the user could perform directly.
This pattern may manifest as:

- OAuth flows where user tokens are obtained and used for downstream API calls
- Session cookies or tokens passed through from user sessions to backend services
- OIDC token exchange where user identity is preserved across service boundaries

The critical aspect is that the agent has no permanent credentials but operates with credentials representing the end user's identity and permissions.

4.9.2.1 Risk: Access Control Boundary Confusion

In delegation models, system designers struggle to define which operations should be restricted at the agent level versus allowed based solely on user permissions. For example, should an agent be allowed to perform mass deletions just because the user has that right? This creates inconsistent security implementations where some sensitive operations might have additional agent-level gates while others rely entirely on user permissions.

Security Impact

Inconsistent protection models: Critical operations are protected inconsistently as some user permissions will vary from others.

Inference failures: Agents may incorrectly infer user intent, and with broad permissions assigned to the user, agents may perform unintended high-impact operations.

Mitigations

Implement explicit operation allowlists: Define specific operations permitted through agent interfaces regardless of user permissions.

Deploy operation sensitivity tiers: Categorize operations by impact and require additional confirmation for higher-tier actions.

Consider agent-specific permission scopes: Define scopes specifically for agent-mediated actions distinct from direct user permissions.

4.9.3 Pattern: Least-Privilege Service Identity

Tools operate with minimal, function-specific service accounts that are scoped precisely to the permissions required for their intended functionality and nothing more.

In practical implementations, this pattern typically manifests as:

- Multiple service accounts with different permission sets based on function categories
- Separate API keys for different external services with minimal necessary scopes
- Role-based access controls where the agent assumes different roles based on the operation

4.9.3.1 Risk: Agent Proliferation Management Complexity

Organizations must deploy, manage, and maintain multiple limited-scope agent instances rather than a single versatile system. This proliferation creates operational overhead, complicates updates, and makes maintaining consistent security controls across the agent ecosystem difficult.

Security Impact

Inconsistent security controls: At scale, security measures can quickly end up implemented unevenly across proliferated agent instances.

Configuration drift: Multiple agent instances may develop security configuration variations over time.

Mitigations

Implement agent templates: Create standardized agent deployment templates with embedded security controls.

Deploy central agent registries: Maintain a central inventory of all agent instances with security configuration tracking.

4.9.3.2 Risk: Cross-Agent Workflow Fragmentation

When tasks require capabilities spanning multiple purpose-specific agents, workflows become fragmented across agent boundaries. This creates new security challenges in securely passing context and data between agent instances while maintaining proper authorization boundaries.

Security Impact

Insecure data passing: Ad-hoc methods to transfer context between agents create data exposure risks.

Authorization boundary bypasses: Workflows spanning multiple agents may circumvent authorization checks at boundary transitions.

Inconsistent security context: Security context and controls may not correctly transfer between agent boundaries.

Mitigations

Leverage workflow orchestration tools: Leverage established workflow orchestration platforms to manage complex flows.

Implement coordinated message passing: Use message queues or event buses for inter-agent communication.

Create cross-agent audit trails: Ensure audit records maintain continuity across agent transitions for end-to-end visibility.

4.9.3.3 Risk: Agent Permission Sprawl

In pursuit of the least privilege, organizations can create numerous purpose-specific agents, each with minimal permissions. However, unless carefully curated, this proliferation can lead to a difficult-to-manage security landscape where the collective permission footprint becomes unclear, inconsistent, and difficult to govern effectively.

Security Impact

Visibility gaps: Security teams may struggle to understand effective permissions across the entire agent ecosystem.

Redundant capability confusion: Different agents approaching the same problem with varying permissions can be difficult to manage at scale.

Mitigations

Implement capability registries: Maintain central tracking of agent capabilities with associated permission requirements to minimize accidental proliferation.

Establish boundary governance processes: Develop explicit review processes for new agent capabilities to prevent permission boundary erosion.

Automated permissions pruning: Use available tools that monitor permission usage and remove unused permissions over time to prevent sprawl.

Key questions for system designers

What mechanisms validate not just the syntax of tool requests but also potentially malicious intent across sequences of operations?

Brokers typically validate individual requests in isolation, creating blindspots where agents could distribute unauthorized operations across multiple seemingly innocuous requests. By default, systems cannot correlate related tool calls and identify potentially dangerous patterns that span multiple valid-looking operations.

How are you detecting and controlling when agents combine individually-permissible tools in ways that bypass security boundaries?

Unlike traditional applications with hardcoded workflows, agents dynamically select and sequence tools based on their understanding of tasks. Without explicit controls on tool combinations, agents may chain permitted tools to achieve effects beyond what any individual tool was designed to enable, potentially circumventing security boundaries.

Under what identity and permission model do your agent's tools execute, and how does this choice impact your security perimeter?

The execution context fundamentally determines what actions tools can perform once invoked. Whether tools operate under broad agent service identities, user delegation, or least-privilege approaches creates distinct security trade-offs that directly influence how effectively other security controls can constrain potentially malicious usage.

