Authenticated Workflows: A Systems Approach to Protecting Agentic AI

Mohan Rajagopalan
MACAW Security, Inc.
mohan@macawsecurity.com

Vinay Rao
ROOST.tools
vinay@roost.tools

***Abstract*—Agentic AI systems automate enterprise workflows but existing defenses—guardrails, semantic filters—are probabilistic and routinely bypassed. We introduce authenticated workflows, the first complete trust layer for enterprise agentic AI. Security reduces to protecting four fundamental boundaries: prompts, tools, data, and context. We enforce intent (operations satisfy organizational policies) and integrity (operations are cryptographically authentic) at every boundary crossing, combining cryptographic elimination of attack classes with runtime policy enforcement. This delivers deterministic security—operations either carry valid cryptographic proof or are rejected. We introduce MAPL, an AI-native policy language that expresses agentic constraints dynamically as agents evolve and invocation context changes, scaling as O(log M + N) policies versus O(M×N) rules through hierarchical composition with cryptographic attestations for workflow dependencies. We prove practicality through a universal security runtime integrating nine leading frameworks (MCP, A2A, OpenAI, Claude, LangChain, CrewAI, AutoGen, LlamaIndex, Haystack) through thin adapters requiring zero protocol modifications. Formal proofs establish completeness and soundness. Empirical validation shows 100% recall with zero false positives across 174 test cases, protection against 9 of 10 OWASP Top 10 risks, and complete mitigation of two high impact production CVEs.**

1. Introduction

Enterprises are struggling to deploy agentic AI systems in production. While these systems promise to automate complex workflows—managing financial transactions, patient records, and critical infrastructure—they introduce security challenges that existing defenses cannot address. For example, within hours of releasing the OpenAI Atlas browser, researchers demonstrated that malicious instructions embedded in webpage content could trigger the assistant to exfiltrate credentials [1]. OpenAI's CISO acknowledged that "prompt injection remains an unsolved problem" [2].

The challenge runs deeper than prompt injection alone. LLMs cannot distinguish instructions from data. Non-deterministic execution paths cannot be predicted or statically analyzed. Multi-turn interactions allow gradual context manipulation where each message appears benign. Existing approaches look for patterns or are probabilistic—both require enumerating attacks, leaving defenders grappling with unknown unknowns.

We eliminate unknown unknowns through a systematic approach: instead of enumerating attacks, we bound the system deterministically. We build on the realization that agentic systems can be abstracted as simple, byzantine, distributed systems – multiple entities interacting across well defined boundaries. Now security reduces to protecting boundary crossings. Conceptually, our approach is based on satisfying two properties at each boundary crossing : understanding **intent** ensuring operations satisfy organizational policies, and enforcing **integrity** to ensure operations are authentic and unmodified. Both are necessary, and neither is sufficient by itself. This design reduces agentic workflows to a deterministic distributed system where boundaries are guarded by policies and crossings are cryptographically protected. Breaking the system requires breaking cryptography, not crafting clever prompts. Many classes of attack such as identity spoofing, session replay, policy substitution etc are eliminated by design – they must break cryptographic primitives to succeed, others such as unauthorized data access, privilege escalation, credential exfil etc are blocked by policy at runtime. Architecturally, **by-policy enforcement** and **by-design elimination** deliver complementary, composable defense-in-depth.

We studied 100+ agentic applications and identified key challenges: **(1) Four simultaneous attack surfaces**—tools, data, prompts, context—where attacks compose across surfaces to achieve objectives impossible through any single surface. **(2) Heterogeneous frameworks**—Agents are built using protocols (MCP, A2A), LLM interfaces (Claude, OpenAI), orchestrators (LangChain, CrewAI, AutoGen). Heterogeneity is a permanent ecosystem feature. **(3) Compositional security gaps**—per-framework checks miss violations spanning compositions. **(4) Developer burden**—requiring security logic at every interaction across heterogeneous frameworks is impractical.

Combined, the attack surface grows faster than tools can address, highlighting a critical gap: a trust layer for AI—security infrastructure positioned between application-layer defenses (don't compose) and infrastructure primitives (lack workflow semantics).

To address (1) we introduce **authenticated workflows**—a protocol-level primitive enforcing *intent* and *integrity* at every interaction. We prove the 4 attack surfaces are minimal and complete. Protocol-level positioning addresses heterogeneity (2) and eliminates compositional gaps (3). Thin wrappers across 9 popular frameworks resolve developer burden (4), establishing the foundation for an enterprise AI trust layer.

Authenticated workflows combine cryptography, zero-trust principles, and runtime policy enforcement to ensure each operation is authentic, authorized, and attested. We present a novel design using distributed Policy Enforcement Points (PEPs) embedded at control surfaces, verifying cryptographic proofs independently with sub-millisecond overhead, requiring no centralized infrastructure. Each surface verifies operations independently before execution, creating defense in depth where compromising one surface does not compromise others.

To specify *intent* we introduce MAPL, a new AI-native policy language that lets users express agentic constraints in a dynamic, scalable manner even as agents evolve and invocation context changes. MAPL provides three key capabilities: *composable grammar with inheritance* enabling policies to layer and compose through intersection semantics; *radically reduced specification* scaling from $O(Musers \times Nresources)$ to $O(\log M + N)$ through hierarchical composition and implicit principal resolution; and *dynamic state via attestations*—signed claims proving operations completed, enabling policies like ”export data only after anonymization-completed attestation exists” with cryptographic enforceability. These advances overcome limitations of traditional policy systems (OPA, AWS Cedar) in agentic environments.

Our focus is control-flow protection across operational boundaries—tool invocations and data retrievals—complementing Rajagopalan et al. [3], which secure data flowing through prompts and context. Together, these deliver a formal trust layer for enterprise AI. We prove practicality through a universal security runtime backing nine frameworks (Section 5) via thin adapters (200–500 LOC) requiring no protocol changes—demonstrating framework-agnostic security.

Formally, we prove that the four boundaries are complete and minimal, distributed enforcement is sound, and that policy composition preserves security properties. Empirically, we validate these properties through comprehensive attack testing and production CVE analysis.

Current agentic defenses—guardrails, prompt filters, semantic analysis (pattern matching, ML models, heuristics)—prioritize precision but deliver low or zero recall. Adversarial prompts, encoding tricks, and novel patterns routinely bypass them via semantic blind spots, while false positives block legitimate operations. In contrast, our cryptographic enforcement guarantees high precision and high recall: operations either bear valid signatures satisfying policies or are rejected outright. Breaking the system demands shattering cryptographic primitives—not crafting clever patterns. This yields complete violation detection, zero false positives, and deterministic trust via computational hardness.

Contributions. We present authenticated workflows—protocol-level primitives enforcing cryptographic verification across four control surfaces (prompts, tools, data, context), proven complete and minimal (Lemma 6). We introduce MAPL, an AI-native policy language providing cryptographic workflow dependencies via attestations and reducing policy specification from $O(M \times N)$ to $O(\log M + N)$ through hierarchical composition (Theorems 1-3). We demonstrate universal deployment across nine heterogeneous frameworks (MCP, A2A, OpenAI, Claude, LangChain, CrewAI, AutoGen, LlamaIndex, Haystack) through thin adapters (200-500 LOC) requiring zero protocol modifications, with distributed Policy Enforcement Points embedded at every boundary providing zero-trust verification with sub-millisecond overhead. Formal proofs establish completeness and soundness (Lemmas 1-7); empirical validation demonstrates deterministic guarantees (100% recall, 0% false positives) across 174 test cases covering 9 of 10 OWASP Top 10 risks and complete mitigation of two production CVEs (OpenAI Atlas, GitHub MCP).

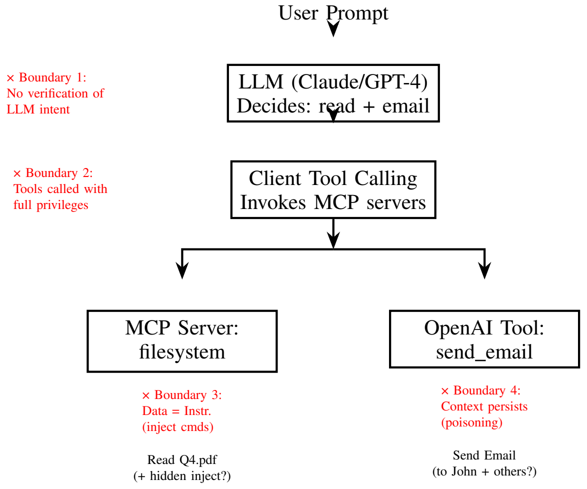


Figure 1. Agentic workflow showing attack cascade across four boundaries.

2. The Problem: How Agents Work

To understand what makes securing agentic systems hard, consider Figure 1, which shows entities involved when you invoke: “Read the Q4 financial documents and email a summary to john@company.com.” The user’s prompt flows to the LLM carrying instructions. The LLM reasons and decides to invoke filesystem read and email send operations—MCP tools, OpenAI function calls, or custom implementations. These tool invocations retrieve data from document storage. As the agent executes this multi-turn workflow, it maintains application-level state tracking interaction history.

These interactions cross four fundamental boundaries. **S1—Prompts** carry instructions into the LLM, directing reasoning and tool selection. **S2—Tools** execute privileged operations—filesystem access, database queries, API calls, email sending. **S3—Data** flows from external sources into agent reasoning—document stores, RAG corpora, vector databases, web scraping. **S4—Context** maintains conversational state across multi-turn interactions. A fundamental challenge: the LLM processes prompts and document contents identically—malicious instructions in Q4.pdf are indistinguishable from user requests. The system operates on implicit trust: context is assumed valid, tools are presumed authorized, data sources treated as benign. These boundaries are framework-agnostic—LangChain chains and CrewAI crews both manifest as prompt→LLM→tool sequences, all crossing the same four surfaces. This universality enables protocol-level security that works uniformly across heterogeneous systems.

Any surface can serve as an attack entry point with consequences cascading across others. In OpenAI's Atlas attack [1], malicious instructions embedded in data (S3) flowed into LLM reasoning (S1), generating tool invocations (S2) to exfiltrate credentials, all recorded in context (S4) as normal execution. Different attacks enter through different surfaces. We formalize this through our threat model.

Threat Model. We assume a sophisticated adversary and formalize the threat model through adversary capabilities and trust assumptions.

Adversary Capabilities A1—Application Control: Adversaries may control application-layer code. **A2—Content Injection:** They inject malicious content into data sources, prompts, or context. **A3—Component Compromise:** They may compromise components and obtain private keys. **A4—Network Attacks:** They intercept, replay, or modify network traffic. **A5—Compositional Attacks:** They perform gradual attacks across interactions or span framework boundaries where each operation appears authorized but collectively violates policies.

Trust Assumptions L1—Cryptographic Hardness: Cryptographic primitives provide computational hardness—adversaries cannot forge signatures without private keys. **L2—Trusted Control Plane:** Control plane services (Agent Registry, Policy Store, Logging, Routing) operate in a minimal trusted computing base with cryptographic integrity guarantees (hash chains for audit logs, Merkle trees for policy store) and administrative access controls. **L3—Enforcement Integrity:** PEP verification logic executes correctly at framework boundaries. Adversaries with application control (A1) can manipulate business logic but cannot bypass framework APIs or corrupt PEP memory in enterprise deployments where frameworks are immutable and operations route through instrumented APIs. Bypassing PEPs requires compromising framework binaries—equivalent to kernel compromise and out of scope. This parallels OS kernel security. System-level attacks (debuggers, memory corruption) are out of scope—even if such attacks bypass a local PEP, the remote side independently verifies invocations, bounding the adversary to their application scope without gaining new privileges.