What mechanisms ensure that tool access remains appropriately scoped when your agent serves multiple users with different permission levels?

As agents serve users with varying authorization levels, permission boundaries become complex. Without clear mechanisms to scope tool operations to appropriate user contexts, agents risk performing actions that exceed a specific user's intended permissions, especially in shared agent environments.

How do you maintain visibility into tool operation chains to understand what happened, why it happened, and who initiated it?

When agents orchestrate sequences of tool operations, attribution, and causality become difficult to track. Without end-to-end tracing mechanisms that preserve context across chained tool calls, organizations struggle to investigate incidents, understand agent behavior, or identify the specific user request that triggered sensitive operations.

What happens to your security posture if your tool access control broker becomes unavailable?

In broker-mediated systems, all tool access depends on broker availability. Understanding how your system behaves during broker degradation or failure helps prevent situations where security completely collapses, or critical functionality is lost when the broker is unavailable.

How will your approach to managing tool permissions scale as you add more agent capabilities and integrate with additional systems?

As agent systems grow, they typically accumulate tools and permissions over time. Without deliberate strategies for permission management at scale, organizations risk permission sprawl, visibility gaps, and an expanding attack surface that becomes increasingly difficult to govern effectively.

5. Conclusion

This document has argued that traditional, deterministic security models are insufficient to address the complexities and emergent behaviors inherent in these autonomous, adaptive, and often interconnected systems. The core contribution of this work is the **trait-based approach**, a novel framework for understanding and mitigating the unique security risks that arise from the very nature of agentic systems.

This approach moves beyond simply listing potential vulnerabilities. Instead, it provides a structured methodology for:

Deconstructing Complexity: By breaking down agentic systems into their fundamental behavioral traits (control, communication, planning, perception, learning, trust, and tool usage), we gain a clearer understanding of the building blocks that give rise to both their power and their potential weaknesses.

Anticipating Emergent Risks: The trait-based approach emphasizes the *interactions* between these traits. This is crucial because many security vulnerabilities in agentic systems are not inherent to individual components but emerge from the combination of seemingly benign design choices. The framework helps designers anticipate these emergent behaviors and their security implications.

Integrating Security by Design: The trait-based approach is not a post-deployment checklist; it's a design philosophy. By mapping traits to specific risks and mitigations, security considerations are woven into the very fabric of the system architecture from the outset. This proactive approach is essential, given the adaptive and potentially unpredictable nature of agentic systems.

Providing Actionable Guidance: The document provides concrete examples of trait patterns (e.g., centralized vs. decentralized control, direct vs. indirect communication), along with their associated risks and specific mitigation strategies. The "Key Questions for System Designers" sections throughout the document offer practical prompts to guide critical thinking and decision-making during the design process. This is not just theory; it's a practical tool.

Adapting to a Rapidly Evolving Landscape: The trait-based approach is designed to be flexible and extensible. As new agentic architectures and technologies emerge, the framework can be adapted by adding new traits, refining existing ones, and updating the associated risk and mitigation analyses. This is crucial in a field that is developing at such a rapid pace.

5.1 Key Takeaways and Call to Action

Shift in Mindset: Securing agentic systems requires a fundamental shift from a "perimeter defense" mentality to a "Zero Trust, continuous verification" approach that acknowledges the inherent autonomy and adaptability of agents.

Proactive, Not Reactive: Security must be considered at the architectural level, not as an afterthought. The trait-based approach enables this proactive stance.

Collaboration is Essential: This document highlights the need for close collaboration between AI practitioners, security professionals, and system architects. The security challenges of agentic systems cannot be solved in isolation.

Continuous Learning and Adaptation: The field of agentic AI is rapidly evolving. Continuous monitoring, threat modeling, and adaptation of security controls are essential. The framework presented here provides a foundation for this ongoing process.

Standardization and future work: This document represents an early step in establishing best practices and building a safer landscape for the agentic system. The Cloud Security Alliance recognizes this working draft as a start of a longer discussion, and that future work is required, specifically to establish clear benchmarks, and formal standards in this evolving landscape. We encourage active engagement to the evolution of this discussion, fostering a more secure, trustworthy ecosystem for Agentic AI.

This document is a starting point for a crucial conversation. The Cloud Security Alliance encourages active participation and feedback from the community. By working together, we can harness the immense potential of agentic systems while mitigating the risks and ensuring a secure and responsible future for this transformative technology. The future of software is autonomous, and the trait-based approach provides a roadmap for building that future securely.

Glossary

CSA Glossary (main/primary)

References

- Agentic AI Threat Modeling Framework: MAESTRO | CSA - Ken Huang
- Agentic AI – Threats and Mitigations - OWASP
- THE LANDSCAPE OF EMERGING AI AGENT ARCHITECTURES FOR REASONING, PLANNING, AND TOOL CALLING: A SURVEY - IBM
- On the Resilience of LLM-Based Multi-Agent Collaboration with Faulty Agents
- AI TRiSM
- AI-TMM
- Agentic AI Identity Management Approach