Security Objectives Against this threat model, we establish five security objectives: **O1—Integrity:** Operations are authentic and unmodified. **O2—Policy Enforcement:** All operations satisfy organizational policies. **O3—Privilege Non-Escalation:** Composed policies cannot grant broader permissions than individual policies. **O4—Context Integrity:** Session state is tamper-evident across interactions. **O5—Accountability:** All operations have non-repudiable audit trails. Section 7 proves authenticated workflows achieve O1-O5 against adversaries with capabilities A1-A5 under assumptions L1-L3.

Scope We focus on protocol-level authorization: cryptographically enforcing that every operation satisfies organizational policy. We provide the mechanism—end users supply the policy. Application-level safety (e.g., detecting malicious code) requires domain-specific semantics and lies outside protocol scope. We treat all applications as untrusted, bound by system-level policies.

Current approaches are reactive—detecting patterns, filtering inputs, sandboxing execution. Each new attack variant requires new detection rules, creating an enumeration problem for an unbounded attack space. These adversary capabilities (A1-A5) target the four attack surfaces: application control and content injection compromise prompts/tools/data, while compositional attacks exploit multi-framework gaps. **A1, A3, A4** render observability-based solutions useless. Recent work addressing **A2** and **A4** with custom models [4] is probabilistic and cannot guarantee safety deterministically.

Our Approach At the protocol level, agent→tool, agent→LLM, and agent→data operations look identical—a boundary crossing. Our approach converts each of these into an authenticated workflow that is bound by policy (O2), verified for authorization (O1, O3, O4) and attested on execution (O5). In the next section 3 we describe a policy algebra that enforces O3. By-design mechanisms (cryptographic signatures, hash chains) address component compromise (A3) and network attacks (A4). By-policy mechanisms (runtime verification at boundaries) address application control (A1), content injection (A2), and gradual attacks (A5). Section 7 proves operations either carry valid cryptographic proof satisfying policies, or are blocked before execution—deterministic guarantees, not probabilistic detection.

Revisiting the example attack: Each boundary is protected by independent policies. Even if reasoning is corrupted, restricted documents remain inaccessible. After compromise, the application is restricted to what policy permits. Breaking the system requires simultaneously breaking all enforcement layers—computationally hard.

The problem decomposes into two questions: How do we specify policies that govern boundaries? How do we verify integrity at runtime? We introduce MAPL (Section 3) for expressing intent and authenticated workflows (Section 4) for enforcing integrity, implemented via a universal trust layer (Sections 6–7).

3. Specifying Intent: MAPL Policy Language

We introduce MAPL, an AI-native policy language to specify intent. Agentic systems exhibit dynamism (agents morph identities, spawn sub-agents, delegate capabilities) and scale (10-100x growth per user) that break traditional policy engines. Three properties challenge existing systems: **Contextual identity**—agents represent different principals based on runtime context, requiring dynamic principal resolution rather than static WHO bindings. **Dual perspective**—maintaining integrity requires validating every invocation from both caller intent and resource constraints independently; traditional policies tightly couple caller and resource, where compromising the caller grants resource access. **Verifiable workflow state**—multi-step workflows need sequential enforcement, since application-reported state is untrusted, we need some form of attestation to verify operation A completed before invoking operation B.

MAPL addresses these through three design choices: (1) Principals inferred from authenticated context at runtime—enabling dynamic identity without policy updates; (2) Caller and resource policies expressed independently, composing via intersection—achieving defense in depth without coordination; (3) Policies reference cryptographically verified attestations—enabling provable workflow dependencies. The combination creates a complete, provable, compositional model for expressing intent.

Policy Grammar. MAPL (MAPL Agentic Policy Language) policies follow a structured grammar enabling machine verification and automated composition:

```
Policy {
  policy_id: <unique_identifier>,
  extends: <parent_policy_id>,
  resources: [<resource_patterns>],
  denied_resources: [<denial_patterns>],
  constraints: {
    parameters: {<resource>: {<param>: [<patterns
        ↪ >]}},
    denied_parameters: {<resource>: {<param>: [<
        ↪ patterns>]}},
    attestations: [<required_attestation_names>]
  }
}
```

A MAPL policy includes: *policy_id* a unique identifier for composition chains and cryptographic binding, *extends* the parent policy reference for hierarchical composition, *resources* allowed operation patterns; wildcards $*/**$ match single/recursive levels, *denied_resources* explicit blocks overriding allowances, and *constraints* (parameters , denied parameters, patterns and attestation requirements). Appendix A demonstrates concrete policies.

Hierarchical Composition. The **extends** field enables policies to inherit from parent policies through intersection semantics. This allows organizational policies to layer (base → department → team) where each child policy refines its parent by adding restrictions, never relaxing constraints. When a policy extends another, the effective permissions are the intersection of parent and child—maintaining monotonic restriction as the composition algebra formalizes below.

Expressiveness. The minimal grammar suffices through four mechanisms: (1) Hierarchical resources with wildcards express unbounded namespaces; (2) Positive and negative specifications express arbitrary boolean combinations; (3) Parameter constraints express any decidable predicate on arguments; (4) Attestations enable sequential constraints through cryptographic state. This covers organizational hierarchies, dual-perspective defense, workflow dependencies, exception patterns, and parameter controls.

Composition Algebra. We formalize how policies compose at runtime and prove the system satisfies key security properties.

Policy Structure and Operations.

A policy $P = (R, D, C)$ consists of:

- R: Allowed resource patterns
- D: Denied resource patterns
- C: Operational constraints (parameters, attestations)

Policy Intersection $P_1 \cap P_2 = (R', D', C')$ where:

- $R' = R_1 \cap R_2$ (allow only if both policies permit)
- $D' = D_1 \cup D_2$ (deny if either policy forbids)
- $C' = \text{MostRestrictive}(C_1, C_2)$ (apply tightest constraint)

$\text{MostRestrictive}(C_1, C_2)$ takes minimum values for numeric limits, intersection for allowed patterns, and union for required attestations and denied patterns.

Policy Enforcement Points compose policies from organizational hierarchy and dual perspectives at runtime. This ensures operations can only add restrictions, never relax them. If a policy is absent, it contributes no constraints (most permissive interpretation violating no stated constraint).

Formal Security Properties

Define the permission function:

$$\pi(P) = \{(r, op) \mid r \notin \text{Match}(D) \wedge r \in \text{Match}(R) \wedge op \text{ satisfies } C\}$$

Theorem 1 (Monotonic Restriction): For composition $P_0 \cap \ldots \cap P_n$:

$$\forall i, j : (i < j) \Rightarrow \pi(P_0 \cap \ldots \cap P_j) \subseteq \pi(P_0 \cap \ldots \cap P_i)$$

Proof Sketch: By construction, $P_{i+1} = P_i \cap P_{\text{next}}$. Resource intersection: $R_{i+1} \subseteq R_i$. Denial union: $D_{i+1} \supseteq D_i$. Therefore $\pi(P_{i+1}) \subseteq \pi(P_i)$. By induction, $\pi(P_j) \subseteq \pi(P_i)$ for $i < j$. □

Theorem 2 (Transitive Denial): If resource r is denied by any policy, it remains denied:

$$\exists i : r \in \text{Match}(D_i) \Rightarrow r \in \text{Match}(D_{\text{eff}})$$

Proof Sketch: $D_{\text{eff}} = D_0 \cup D_1 \cup \ldots \cup D_n$. If $r \in D_i$ for any i, then $r \in D_{\text{eff}}$ by set union. □

Theorem 3 (No Privilege Escalation): If base policy P_0 denies r, no composition grants access:

$$(r \in \text{Match}(D_0)) \Rightarrow r \notin \text{Allowed}(P_{\text{eff}})$$

Proof: By Theorem 2, if $r \in \text{Match}(D_0)$, then $r \in \text{Match}(D_{\text{eff}})$. By definition of π, denied resources cannot be allowed. □

Security Implications These theorems provide formal guarantees: **Theorem 1** prevents privilege expansion—adding policies can only narrow permissions; **Theorem 2** ensures any layer's denial is absolute; **Theorem 3** prevents escalation where combining policies grants denied permissions.

Together, these achieve policy enforcement (O2) and privilege non-escalation (O3) from Section 2. Even when adversaries compromise components and obtain their keys

(A3), the compromised component can only perform operations within its policy-permitted scope—monotonic restriction prevents privilege expansion, and transitive denial ensures root denials propagate through all derivations. These properties hold mathematically—independent of attacker sophistication or LLM behavior. Breaking them requires breaking cryptographic primitives (digital signatures, hash functions), not manipulation. **No arbitrary overrides**. MAPL explicitly disallows overrides—they break provability where Theorems 1-3 do not hold. Instead, administrators create temporary groups with time-bounded validity (analogous to Unix groups/permissions). These compose via intersection, maintaining formal guarantees while supporting operational flexibility. Appendix A demonstrates time-bounded emergency access.

Enabling Scale: From O(MxN) rules to O(log M + N) policies Traditional policy engines require O(M×N) explicit rules for M principals and N resources. Managing rules that scale combinatorially is impractical; MAPL provides a practical solution as enterprises scale agents across thousands of users. MAPL's hierarchical composition and dynamic pricipal binding eliminates this explosion by mirorring organizational structure – with a branching factor of 8-16, departments scale logarithmically: $D \approx \log_{8-16}(M)$—exponentially smaller than M, teams inherit upwards absorbing into the hierarchical tree. Resources grow with functionality, independent of user count. Thus MAPL requires $\log(M) + N$ policies – a significant, practical reduction.

Together, MAPL's compositional algebra addresses all three agentic requirements: contextual identity through runtime principal resolution, dual perspective through independent policy composition, and verifiable workflow state through cryptographic attestations.

4. Authenticated Workflows

We now introduce authenticated workflows—a protocol-level mechanism that realizes deterministic verification at every agentic boundary crossing. Against adversaries with capabilities A1-A5 (Section 2), authenticated workflows provide four **cryptographic guarantees** that together achieve security objectives O1-O5:

Authenticity (achieves O1: Integrity): Every accepted invocation is cryptographically linked to the principal that initiated it and the operation being performed. Attackers cannot forge invocations without possessing the principal's private key, binding both WHO (principal) and WHAT (operation).

Policy Binding (achieves O2-O3: Policy Enforcement, Privilege Non-Escalation): The policy governing an operation is cryptographically bound to the invocation, preventing attackers from substituting policies without invalidating signatures.

Tamper Evidence (achieves O4: Context Integrity): Any modification to invocations, context state, or policy bindings is cryptographically detectable through hash chains and sequence numbers.

Non-Repudiation (achieves O5: Accountability): Digital signatures create undeniable proof linking principals to their operations, enabling forensic analysis and compliance verification with tamper-evident audit logs.

As with authenticated system calls [5], [6], the core idea is conceptually simple: augment every inter-entity invocation with a policy and cryptographic signature that ensures integrity.

```
Invocation = (Args, Policy, MAC)
MAC = Sign(K, Args || Policy)
```

where the Message Authentication Code(MAC) signature cryptographically binds the arguments and policy identifier together using a secret or private key K. On the receiving end, verification proceeds in three steps: validate the signature to ensure authenticity and integrity, retrieve and verify the policy binding to prevent substitution, then evaluate whether the operation satisfies policy rules.

4.1. Design Principles

Realizing authenticated workflows requires four principles that compose to deliver the cryptographic guarantees:

Zero-Trust Identity: Every entity—agents, tools, data sources, LLMs—possesses unique cryptographic identity (keypair). Even entities in the same process verify each other independently. No implicit trust propagates. We bridge enterprise identity with agent-scale verification: External principals authenticate via enterprise IAM; the runtime propagates these identities as tamper-proof session context. Internally, lightweight ephemeral signatures identify agents, addressing dynamic identity challenges (Section 3).

Boundary Verification: Every inter-entity communication—prompt→LLM, agent→tool, tool→data—requires independent cryptographic verification. Operations carry signatures and policy identifiers; receivers verify signatures using registered public keys, retrieve policies, and evaluate constraints. Verification is transport-agnostic—identical across MCP, LangChain, OpenAI APIs, or custom protocols.

Policy Enforcement Points: PEPs provide independent verification at every boundary—entities cannot verify their own operations. Deployed in-process (linked library) or as sidecars, PEPs verify invocations before execution, providing horizontal scalability, zero-trust verification, tamper resistance, and enforcement independence. This relies on L3 (enforcement integrity).

Trustworthy Authenticated Context: Runtime provides tamper-evident session state such as from [3] that ensures integrity including binding operations to session scope (context IDs, session IDs), preventing replay (sequence numbers), detecting tampering (hash-chains), and enabling workflow dependencies (attestations proving prerequisite operations completed).

The Protocol. These principles compose into a four-phase protocol that every boundary crossing is an authenticated workflow. Figure 2 shows the complete flow.

Registration: Each entity registers, receiving (agent_id, keypair). Registration happens at finest granularity—individual tools within agents receive independent (tool_id, keypair), minimizing blast radius.

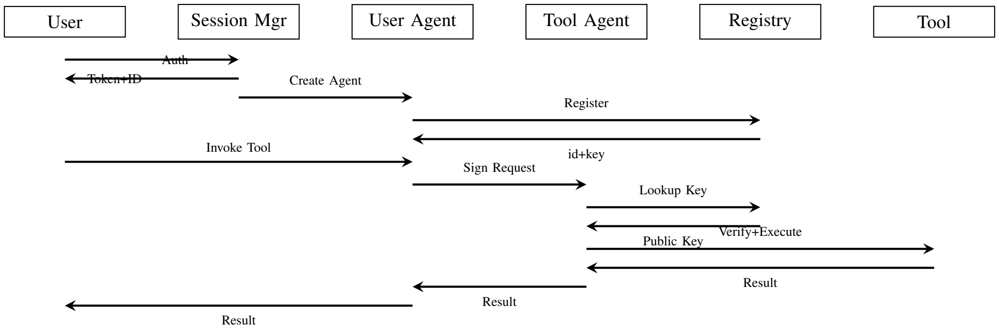


Figure 2. Registration and invocation flow showing authentication, entity registration, signed invocation, and bidirectional verification.

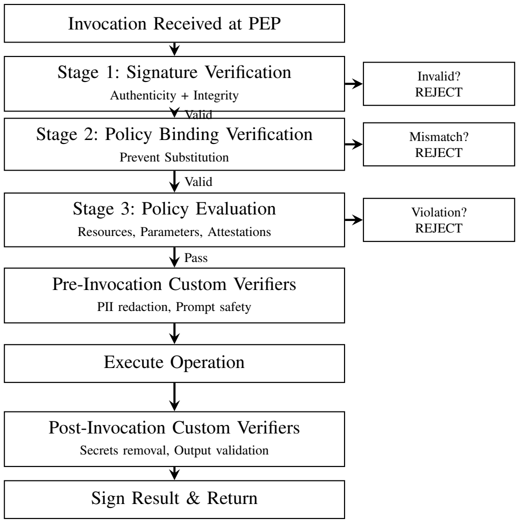

Invocation Received at PEP
Stage 1: Signature Verification
Authenticity + Integrity
Invalid?
REJECT
Valid
Stage 2: Policy Binding Verification
Prevent Substitution
Mismatch?
REJECT
Valid
Stage 3: Policy Evaluation
Resources, Parameters, Attestations
Violation?
REJECT
Pass
Pre-Invocation Custom Verifiers
PII redaction, Prompt safety
Execute Operation
Post-Invocation Custom Verifiers
Secrets removal, Output validation
Sign Result & Return

Figure 3. PEP verification flow with three-stage cryptographic verification and optional custom verifiers.

Invocation: Caller constructs signed invocation binding operation, arguments, policy, and session context.

Verification: The recieving PEP independently verifies through three stages: (1) signature verification (authenticity + integrity), (2) policy binding verification (prevent substitution), (3) policy evaluation (resource permissions, parameter constraints, attestations). Figure 3 details verification.

Attestation: Upon completion, the service signs the result. Context is updated with cryptographically signed attestation proving the operation completed, maintaining Authenticated Context for downstream operations.

Services sign results; callers verify service signatures before accepting results—realizing bidirectional authentication analogous to mutual TLS. This stateless protocol ensures each hop is independently verified without implicit trust propagating through multi-step workflows.

4.2. Verification Flow

Figure 3 shows the verification pipeline—the heart of our integrity claim. The pipeline is always enforced and cannot be escaped: every invocation undergoes cryptographic verification before execution, with receivers independently verifying callers using public keys retrieved from the registry.

Policy Construction: Stage 3 constructs the effective policy independently at the receiver through dual-perspective enforcement. The PEP retrieves organizational policies (company, business unit, team) and resource-specific constraints, computing $P_{\text{eff}} = \text{Intent} \cap \text{Resource}$ (Section 3). The caller's claimed intent is independently verified against resource constraints—callers cannot influence policy evaluation. The intersection algebra (Theorems 1-3) guarantees monotonic restriction: composed policies only narrow permissions, never broaden them.

Custom Verifiers: MAPL's declarative constraints handle resource permissions and parameters, but some security checks require imperative logic on invocation context. Custom verifiers extend policy evaluation with PII detection, SQL injection prevention, prompt safety via LLM classifiers (LLMs as judges), rate limiting, and geolocation checks. Pre-invocation verifiers execute after policy evaluation; post-invocation verifiers process results. Operating on cryptographically verified context, verifiers are trusted code checked in by system administrators, not application developers. Once declared in the verification flow, the PEP enforces their execution—they cannot be bypassed. This provides domain-specific extensibility (HIPAA, GDPR, SOC 2) while maintaining integrity guarantees.

Four production verifiers demonstrate this architecture: (1) MemoryIntegrityVerifier prevents memory poisoning (OWASP LLM01) through rate limiting, protected field enforcement, and cryptographic integrity checks; (2) WorkflowIntegrityVerifier prevents workflow hijacking (OWASP LLM04) by enforcing prerequisite steps and maintaining attestation chains; (3) ToolAuthorizationVerifier prevents tool misuse (OWASP LLM06) via RBAC and dangerous pattern detection (e.g., file_write→command_execute); (4) StorageIntegrityVerifier prevents data exfiltration (OWASP LLM02) through path traversal prevention and encryption enforcement. Appendix A provides implementation details.

This separation is crucial: the core verification pipeline (Stages 1-3: signature verification, policy binding, policy evaluation) provides deterministic guarantees—100% recall with zero false positives through cryptographic enforcement of MAPL's declarative constraints (Lemmas 1, 4).

Custom verifiers (Stages 4-5) are optional administrative extensions that may use heuristics (ML-based PII detection, LLM safety classifiers) and could introduce false positives if misconfigured, but cannot weaken core guarantees—they can only add restrictions, never remove them. Verifiers execute only after cryptographic verification passes, operating on authenticated invocations and tamper-evident state, not unbounded application input. Even if a verifier fails or is compromised, Stages 1-3 still enforce cryptographic integrity and policy constraints. Bypassing verifiers requires corrupting trusted administrator code (L3), not manipulating application logic (A1).

Security Guarantees. Against adversaries A1-A5 (Section 2), the protocol mechanisms compose into defense-in-depth through four independent cryptographic layers: **Integrity** (signatures establish authenticity, defeating network attacks A4); **Authorization** (policy intersection enforces intent—operations must satisfy composed policies even with valid signatures, preventing privilege escalation under application control A1); **Provenance** (hash chains and sequence numbers make context tamper-evident, defeating content injection A2 and multi-turn manipulation A5); **Compliance** (custom verifiers add domain-specific validation on cryptographically verified context, providing non-repudiation). Together these achieve security objectives O1-O5.

The protocol provides defense-in-depth: each layer operates independently at each boundary (Lemma 7, Section 7). Compromising signature verification doesn't bypass policy evaluation; tampering with context doesn't defeat tool PEPs; bypassing one verifier doesn't invalidate cryptographic checks. Section 7 formalizes these independence guarantees.

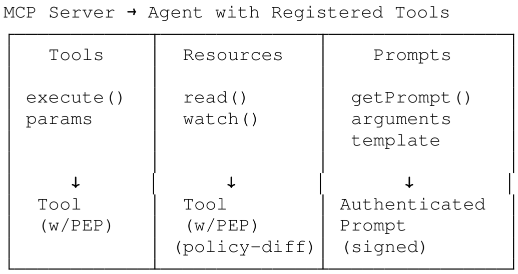
5. A Universal Security Abstraction

Authenticated workflows provide a universal security abstraction for agentic AI— We validate this claim by securing nine frameworks spanning three architectural layers: protocol standards (MCP, A2A), LLM APIs (OpenAI, Claude), and orchestration frameworks (LangChain, CrewAI, AutoGen, LlamaIndex, Haystack). Regardless of framework architecture, every agentic interaction maps completely to a verifiable boundary crossing governed by our four-phase protocol: Registration → Invocation → Verification → Attestation.

5.1. Framework Integration

Table 1 shows how nine frameworks map to authenticated workflow primitives. We highlight representative examples at each architectural layer:

Protocol Layer: MCP. MCP servers expose tools, resources and prompts. Resources capture static data-sets intended to give LLMs additional context to work with, while MCP tools are active functions with side effects, and parameters. Servers map to agents within our system, prompts map to Authenticated Prompts [3], and resources and tools are abstracted as verified tools guarded by different types of policies – each with a unique independent identity and key pairs.



```
MCP Server → Agent with Registered Tools
┌──────────────┬────────────────┬──────────────────┐
│   Tools      │  Resources     │    Prompts       │
│              │                │                  │
│ execute()    │  read()        │  getPrompt()     │
│ params       │  watch()       │  arguments       │
│              │                │  template        │
│      ↓       │      ↓         │       ↓          │
│  Tool        │  Tool          │ Authenticated    │
│  (w/PEP)     │  (w/PEP)       │ Prompt           │
│              │ (policy-diff)  │ (signed)         │
└──────────────┴────────────────┴──────────────────┘
```

Protocol Layer: A2A. A2A—the agent-to-agent protocol—maps directly to authenticated workflows. Agents register with cryptographic identity. Delegation becomes signed tokens with scope constraints following MAPL's intersection algebra—delegates cannot grant broader permissions than received. Attestations enable workflow dependencies through cryptographic proof, ensuring operations complete in required order.

LLM Interfaces: OpenAI and Claude. OpenAI and Claude follow identical patterns. API endpoints register as agents. LLM inference operations (generate, complete, embed) register as tools with embedded PEPs. Two deployment variants provide flexibility: *two-sided wrapping* treats both the LLM API and client application as agents—the API-side PEP verifies incoming requests satisfy provider policies, while the client-side PEP verifies function calls satisfy application policies; *one-sided wrapping* wraps only client-side tools, treating the LLM API as passthrough, simplifying deployment when the LLM provider is trusted for prompt security and the focus is protecting client side tool calling. This addresses the challenge that LLMs are untrusted for authorization decisions—verification must happen at function execution boundaries (S2) where PEPs independently verify signatures and evaluate policies.

Orchestration Layer: LangChain. Orchestration frameworks coordinate multi-step workflows across tools and LLMs, exposing multiple control points: chain composition logic, agent reasoning, memory operations, inter-agent communication. Securing only individual tools would leave composition and state management unprotected.

Our design addresses security at four layers. Agents register with cryptographic identity. Tools register independently with embedded PEPs—each tool invocation undergoes signature verification and policy evaluation. Chains—sequential compositions of operations—enforce policy composition through intersection (Section 3); if a tool policy denies operations, composing that tool into a chain cannot relax the restriction—effective policy becomes more restrictive through intersection, never more permissive. Memory—persistent state across interactions—enforces context integrity through authenticated context [3], preventing context poisoning (S4).

Framework	Layer	Mapping to Authenticated Workflow Primitives
MCP	Protocol	Server→Agent, Tools→Tools (w/PEP), Resources→Tools (policy-differentiated), Prompts→AuthenticatedPrompt
A2A	Protocol	Agents→Agents (cryptographic identity), Delegation→Signed tokens (scope constraints), Attestations→Attestations (workflow dependencies)
OpenAI	LLM API	Endpoint→Agent, Operations (generate/complete/embed)→Tools (w/PEP), Two deployment variants (two-sided/one-sided wrapping)
Claude	LLM API	Endpoint→Agent, Operations (generate/complete/embed)→Tools (w/PEP), Two deployment variants (two-sided/one-sided wrapping)
LangChain	Orchestration	Agents→Agents, Tools→Tools (w/PEP), Chains→Policy intersection, Memory→AuthenticatedContext (hash chains prevent poisoning)
CrewAI	Orchestration	Crew members→Agents, Roles→Attestations (cryptographically bound), Tasks→Signed invocations (role-based authorization)
AutoGen	Orchestration	Conversation agents→Agents, Code execution→Tool (w/PEP), Policies→Pluggable verifiers (AST analysis, allowlists, sandbox constraints)
LlamaIndex	Orchestration	RAG pipeline→Agent, Query/Retrieve operations→Tools (w/PEP enforcing document access control to prevent prompt injection via retrieval)
Haystack	Orchestration	Document pipeline→Agent, Pipeline nodes→Tools (w/PEP), Sequential execution→Policy intersection (nodes cannot relax constraints)

TABLE 1. FRAMEWORK MAPPING TO AUTHENTICATED WORKFLOW PRIMITIVES.

LangChain Orchestrator (Agent with Tools)

Agents	Tools	Chains	Memory
Coordinate workflow	Individual operations	Sequential compose	Persistent state
↓	↓	↓	↓
Agent	Tool (w/PEP)	Policy ∩	Authentic-ated Context
Each agent has own identity + policy	Each tool verifies signed invoc. before execution	Effective policy = ∩ of all tool policies	Context integrity verified every transition

Other orchestration frameworks (CrewAI, AutoGen, LlamaIndex, Haystack) follow similar patterns—roles map to attestations for cryptographic role binding, code execution uses pluggable verifiers for AST analysis, RAG pipelines enforce document access control as detailed in Appendix A

5.2. Protocol-Level Universality

At the protocol level, all nine frameworks collapse to a uniform abstraction: agents invoke operations through signed invocations; PEPs verify signatures and policies; operations execute or are rejected. This uniformity enables consistent security across heterogeneous compositions—LangChain orchestrating OpenAI invoking MCP tools—with independent verification at each boundary.

When workflows span frameworks, compromising one framework's verification does not bypass others. Each boundary's PEP independently verifies: (1) signature validity (authenticity); (2) identity (preventing impersonation); (3) policy evaluation of resource permissions, parameter constraints, and attestations (authorization). This compositional approach—securing boundaries rather than frameworks—enables uniform protection across heterogeneous systems. Section 7 formalizes these guarantees through Lemmas 6-7.

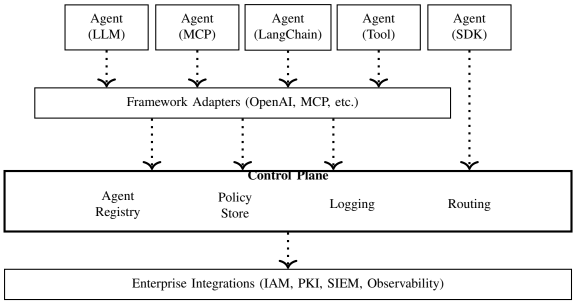


Figure 4. Agentic AI Trust Layer.

6. The Agentic AI Trust Layer

In this section we describe an Agentic AI Trust Layer, which includes authenticated workflows and authenticated prompts and context as described in [3]. All nine frameworks map onto the trust layer using thin framework adapters. The trust layer comprises a control plane providing core services (identity, policy, logging, routing), manages integrations with enterprise infrastructure (IAM, PKI, storage, etc), a verification gateway (sidecar PEP) and client libraries that embed PEPs within agentic applications. Due to space we focus our discussion on key design elements instead of an exhaustive overview.

6.1. Architecture

Figure 4 shows the trust layer architecture with agents interfacing with the control plane.

Framework Adapters. Each framework integrates through thin adapters that translate framework-native operations into authenticated workflows. Adapters implement the four-phase protocol—registration, invocation signing, verification, and attestation—while presenting the framework's native API. Integration is transparent; developers use standard framework APIs (MCP servers, OpenAI clients, LangChain agents) while cryptographic enforcement happens automatically.

Control Plane Services. The control plane provides four core services:

- *Agent Registry*: Stores agent identifiers and public keys, and tool identifiers and public keys within agents, preserving cryptographic lineage. Enables instant credential revocation. PEPs retrieve public keys during signature verification.
- *Policy Store*: Tamper-proof policy store where PEPs retrieve and compose hierarchical policies (e.g., company, business unit, team) through intersection.
- *Routing*: Routes signed invocations between agents across network boundaries, enabling callees to access callers' public keys for verification.
- *Logging*: Records all security events with cryptographic lineage through tamper-evident hash chains. Integrates with enterprise SIEM platforms. Blocked operations generate structured logs indicating rejection reason (signature invalid, policy violation, missing attestation, verifier failure) with sufficient detail for debugging while avoiding policy leakage. Audit logs enable root-cause analysis and compliance reporting.

Distributed Enforcement. These services run in highly available configurations, integrating with enterprise infrastructure (IAM, PKI, SIEM, observability platforms). The control plane is multi-tenant—organizations deploy shared infrastructure while maintaining cryptographic isolation between tenants. The architecture realizes zero-trust distributed verification—each PEP independently verifies invocations using public keys from the registry and policies from the store. PEPs are embedded in-process by default (linked library) or deployed as sidecars, providing horizontal scalability. With embedded PEPs, no centralized bottleneck exists; verification happens locally at each boundary with sub-millisecond overhead. Administrators can instantly revoke credentials, update policies, or isolate agents through the control plane. Due to space constraints, full architectural details are out of scope; we refer interested readers to our technical report for complete implementation details.

6.2. Design Decisions

Realizing authenticated workflows and our trust layer in production exposed several practical challenges that needed an opinionated design—we discuss six of the most interesting design decisions addressing these challenges.

Identity Bridging. Enterprise identities need bridging across frameworks—each has its own authentication model (OAuth, API keys, SAML). Instead of solving this per-framework, we bridge at the protocol level. When users authenticate, the trust layer issues ephemeral keypairs bound to their sessions and propagates cryptographically-verified principals across all frameworks. Signed invocations carry user identity through delegation chains; each PEP independently verifies the originating principal from the Agent Registry. This centralizes IAM integration—frameworks don't individually integrate with corporate identity systems; the trust layer handles authentication once. When OAuth-authenticated Alice invokes API-key-authenticated Claude which calls MCP-authenticated tools, logs show "Alice via Claude accessed file" preserving audit trails. Protocol-level bridging provides a simpler programming model with consistent principal propagation, eliminating per-framework integration complexity.

Session Management. Many frameworks lack native session concepts (MCP, LangChain are stateless), yet cross-turn correlation is essential for security. Instead of modifying each framework to add sessions, we track at the protocol level using context identifiers and sequence numbers transparent to frameworks. The trust layer maintains authenticated context regardless of framework support—frameworks without sessions receive invisible protocol-level tracking; frameworks with sessions map directly. Multi-tenant architecture ensures cryptographically isolated contexts per session, preventing cross-session contamination. This enables unified audit trails across multi-framework workflows without requiring protocol modifications to individual frameworks.

Verifiable Delegation. Multi-hop delegation creates privilege escalation risks—compromised intermediate agents could claim broader permissions than granted. Instead of trust-based delegation, we enforce cryptographic scope narrowing. Signed delegation tokens contain delegator identity, delegate identity, scope constraints, and expiration. The trust layer enforces scope narrowing via intersection: delegated scope = what delegator grants ∩ what delegate possesses, extending MAPL's algebra (Section 3). Compromised agents cannot claim broader permissions than granted; downstream services cryptographically verify delegation validity. This provides provable least privilege across heterogeneous frameworks.

Forward Secrecy. Long-lived API keys create operational and security risks—keys rarely rotate, and a single leak compromises all sessions. Instead of relying on operational key rotation policies, we use ephemeral keys by design. Each session receives unique ephemeral keypairs that expire automatically. Key rotation happens naturally via session lifecycle without infrastructure modifications, providing session-level forward secrecy. Compromising one session's key doesn't affect other sessions, limiting blast radius without operational overhead.

Service Accounts at Scale. Agentic systems spawn entities dynamically at runtime—orchestrators spawn agents, agents spawn tools, workflows create sub-workflows. Traditional IAM's manual provisioning doesn't scale to runtime dynamics. Instead of manual provisioning, we automatically provision cryptographic identities during registration (Section 4) at tool-level granularity—assigning (tool_id, keypair) and recording in the Agent Registry. No manual provisioning, no credential distribution, no IAM bottleneck. Each tool maintains independent cryptographic accountability, minimizing blast radius. This enables dynamic agent deployment at enterprise scale.

Bidirectional Accountability. Unidirectional authentication creates accountability gaps in agentic environments—callers cannot prove services performed operations correctly; services cannot prove callers requested operations. Instead of one-way signatures, we use dual signatures for mutual non-repudiation. Callers sign requests; services sign results. Each operation produces cryptographic proof from both parties, providing compliance-grade audit trails—services cannot repudiate actions; callers cannot repudiate requests. Combined with identity bridging and tamper-evident logging, this delivers complete accountability across heterogeneous authentication boundaries.

7. Formal Analysis

We prove authenticated workflows achieve O1-O5 against adversaries A1-A5 under assumptions L1-L3 via seven lemmas organized around intent and integrity. Integrity properties (Lemmas 1, 2, 3) ensure operations are authentic and tamper-evident. Intent properties (Lemmas 4, 5) ensure operations satisfy policies and compose safely. Completeness properties (Lemmas 6, 7) ensure all operations are verified and verification is independent.

Main Security Theorem. Theorem (Authenticated Workflows Security): Under trust assumptions L1-L3, authenticated workflows achieve integrity (O1), policy enforcement (O2), privilege non-escalation (O3), context integrity (O4), and accountability (O5) against adversaries with capabilities A1-A5.

Integrity Properties. These lemmas ensure operations are authentic, tamper-evident, and non-repudiable—addressing adversaries with component compromise (A3) and network attack (A4) capabilities.

Lemma 1 (Authenticity): All accepted invocations have valid cryptographic signatures binding the principal to the operation, arguments, policy, and context state.

Proof: Policy Enforcement Points verify signatures using principals' public keys retrieved from the Agent Registry (Section 5). Under cryptographic hardness assumption L1, forging signatures without the private key is computationally infeasible—even adversaries with application control (A1), network access (A4), or compromised components (A3) cannot produce valid signatures for other principals' identities. This achieves integrity (O1): operations are cryptographically authentic. □

Lemma 2 (Tamper Evidence): Modifications to context state or audit logs are cryptographically detectable, and replay attacks are prevented.

Proof: Context state is maintained through hash chains linking sequential states: $h_i = \text{Hash}(h_{i-1} \parallel \text{state}_i \parallel \text{sequence}_i)$. Each invocation includes a monotonically increasing sequence number bound in the signature. Modifying any state requires finding hash collisions; replaying old invocations is detected because PEPs reject sequence numbers less than or equal to previously processed values. Under cryptographic hardness (L1), finding collisions is computationally infeasible. Multi-turn manipulation attacks (A5) attempting gradual state poisoning or replayed operations are detected because tampering breaks the hash chain or violates sequence monotonicity. Audit logs use the same hash chain mechanism. This achieves context integrity (O4). □

Lemma 3 (Non-Repudiation): Principals cannot deny performing operations recorded in audit logs with valid signatures.

Proof: Only the principal possessing the private key can generate valid signatures (Lemma 1). Audit logs maintain tamper-evident records (Lemma 2). Therefore, signed operations in audit logs constitute cryptographic proof of principal actions. This achieves accountability (O5). □

Intent Properties. These lemmas ensure operations satisfy policies and compose safely—addressing adversaries with application control (A1), content injection (A2), and gradual attack (A5) capabilities.

Lemma 4 (Policy Enforcement): All executed operations satisfy the effective policy composed from organizational hierarchy and resource constraints, with the policy identifier cryptographically bound in the signature.

Proof: Each invocation includes a policy identifier in the signed payload. The PEP retrieves policies and computes the effective policy P_{eff} through intersection of all applicable policies—organizational (base, department, team) and resource-specific (Section 3). Adversaries attempting policy substitution attacks (capabilities A1, A4) fail because modifying the policy identifier invalidates the signature (Lemma 1). Policy intersection ensures monotonic restriction (Section 3, Theorems 1-3)—composed policies can only become more restrictive, never more permissive. This achieves policy enforcement (O2) and privilege non-escalation (O3). □

Lemma 5 (Composition Safety): When workflows span multiple frameworks, security properties are preserved across framework boundaries.

Proof: Consider operation O1 on framework F1 invoking operation O2 on framework F2. Both invocations carry signatures binding principals and policies (Lemma 1). Framework F2's PEP independently verifies O2's signature and computes effective policy as the intersection of F1's policy and F2's policy (Lemma 4). Framework F1 cannot bypass F2's verification because: (a) F2's PEP verifies signatures independently (Lemma 7, proven below), and (b) policy intersection ensures O2 must satisfy both F1 and F2 constraints. Composition across N frameworks creates N independent verification points with composed policy $P_{\text{eff}} = P_1 \cap P_2 \cap \ldots \cap P_N$, achieving monotonic restriction. Section 5 demonstrates this across nine framework combinations. □

Completeness Properties. These lemmas ensure all operations are verified and verification is independent across control surfaces.

Lemma 6 (Surface Completeness and Minimality): The four control surfaces {S1, S2, S3, S4} are complete (all resource access operations cross at least one surface) and minimal (each surface is necessary).

Proof by Enumeration and Necessity:

Completeness: We enumerate all resource access operations and show each crosses at least one surface: (1) *Computational Resources*—LLM inference crosses S1 (prompts carry instructions); tool execution crosses S2 (privileged operations). (2) *Data Resources*—RAG retrieval, database queries, web scraping cross S3 (external data flows into reasoning)—exploited by content injection attacks (A2). (3) *State Resources*—Session state, workflow context, memory cross S4 (persistent state across turns)—targeted by multi-turn manipulation (A5). (4) *Cross-Agent Communication*—Delegation crosses S1 (instruction propagation) or S2 (invocations) plus S4 (context inheritance). Operations not accessing external resources (pure computation) require no protection—they cannot exfiltrate data or violate policies.

Minimality: Each surface is necessary—removing any surface leaves attacks unprotected: Remove S1 (Prompts): indirect prompt injection via S3 data→LLM bypasses verification. Remove S2 (Tools): unauthorized tool execution despite prompt verification. Remove S3 (Data): poisoned RAG data→tool invocations despite S1/S2 verification. Remove S4 (Context): session hijacking and context poisoning across multi-turn workflows. Therefore, $\{S1, S2, S3, S4\}$ is complete and minimal. Section 5 empirically validates this across nine frameworks spanning three architectural layers. □

Lemma 7 (PEP Independence): Compromising one Policy Enforcement Point does not weaken verification at other PEPs.

Proof: Each PEP verifies operations independently using only: (1) cryptographic primitives for signature and hash chain validation (L1), (2) public keys and policies retrieved from the trusted control plane (L2), and (3) its own verification logic (L3). PEPs share no runtime state and perform no coordination. An adversary compromising PEP at one surface (capability A3) can only bypass verification at that surface—other surfaces perform independent verification using their own PEP instances. Multi-surface attacks require compromising multiple independent PEPs, each protected by L3. This realizes defense in depth: blast radius is limited to the compromised PEP's surface. □

Main Theorem Proof. *Proof of Main Theorem*: By Lemma 6, all resource access crosses at least one control surface. Each surface has an embedded PEP enforcing independent verification (Lemmas 1-4, 7). Multi-framework workflows compose verification across framework boundaries (Lemma 5). Verification enforces intent via policy evaluation (Lemma 4) and integrity via signatures (Lemma 1) and hash chains (Lemma 2). Attestations provide workflow dependencies through signed, hash-chained claims.

Every resource access operation has one of two outcomes: (1) *Authorized Execution*—operation satisfies all cryptographic checks (valid signature, tamper-free context, valid sequence number) and policy constraints (allowed resource, parameters within bounds, required attestations present), executes, and is logged with non-repudiable audit trail (Lemma 3); (2) *Blocked*—operation fails at least one check (invalid signature, policy violation, tampered context, sequence violation, or missing attestation), rejected before execution with violation logged.

Concrete Example: In the Q4 attack (Section 2), poisoned data instructs the LLM to exfiltrate credentials. When the LLM generates a filesystem read for `credentials.db`, the tool's PEP evaluates policy—which denies paths matching `*credential*`—blocking the operation despite valid signature from the authenticated LLM. Policy enforcement prevents unauthorized access regardless of signature validity. Even if multiple boundaries are compromised, each surface enforces independent verification (Lemma 7), limiting blast radius.

Therefore, under L1-L3, adversaries with capabilities A1-A5 cannot perform unauthorized resource access without detection, achieving O1-O5. □

Practical Implications. These formal properties enable the operational solutions in Section 6: Lemma 1 enables principal propagation, Lemmas 4+5 enable verifiable delegation, Lemma 2 enables session-level forward secrecy, and Lemma 3 enables bidirectional accountability.

8. Attack-Defense Validation

Section 7 proved authenticated workflows achieve O1-O5 against adversaries A1-A5 under assumptions L1-L3. We validate these formal guarantees through OWASP Top 10 coverage (9 of 10 risks), systematic attack testing (11 patterns, 174 test cases, 100% recall, 0% false positives), and production CVE analysis demonstrating protection where baseline systems failed.

OWASP Top 10 for LLM Applications 2025 Coverage. Table 2 maps our defenses to the OWASP Top 10 for LLM Applications 2025 [7], demonstrating systematic coverage with explicit defense mechanisms and formal guarantees.

These defenses address all adversary capabilities (Section 2): by-policy mechanisms block A1, A2, A5 through runtime enforcement (Lemmas 4, 5); by-design mechanisms eliminate A3, A4 through cryptographic hardness (Lemmas 1, 2, 3). Together, they achieve O1-O5. This validates the dual framework: neither mechanism alone suffices; both together provide complete protection.

Systematic Attack Validation. Beyond OWASP's broad risk categories, we analyze attack mechanisms—how attacks manifest in workflow composition. We identify 8 mechanism categories spanning prompt manipulation, tool chaining, credential theft, data exfiltration, malware/code execution, resource exhaustion, multi-agent attacks, and policy violations. Detailed taxonomy appears in Appendix C. Table 3 shows 11 attack patterns providing representative coverage.

Each pattern includes multiple test scenarios: Data Exfiltration tests bulk export, multi-resource access, PII extraction, unauthorized access; Inference Attack tests attribute, membership, reconstruction attacks. Our 174 test cases span framework combinations (OpenAI, Anthropic, LangChain, etc) and configuration variants.

Table 3 shows defense classification. By-design mechanisms realize integrity properties (Lemmas 1, 2, 3), rendering attacks cryptographically impossible: identity spoofing, session replay, policy substitution, context tampering, audit manipulation, attestation forgery. By-policy mechanisms realize intent properties (Lemmas 4, 5), blocking violations through runtime verification: prompt injection, privilege escalation, credential exfiltration, data harvesting, supply chain, resource exhaustion, cross-agent attacks. Complex

TABLE 2. OWASP TOP 10 FOR LLM APPLICATIONS 2025 COVERAGE

OWASP Risk	Our Attack Tests	Defense Mechanism	Section
LLM01: Prompt Injection	Prompt Injection, Atlas	By-policy: Policy blocks unauthorized operations + MemoryIntegrityVerifier prevents goal manipulation	§4, §7
LLM02: Info Disclosure	Token Hijack, Data Exfil, Side Channel, Inference	By-policy: Resource denial + StorageIntegrityVerifier detects exfiltration patterns	§4, §5
LLM03: Supply Chain	Rogue Tool, Supply Chain	By-design + By-policy: Code signing + tool registry with approval policies	§5, §7
LLM04: Data Poisoning	Context tamper	By-design: Hash chains + WorkflowIntegrityVerifier enforces step sequences	§4, §7
LLM05: Output Handling	(Cross-cutting)	By-policy: Post-invocation verifiers sanitize outputs	§5
LLM06: Excessive Agency	Keys to Kingdom, Confused Deputy	By-policy: Policy composition + ToolAuthorizationVerifier enforces RBAC with pattern detection	§3, §7
LLM07: Prompt Leakage	Prompt variants	By-design: Cryptographic segmentation (AuthPrompt)	§4
LLM08: Vector Weak.	(RAG scenarios)	By-policy: Attestations verify sources; policies restrict access	§3, §5
LLM09: Misinform.	(Tangential)	Pluggable verifiers; not primary security focus	§5
LLM10: Unbounded Cons.	Denial of Service	By-policy: Rate limiting + resource quotas	§5

Coverage: 9 of 10 OWASP risks. LLM04 partial (runtime poisoning prevented; training data out of scope). LLM09 tangential (quality vs. security).

TABLE 3. VALIDATED ATTACK PATTERNS AND DEFENSE CLASSIFICATION

Attack Pattern	Category	Defense Class	OWASP
Prompt Injection	Prompt Manipulation	By-Policy	LLM01
Keys to Kingdom	Tool Chaining	By-Policy	LLM06
Confused Deputy	Tool Chaining	By-Policy	LLM06
Token Hijacking	Credential Theft	By-Policy	LLM02, LLM06
Session Fixation	Credential Theft	By-Design	Beyond OWASP
Data Exfiltration	Data Exfiltration	By-Policy	LLM02
Side Channel	Data Exfiltration	By-Policy	LLM02
Inference Attack	Data Exfiltration	By-Policy	LLM02
Rogue Tool	Malware/Code Exec	By-Policy	LLM03
Supply Chain	Malware/Code Exec	By-Policy	LLM03
Denial of Service	Resource Exhaustion	By-Policy	LLM10
Total: 11 patterns, 18 explicit variants, 174 test cases including framework permutations			

Coverage: 6 of 8 mechanism categories explicitly implemented; Multi-Agent and Policy Violation attacks addressed through compositional mechanisms (see text).

attacks like supply chain require both—code signing plus tool approval policies. Six categories receive explicit implementations; Multi-Agent Attacks are prevented through independent PEP verification (Lemma 7).

Empirical Results: Our evaluation achieves 100% recall with zero false positives across 174 test cases. By-design elimination provides deterministic guarantees—violations are cryptographically detected, not pattern-matched. Breaking verification requires solving computationally hard problems (L1) rather than crafting adversarial inputs, explaining why attacks bypassing semantic defenses (Atlas, GitHub MCP CVEs) are deterministically blocked by authenticated workflows.

Real-World Attack Validation. We validate authenticated workflows against two attacks that compromised production systems—OpenAI's Atlas agentic browser and GitHub's Model Context Protocol server. Both attacks succeeded against baseline defenses using semantic guardrails; both are completely blocked by authenticated workflows through independent verification at control surfaces. We then demonstrate deployment feasibility through enterprise integration scenarios.

OpenAI Atlas Browser Attack. OpenAI's Atlas agentic browser [1] was compromised through prompt injection causing credential exfiltration (OWASP LLM01, LLM06, LLM02).

We emulate this attack: a legitimate workflow reads financial reports; malicious input embeds hidden instructions to leak credentials. The attack succeeds against semantic guardrails which cannot distinguish adversarial prompts from legitimate document content.

Policy enforcement blocks this cascade. Filesystem policies deny credential paths; network policies restrict external endpoints. PEP verification (Lemma 4) rejects unauthorized operations before execution regardless of LLM reasoning. We tested allowlist, denylist, combinations across framework variants. All blocked the attack with zero false positives.

GitHub MCP Prompt Injection. GitHub's Model Context Protocol server [8] (558,846+ downloads) was compromised through malicious prompts in GitHub issues invoking MCP filesystem tools for credential exfiltration (OWASP LLM01, LLM03, LLM02).

Multiple barriers block this cascade. First, tool authorization requires cryptographic signatures from approved registries—malicious tools rejected at discovery (Lemma 1). Second, filesystem policies block credential access (Lemma 4). Third, cross-framework composition ensures filesystem PEP verifies operations independently (Lemma 5). The attack is blocked through independent verification at multiple surfaces (Lemma 7).

Both attacks demonstrate that application-layer defenses alone are insufficient—production systems were compromised despite semantic guardrails. Authenticated workflows achieve complete protection through defense in depth: multiple independent cryptographic barriers must all be defeated simultaneously. All test scenarios maintained 100% recall with zero false positives, validating that formal guarantees

(Section 7) translate to production deployments. Section 10 positions these results against existing defenses.

9. Performance

We validate practical overhead through microbenchmarks on commodity hardware (8-core, 16GB RAM) using ECDSA-256 signatures and SHA-256 hashing. **Cryptographic operations** typically under ∼0.2ms overhead (ECDSA signature generation $\sim 102\mu$s, verification $\sim 89\mu$s, hash chain updates $<15\mu$s per operation). **Policy operations**: retrieval latency depends on backend: in-memory ($\sim$5-50μs), filesystem ($\sim$30-200μs), Redis ($\sim$50-300μs); policy intersection depends on composition—with typical 3-5 deep hierarchy remains under 100μs. **Custom verifiers**: deterministic checks (PII detection, path validation) add 30-400μs; LLM-based verifiers (prompt safety) add 150-500ms, dominating when enabled but providing semantic analysis orthogonal to cryptographic guarantees. **End-to-end impact**: For network-bound operations (LLM inference, remote APIs), cryptographic overhead (∼0.2ms) is negligible versus network latency (50-500ms). For local tool invocations, total PEP verification overhead remains under 1ms without LLM verifiers. Audit logging occurs asynchronously via buffered I/O.

A companion paper provides comprehensive system evaluation including production workload characterization, framework integration costs across all nine frameworks, large-scale deployment experience, and detailed performance analysis with production-scale benchmarks.

10. Related Work

Recent surveys [9]–[12] identify four critical gaps in agentic AI security: unpredictable multi-step workflows, cross-framework compositions, variable environments, and untrusted entity interactions. These surveys diagnose fragmentation but lack systems solutions. Prior work addresses either application semantics or infrastructure primitives; we provide a trust layer positioned between both—protocol-level primitives protecting four fundamental control surfaces (prompts, tools, data, context) where all agent-resource interactions occur.

Application-Layer Defenses Fail at Composition. Framework-specific security operates in isolation. LangChain provides schemas and validation [13]; MCP implements authentication and permissions [14]; OpenAI enforces rate limits and moderation [15]. Recent vulnerabilities demonstrate composition gaps: CVE-2024-8309 (LangChain prompt injection→SQL, CVSS 9.0) [16], CVE-2024-36480 (RCE via unsanitized eval()) [17], CVE-2025-6514 (MCP supply chain, 558K+ downloads) [8], Atlas browser attack (October 2025) [1], [2]. No cryptographic binding exists across framework boundaries.

Recent academic work addresses prompt injection defenses [18]–[20], tool use safety [21], and RAG retrieval poisoning [22], [23]—malicious instructions embedded in documents that compromise reasoning. Semantic defenses (guardrails [24]–[26]) achieve 60-80% detection but remain bypassable [27], [28]. We provide deterministic guarantees (100% recall, zero false positives) through defense in depth: by-design elimination renders attacks cryptographically infeasible; by-policy enforcement blocks violations. The approaches are complementary—semantic defenses reduce attack surface; cryptographic enforcement provides non-bypassable verification.

Infrastructure Primitives Lack Workflow Semantics. Infrastructure primitives provide strong guarantees but lack workflow semantics. Identity systems (SPIFFE [29], AWS IAM [30]) and policy engines (OPA [31], Cedar [32]) compose policies through intersection but cannot express temporal dependencies—"operation B requires proof that A completed." Cedar composes policies via intersection but lacks: (1) attestation-based workflow dependencies, (2) dynamic principals resolved at runtime, (3) cryptographic binding across four control surfaces. Without cryptographic proof of prerequisite operations, policies must trust application-reported state that attackers can forge.

MAPL addresses this through attestations—unforgeable cryptographic proofs that operations completed, enabling provably correct multi-step workflows. Unlike platform attestation (TPM [33], SGX [34]) proving code integrity through measurements, MAPL attestations prove *workflow completion*—cryptographic proofs that operations executed with specific results, enabling temporal dependencies impossible with measurement-based attestation. We extend authenticated system calls [5] and inline reference monitors [6], [35] from kernel boundaries to four control surfaces, adding stateful verification via authenticated context for workflow dependencies and pluggable verifiers for domain-specific checks (Appendix A).

Orthogonal Approaches. AI safety [36]–[38] operates at training-time; we provide runtime enforcement. Confidential computing [34], [39] protects data confidentiality; we address workflow integrity. These approaches are complementary.

11. Conclusion

We presented authenticated workflows—the first complete trust layer for enterprise agentic AI. Our key architectural insight: positioning at the protocol level—between application-layer defenses that fail at composition and infrastructure primitives that lack workflow semantics—enables uniform protection across heterogeneous frameworks while maintaining workflow context. All agent-resource interactions cross four fundamental control surfaces (prompts, tools, data, context); protecting these surfaces uniformly achieves compositional security impossible at either adjacent layer alone.

Three contributions realize this vision: MAPL provides cryptographic workflow dependencies via attestations, enabling sequential constraints without trusting application-reported state; embedded independent PEPs realize defense in depth where multiple cryptographic barriers must be defeated simultaneously; integration across nine frameworks through thin adapters validates surface completeness. Formal proofs (Lemmas 1-7, Theorems 1-3) establish security properties; empirical validation demonstrates 100% recall, zero false positives across 174 test cases, and protection against two production CVEs that compromised baseline systems. This addresses the five challenges posed in Section 1: cryptographic integrity across frameworks, policy enforcement at scale, privilege non-escalation through composition, context integrity via hash chains, and complete accountability through non-repudiation. As agentic AI transitions to production, authenticated workflows provide the security substrate enabling safe deployment where composition gaps currently block adoption.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was supported in part by [REDACTED FOR BLIND REVIEW].

References

[1] Security Researchers, "Prompt injection vulnerability in OpenAI Atlas browser," Security Advisory, October 2025, disclosed October 21, 2025.

[2] OpenAI, "Statement on prompt injection challenges from OpenAI CISO," Official Statement, October 2025.

[3] M. Rajagopalan and V. Rao, "Protecting context and prompts: Deterministic security for non-deterministic AI," in *Proceedings of the 1st International Workshop on AI Security and Evaluation (IASEAI'26), co-located with IEEE Symposium on Security and Privacy*. MACAW Security, Inc. and ROOST.tools, 2026, to appear.

[4] G. Li, H. A. A. K. Hammoud, H. Itani, D. Khizbullin, and B. Ghanem, "CAMEL: Communicative agents for mind exploration of large scale language model society," *arXiv preprint arXiv:2303.17760*, 2023.

[5] M. Rajagopalan, M. A. Hiltunen, T. Jim, and R. D. Schlichting, "Efficient and flexible security support for distributed component-based systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 4, pp. 352–369, 2006.

[6] Ú. Erlingsson and F. B. Schneider, "The inlined reference monitor approach to security policy enforcement," *ACM Transactions on Programming Languages and Systems*, vol. 28, no. 1, pp. 2–44, 2004.

[7] OWASP Foundation, "OWASP top 10 for LLM applications 2025," https://genai.owasp.org/, 2025.

[8] Invariant Labs, "CVE-2025-6514: Prompt injection in GitHub model context protocol server," https://www.invariant.sh/blog/cve-2025-6514, May 2025, disclosed May 26, 2025; 558,846+ downloads.

[9] Y. Wang, X. Chen, J. Li, and W. Zhang, "AI agents under threat: A survey of key security challenges and future pathways," *ACM Computing Surveys*, 2024, arXiv:2406.02630.

[10] Y. Zhou, T. Liu, and J. Wang, "A survey on agentic security: Applications, threats and defenses," *arXiv preprint arXiv:2510.06445*, 2025.

[11] N. Gong, Y. Liu, and Y. Zhang, "Formalizing and benchmarking prompt injection attacks and defenses," in *Proceedings of the 33rd USENIX Security Symposium*. USENIX, 2024.

[12] Y. Liu, X. Chen, and Z. Wang, "From prompt injections to protocol exploits: Threats in LLM-powered AI agents workflows," *arXiv preprint arXiv:2506.23260*, June 2025.

[13] L. Inc., "Langchain: Building applications with llms," https://www.langchain.com/, 2023.

[14] Anthropic, "Model context protocol (MCP)," https://modelcontextprotocol.io/, 2024.

[15] OpenAI, "OpenAI Assistants API," https://platform.openai.com/docs/assistants/, 2024.

[16] Huntr Dev, "CVE-2024-8309: Prompt injection in LangChain's GraphCypherQAChain leads to full database compromise," https://nvd.nist.gov/vuln/detail/CVE-2024-8309, 2024.

[17] NVD, "CVE-2024-36480: Remote code execution in LangChain," https://nvd.nist.gov/vuln/detail/CVE-2024-36480, 2024, cVSS 9.0 - Critical.

[18] F. Perez and I. Ribeiro, "Ignore previous prompt: Attack techniques for language models," *arXiv preprint arXiv:2211.09527*, 2022.

[19] Y. Liu, G. Deng, Z. Xu, Y. Li, Y. Zheng, Y. Zhang, L. Zhao, T. Zhang, and Y. Liu, "Prompt injection attack against LLM-integrated applications," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2023, pp. 1–15.

[20] N. Jain, A. Schwarzschild, Y. Wen, G. Somepalli, J. Kirchenbauer, P.-y. Chiang, M. Goldblum, A. Saha, J. Geiping, and T. Goldstein, "Baseline defenses for adversarial attacks against aligned language models," *arXiv preprint arXiv:2309.00614*, 2023.

[21] Y. Ruan, H. Dong, A. Wang, S. Pitis, Y. Zhou, J. Ba, Y. Dubois, C. J. Maddison, and T. Hashimoto, "Identifying the risks of LM agents with an LM-emulated sandbox," *arXiv preprint arXiv:2309.15817*, 2023.

[22] W. Zou, R. Geng, B. Wang, and J. Jia, "PoisonedRAG: Knowledge poisoning attacks to retrieval-augmented generation of large language models," in *arXiv preprint arXiv:2402.07867*, 2024.

[23] Z. Cheng, G. Li, J. Liu, Q. Zhang, and J. Shao, "PARDEN: Toward detecting backdoor attacks in retrieval-augmented generation," *arXiv preprint arXiv:2409.01573*, 2024.

[24] NVIDIA, "NeMo Guardrails: Programmable guardrails for LLM applications," https://github.com/NVIDIA/NeMo-Guardrails, 2023.

[25] Rebuff.ai, "Rebuff: Prompt injection detection," https://github.com/protectai/rebuff, 2023.

[26] H. Inan, K. Upasani, J. Chi, R. Rungta, K. Iyer, Y. Mao, M. Tontchev, Q. Hu, B. Fuller, D. Testuggine, and M. Khabsa, "Llama Guard: LLM-based input-output safeguard for human-ai conversations," 2023.

[27] A. Wei, N. Haghtalab, and J. Steinhardt, "Jailbroken: How does LLM safety training fail?" *Advances in Neural Information Processing Systems*, vol. 36, 2023.

[28] A. Zou, Z. Wang, J. Z. Kolter, and M. Fredrikson, "Universal and transferable adversarial attacks on aligned language models," *arXiv preprint arXiv:2307.15043*, 2023.

[29] CNCF, "SPIFFE: Secure production identity framework for everyone," https://spiffe.io/, 2023.

[30] Amazon Web Services, "AWS identity and access management: Policy composition and authorization," https://aws.amazon.com/iam/, 2023.

[31] CNCF, "Open policy agent: Policy-based control," https://www.openpolicyagent.org/, 2023.

[32] A. W. Services, "Cedar: Policy language for authorization," https://www.cedarpolicy.com/, 2023.

[33] Trusted Computing Group, "TPM 2.0 library specification," Trusted Computing Group, Tech. Rep., 2016, specification Family "2.0", Level 00 Revision 01.38.

[34] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptology ePrint Archive*, vol. 2016, p. 086, 2016.

[35] F. B. Schneider, "Enforceable security policies," *ACM Transactions on Information and System Security*, vol. 3, no. 1, pp. 30–50, 2000.

[36] Y. Bai, S. Kadavath, S. Kundu, A. Askell, J. Kernion, A. Jones, A. Chen, A. Goldie, A. Mirhoseini, C. McKinnon *et al.*, "Constitutional AI: Harmlessness from AI feedback," *arXiv preprint arXiv:2212.08073*, 2022.

[37] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, “Training language models to follow instructions with human feedback,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 27 730–27 744, 2022.

[38] D. Ganguli, L. Lovitt, J. Kernion, A. Askell, Y. Bai, S. Kadavath, B. Mann, E. Perez, N. Schiefer, K. Ndousse *et al.*, “Red teaming language models to reduce harms: Methods, scaling behaviors, and lessons learned,” *arXiv preprint arXiv:2209.07858*, 2022.

[39] C. Gentry, “A fully homomorphic encryption scheme,” Ph.D. dissertation, Stanford University, 2009.

Appendix

Appendix A: MAPL Policy Examples

Base Organizational Policy

```
{
  "policy_id": "acme:base",
  "resources": ["tool:*", "
      ↪ llm:*"],
  "denied_resources": ["*
      ↪ credential*"],
  "constraints": {
    "attestations": ["
        ↪ user_authenticated
        ↪ "]
  }
}
```

Baseline with credential blocks.

Department Policy (Finance)

```
{
  "policy_id": "acme:finance
      ↪ ",
  "extends": "acme:base",
  "resources": ["data:
      ↪ finance:**"],
  "constraints": {
    "attestations": ["
        ↪ mfa_verified"]
  }
}
```

Inherits base, adds MFA.

Attestation Chaining

```
// Step 1: Analysis
{"policy_id": "fin:analyze
     ↪ ",
 "resources": ["tool:analyze
     ↪ "],
 "constraints": {"
     ↪ attestations":
   ["user_authenticated"]}}
// -> Produces:
     ↪ analysis_done

// Step 2: Requires Step 1
{"policy_id": "fin:report",
 "resources": ["tool:report
     ↪ "],
 "constraints": {"
     ↪ attestations":
   ["analysis_done"]}}
// -> Produces: report_done

// Step 3: Requires Step 2
{"policy_id": "fin:send",
 "resources": ["tool:email
     ↪ "],
 "constraints": {"
     ↪ attestations":
   ["report_done"]}}
```

Cryptographic proof prevents workflow hijacking.

Resource Policy (Database)

```
{
  "policy_id": "database:
      ↪ customer_db",
  "constraints": {
    "denied_parameters": {
      "query": ["DROP", "
          ↪ DELETE"]
    }
  }
}
```

Dual-perspective constraints.

Delegation Policy

```
{
  "policy_id": "agent:
      ↪ sub_agent",
  "extends": "acme:finance",
  "resources": ["data:
      ↪ finance:reports
      ↪ :*"],
  "constraints": {
    "parameters": {"
        ↪ date_range":
      ["2024-Q1", "2024-Q2"]}
  }
}
```

Narrowed delegation scope.

Emergency Access

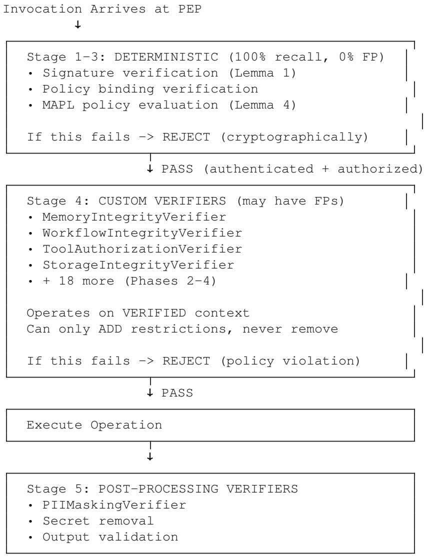
```
{
  "policy_id": "emergency:
      ↪ incident",
  "extends": "acme:base",
  "validity": {
    "not_after": "2024-10-25
        ↪ T16:00:00Z"
  },
  "constraints": {"
      ↪ attestations":
    ["ciso_approved"]}
}
```

Time-bounded (2-hour window).

Appendix B: Custom Verifier Implementation

Verification Pipeline Architecture

The PEP verification pipeline composes five stages:



```
Invocation Arrives at PEP
         ↓

  Stage 1-3: DETERMINISTIC (100% recall, 0% FP)
  • Signature verification (Lemma 1)
  • Policy binding verification
  • MAPL policy evaluation (Lemma 4)

  If this fails -> REJECT (cryptographically)

                  ↓ PASS (authenticated + authorized)

  Stage 4: CUSTOM VERIFIERS (may have FPs)
  • MemoryIntegrityVerifier
  • WorkflowIntegrityVerifier
  • ToolAuthorizationVerifier
  • StorageIntegrityVerifier
  • + 18 more (Phases 2-4)

  Operates on VERIFIED context
  Can only ADD restrictions, never remove

  If this fails -> REJECT (policy violation)

                  ↓ PASS

  Execute Operation

                  ↓

  Stage 5: POST-PROCESSING VERIFIERS
  • PIIMaskingVerifier
  • Secret removal
  • Output validation
```

Even if custom verifiers are compromised, Stages 1-3 enforce cryptographic integrity and policy constraints.

Production Verifiers

Four production verifiers address OWASP Top 10 threats through domain-specific checks on cryptographically verified context:

MemoryIntegrityVerifier
Prevents memory poisoning (OWASP LLM01)
- Rate limiting (10 updates/min)
- Protected fields (goals, system_prompt)
- Pattern detection (suspicious updates)
- Cryptographic integrity checks

WorkflowIntegrityVerifier
Prevents workflow hijacking (OWASP LLM04)
- Sequence enforcement (prerequisites)
- State tracking (prevent re-execution)
- Attestation chains (crypto proof)
- Duration limits (max 3600s)

ToolAuthorizationVerifier
Prevents tool misuse (OWASP LLM06)
- RBAC (role-to-tool mapping)
- Dangerous patterns (write→execute)
- Usage rate limiting (30 tools/min)
- Privilege escalation prevention

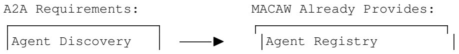
StorageIntegrityVerifier
Prevents data exfiltration (OWASP LLM02)
- Path traversal prevention
- Data classification (secret/sensitive)
- Encryption enforcement
- Exfiltration detection (bulk reads)

Appendix C: Framework Integration Details

Detailed mappings showing how framework constructs map to authenticated workflow primitives across nine frameworks.

A2A (Agent-to-Agent Protocol)

A2A defines standards for agent discovery, trust establishment, and communication. MACAW provides the cryptographic infrastructure A2A requires:



```
A2A Requirements:                 MACAW Already Provides:

Agent Discovery          ──▶     Agent Registry
```

```
                             - Agent directory
                             - Capability ads

Identity Verify     ──▶     MA (Crypto Identity)
                             - Public/private keys
                             - X.509 certificates

Trust Establish     ──▶     Attestation System
                             - Claims about agent
                             - Verifiable proofs

Secure Messaging    ──▶     CAW (Signed Invocation)
                             - Message signing
                             - Replay protection

Protocol Negotiat   ──▶     Capability Exchange
                             - Version negotiation
                             - Feature discovery
```

```
Complete A2A Flow:
Agent A        1. Discovery Request       Agent B
             ──────────────────────▶
               "Who can analyze data?"
   │                                        │
   ▼                                        ▼
                 Agent Registry
  Registry:
   - Agent B: capabilities["data_analysis"]
   - Agent B: public_key="..."
   - Agent B: attestations["certified_analyst"]
   │ 2. Returns Agent B info
   ▼
                3. Establish trust
  Agent A     ──────────────────────▶  Agent B
                - Verify B's signature
                - Check attestations
                - Validate policy

                4. Send CAW
              ──────────────────────▶
                - Message signed by A
                - A's attestations
                - Policy requirements
```

Security Challenge: Agents must verify identity and capabilities before delegating authority. Attestations provide cryptographic claims that enable trust decisions without centralized authorities.

MCP (Model Context Protocol)

MCP servers expose three capability types: tools (executable functions), resources (data access with read/write/watch operations), and prompts (retrievable templates). The MACAW mapping:

```
MCP Core Concepts:
                 MCP Server
  Tools       Resources          Prompts
 execute()   read/write()     getPrompt()
```

```
MACAW Mapping:
             macawAgent (MCP Server)
  Tool       Tool            Authenticated
  (execute   (Resource)      Prompt
   → CAW)    (read → CAW)    (signed template)
```

```
Resource Flow:
Client: "Read file.txt" → SecureMCP: invoke_tool()
  → PEP: signature valid? → Policy: allow read?
  → Execute if allowed
```

Security Challenge: Resources represent read-only data access, while tools can modify state. The mapping enforces read-only constraints through policy, preventing privilege escalation from data retrieval to data modification.

OpenAI and Claude (LLM Interfaces)

LLM providers expose inference APIs with function calling. Two deployment variants exist based on trust assumptions:

```
Two-sided Wrapping: Both LLM API and Client as macawAgents
             Application Code
                     │
                     ▼
          SecureOpenAI/SecureClaude
  LLM Wrapper               Tool Wrapper

 chat.complete()           search_tool()
      ↓                         ↓
  INTERCEPT                 INTERCEPT
      ↓                         ↓
  Sign as macawAgent        Sign invocation
      ↓                         ↓
  PEP verifies:             PEP verifies:
  - Signature               - Signature
  - Provider policy         - App policy
      ↓                         ↓
  Execute LLM call          Execute tool
```

```
One-sided Wrapping: Only Client Tools as macawAgents
             Application Code
                     │
                     ▼
          SecureOpenAI/SecureClaude
  LLM Passthrough           Tool Wrapper

 chat.complete()           search_tool()
      ↓                         ↓
  Direct to API             INTERCEPT
  (no interception)             ↓
                            Sign invocation
                                ↓
                            PEP verifies:
                            - Signature
                            - Policy
                                ↓
                            Execute tool
```

Security Challenge: LLMs are untrusted for authorization decisions. Even with prompt engineering, adversarial prompts can cause unauthorized function calls. PEPs verify at function execution boundaries regardless of LLM decisions.

LangChain (Multi-Agent Orchestration)

LangChain coordinates multi-step workflows across tools and LLMs. Multi-level integration addresses four layers:

```
LangChain Architecture:
         LangChain Orchestrator
              (macawAgent)
 Agents      Tools       Chains       Memory

Coordinate  Individual  Sequential  Persistent state
workflow    operations  compose     across turns

Agent →     Tool →      Chain →     Memory →
macawAgent   Tool        Policy     AuthenticatedCntx
             (w/PEP)     Intersect   (hash chains)

Each agent  Each tool   Effective   Context integrity
has own     verifies    policy =    verified every
identity    CAW         ∩ of all    state transition
and policy  before      tool
            execution   policies    Prevents context
                                    poisoning attacks
```

```
Workflow Flow:
User: "Analyze documents and send summary"
  ↓
LangChain Agent (macawAgent)
  ↓
Step 1: Read docs
  → Tool invocation → CAW → PEP check → Execute
  ↓
Step 2: Analyze content
  → LLM invocation → CAW → PEP check → Execute
  ↓
Step 3: Send email
  → Tool invocation → CAW → PEP check → Execute
```

```
Policy Composition via Intersection:
If tool_1 policy allows {A, B, C}
And tool_2 policy allows {B, C, D}
And chain composes tool_1 → tool_2
Then effective policy = {B, C} (intersection)
Chain composition cannot relax tool policies.
```

Security Challenge: Orchestrators expose multiple control points beyond individual tools—chain composition logic, agent reasoning, memory operations. Securing only tools would leave composition and state management unprotected.

CrewAI (Role-Based Multi-Agent)

CrewAI organizes agents into crews where each member has a role executing tasks collaboratively:

```
CrewAI Architecture:
                 Crew
             (macawAgent)
 Members     Roles      Tasks      Collaboration
```

Agents with specific roles	Role-based perms	Work items assigned to agents	Inter-agent communication
Member → macawAgent	Role → Attestation	Task → CAW	Message → CAW with role check
Writer cannot access DB tools	"writer" attested cryptographical	Execute task only if role matches	Writer cannot invoke DB tools reserved for Researcher role

```
Role-Based Flow:
Crew: Researcher + Writer
  ↓
Task: "Write report on Q4 data"
  ↓
Researcher (macawAgent with "researcher" attestation):
  → Invokes database_query tool
  → PEP checks: signature valid? role="researcher"?
  → Policy: "database_query" requires attestation["researcher"]
  → Allowed: execute query
  ↓
Writer (macawAgent with "writer" attestation):
  → Receives data from Researcher
  → Attempts database_query tool
  → PEP checks: signature valid? role="writer"?
  → Policy: "database_query" requires attestation["researcher"]
  → Denied: Writer lacks required attestation
```

Security Challenge: Role-based permissions must be cryptographically enforced. A "writer" role should not access databases reserved for "researchers," even if the LLM powering the writer attempts unauthorized invocations. Attestations bind roles cryptographically to agent identities.

AutoGen (Autonomous Code Generation)

AutoGen enables autonomous code generation and execution. The security challenge is unrestricted code execution:

AutoGen Architecture:

AutoGen Conversation (macawAgent)			
Agents	Code Gen	Exec	Validation
Conversational agents	Generate Python code to solve tasks	Execute generated code	Verify code before execution
Agent → macawAgent	LLM → Tool	Exec → Tool (w/PEP)	Verifiers: - AST analysis - Allowlist check - Sandbox verify
Each agent signs requests	Code gen becomes CAW with code	Execution request = CAW	Pre-invocation verifiers analyze code content before execution

```
Code Execution Flow:
User: "Calculate Fibonacci(10)"
  ↓
AutoGen Agent generates code:
  def fib(n): return n if n<2 else fib(n-1)+fib(n-2)
  ↓
Execution request → CAW
  ↓
PEP invokes pre-execution verifiers:
  1. AST analysis: detect dangerous patterns
     (os.system, subprocess, eval, exec, import sys)
  2. Allowlist check: only math/basic operations
  3. Sandbox verification: no network/file access
  ↓
Verifiers approve → Execute in sandbox → Return result
  ↓
Malicious attempt: "os.system('rm -rf /')"
  ↓
AST analysis detects os.system → DENY
```

Security Challenge: Unrestricted code execution enables arbitrary operations including data exfiltration and privilege escalation. Pluggable verifiers analyze code through AST parsing before execution, enforcing allowlists and sandbox constraints.

LlamaIndex (RAG Pipelines)

LlamaIndex provides RAG (Retrieval Augmented Generation) pipelines for document processing and question answering:

LlamaIndex Architecture:

LlamaIndex Pipeline (macawAgent)			
Index	Query Engine	Retrieve	Generate
Document indexing	Process queries	Vector search	LLM generates answer from retrieved docs
Index → macawAgent	Query → Tool (w/PEP)	Retrieve → Tool (w/PEP)	Generate → Tool (w/PEP)
Vector DB becomes macawAgent	Query validated against policy	Document access validated against policy	LLM receives only allowed documents

```
RAG Flow with Security:
User: "What were Q4 earnings?"
  ↓
Query Engine (Tool with PEP)
  → CAW: query="Q4 earnings"
  → PEP checks:
     - Signature valid?
     - Policy allows query pattern?
     - Denied patterns: *password*, *credentials*
  → Approved: continue
  ↓
Retriever (Tool with PEP)
  → CAW: retrieve docs matching "Q4 earnings"
  → PEP checks:
     - Can access these documents?
     - Allowed files: *public*, *reports*
     - Denied files: *private*, *secret*
  → Approved: retrieve documents
  ↓
Generator (Tool with PEP)
  → CAW: generate answer from retrieved docs
  → PEP checks: output validation
  → Return answer to user
```

Security Challenge: RAG pipelines access untrusted document stores. Malicious documents could contain prompt injection or sensitive data. PEPs verify document access patterns and query constraints before retrieval.

Haystack (Document Processing Pipelines)

Haystack provides document processing pipelines with nodes for retrieval, preprocessing, and generation:

Haystack Architecture:

Haystack Pipeline (macawAgent)			
Pipeline	Nodes	Retrieve	Process
Sequential execution of nodes	Pipeline steps	Document retrieval	Text processing and generation
Pipeline → macawAgent	Node → Tool (w/PEP)	Retrieve → Tool (w/PEP)	Process → Tool (w/PEP)
Each node in chain enforces policy	Each node verifies CAW	Document access controlled by policy	Processing validated against policy

```
Pipeline Flow:
User: "Process documents in ./reports/"
  ↓
Pipeline (macawAgent)
  ↓
Node 1: FileReader (Tool with PEP)
  → CAW: read("./reports/")
  → PEP checks:
     - Allowed paths: *reports*, *public*
     - Denied paths: *credentials*, *private*
  → Approved: read files
  ↓
Node 2: Preprocessor (Tool with PEP)
  → CAW: clean text, remove PII
  → PEP checks: PII detection verifier
  → Execute with redaction
  ↓
Node 3: Generator (Tool with PEP)
  → CAW: generate summary
  → PEP checks: output validation
  → Return result

Policy Composition Across Nodes:
If Node 1 allows {A, B, C}
And Node 2 allows {B, C, D}
And pipeline chains Node 1 → Node 2
Then effective policy = {B, C} (intersection)
```

Security Challenge: Document processing pipelines access file systems and external data sources. Nodes must enforce path constraints and content validation to prevent unauthorized access or data exfiltration.

Appendix D: AI Agent Threat Taxonomy

ID	**Attack**	**Description & Impact**
Category A: Prompt Injection & Input Manipulation		
T-A1	Hidden Prompt Injection	Embedded via white-on-white text, CSS, HTML comments. Impact: Command execution, credential theft.
T-A2	Semantic Obfuscation	Rephrasing to evade filters, base64 encoding. Impact: Filter bypass.
T-A3	Multi-Turn Poisoning	Gradual injection across turns, fabricated history. Impact: Privilege escalation.
T-A4	Delayed Activation	Conditional triggers (time-bombs). Impact: Evading immediate detection.
Category B: Tool Chaining & Derivation Exploits		
T-B1	Privilege Escalation via Chaining	Combining benign operations (search→list→read). Impact: Policy circumvention.
T-B2	Semantic Drift	Progressive deviation from intent. Impact: Compounding interpretation errors.
T-B3	Derivation Depth Exhaustion	Deep recursion chains. Impact: Resource exhaustion, policy drift.
Category C: Credential & Session Attacks		
T-C1	Token Extraction	Reading credentials from context. Impact: Session hijacking, cloud access.
T-C2	Credential Exfiltration	Tool abuse sequence (read→email). Impact: Persistent unauthorized access.
T-C3	OAuth Flow Manipulation	Tricking OAuth to attacker apps. Impact: Account takeover.
T-C4	Session Replay	Reusing valid signed prompts. Impact: Stale authentication exploitation.
Category D: Data Exfiltration & Privacy Breaches		
T-D1	Conversation History Mining	Extracting PII/secrets from history. Impact: GDPR/HIPAA violations, IP theft.
T-D2	Cross-Principal Leakage	Insufficient context isolation. Impact: Unauthorized cross-user access.
T-D3	Embedding Space Poisoning	Polluting vector DBs/RAG. Impact: Persistent data contamination.
T-D4	Document Harvesting	Systematic file extraction. Impact: Mass data theft.
Category E: Malware & Code Execution		
T-E1	AI-Invoked Malware	Downloads/executes malicious code. Impact: Ransomware, backdoors.
T-E2	Code Generation Exploitation	Generates disguised malicious scripts. Impact: Resource hijacking.
T-E3	Persistence Installation	Creating backdoors (.bashrc, cron). Impact: Persistent compromise.
T-E4	Supply Chain via Packages	Installing trojanized dependencies. Impact: Dependency confusion.
Category F: Resource Exhaustion & Cost Attacks		
T-F1	Computational Exhaustion	Recursive ops, infinite loops. Impact: DoS, billing inflation.
T-F2	API Rate Limit Exhaustion	Depleting quotas via loops. Impact: Service degradation, $1000s+ costs.
T-F3	Storage/Bandwidth Saturation	Unbounded downloads/generation. Impact: System failure.
Category G: Multi-Agent System Attacks		
T-G1	Agent-to-Agent Infection	Poisoned prompts via MCP/A2A. Impact: Cascading compromise.
T-G2	Workflow Hijacking	Crafted inter-agent requests. Impact: Distributed policy bypass.
T-G3	Byzantine Agent Behavior	Malicious agent with valid credentials. Impact: Authorized attacks.
T-G4	Trust Anchor Compromise	Corrupting registry/identity provider. Impact: Complete compromise.
Category H: Policy & Compliance Violations		
T-H1	Policy Confusion	Exploiting rule ambiguities. Impact: Authorization gaps.
T-H2	Audit Trail Manipulation	Corrupting logs, false entries. Impact: Loss of forensic evidence.
T-H3	Compliance Bypass	GDPR/HIPAA/SOC2 violations. Impact: Legal liability, 4% revenue fines.
T-H4	Attestation Forgery	Fake security check proofs. Impact: Workflow security bypass.

Comprehensive taxonomy of 25 attack variants across eight categories (T-A through T-H). Section 8 validates coverage.