MACAW SECURITY

Research Preview — Advanced Distribution

The External MCP Problem:

Inline Gateway Pattern for Agentic Security

March 2026

Confidential

Executive Summary

External MCP servers are becoming load-bearing enterprise infrastructure. Salesforce, Snowflake, Databricks, Slack, GitHub, and dozens of other platforms now expose MCP endpoints, and production agentic workflows are calling them — often across multiple hops, on behalf of real users, with real data access. The security posture across this ecosystem is, today, largely a PAT. An OAuth session. Credentials that grant access for a duration. When one is compromised, everything it covers is exposed. There is no per-invocation proof of authorization. There is no cryptographic record of user intent.

When organizations recognize this gap, the first instinct is a gateway: a centralized enforcement point between agents and external services. Gateways are a reasonable response to a real problem. We built one. We then built a sidecar. Then an inline gateway. Each time, the security model was identical — the topology changed. What we found: the further you move enforcement toward the calling application, the better the security properties, the lower the latency, and the more expressive the policy. The inline gateway pattern is not a refinement of the gateway approach. It is architecturally superior, and provably so.

What makes this possible is a deeper architectural observation: in MACAW's model, LLM APIs and MCP servers are semantically equivalent. Both receive requests, execute against them, and return results. The inline gateway pattern applies uniformly across all of them. One trust layer. One policy model. One audit trail — across every service in the workflow. Open source: github.com/macawsecurity/secureAI

1. External MCP Is Becoming Load-Bearing Infrastructure

Model Context Protocol has moved faster than most protocol adoptions. Within months of its introduction, major enterprise platforms began exposing MCP endpoints as first-class integration surfaces. An agent workflow that calls an LLM, reads from Salesforce, queries a data warehouse, runs analytics against a lakehouse, and notifies Slack is not a prototype — it is a production pattern running in enterprises today.

```
MODERN AGENTIC WORKFLOW:

  User Request
       │
       ▼
  ┌─────────────────┐
  │  Orchestrator   │  (LangChain / OpenClaw / NanoClaw)
  └────────┬────────┘
           │
   ┌───────┼───────┬──────────┬──────────┬──────────┐
   ▼       ▼       ▼          ▼          ▼          ▼
 LLM:    MCP:    MCP:        MCP:       MCP:       MCP:
 OpenAI  Sales-  Snow-   Databricks   Slack      GitHub
 Claude  force   flake
   │       │       │          │          │          │
   ▼       ▼       ▼          ▼          ▼          ▼
 Reason/ CRM   Warehouse  Lakehouse   Notify    Code /
 Generate Data   Queries    + ML       Teams     Repos
```

Every node in this diagram — LLM APIs included — is a service that receives a request and acts on it. Each invocation carries an auth token issued to a session, not to a specific operation. None carry a record of which user authorized the call, what they intended, or whether parameters arrived unmodified.

The multi-hop case makes this concrete. When an orchestrator calls four services in sequence — using tokens received from step one to authorize step two — the authorization chain has no cryptographic backbone. If any token is replayed, injected, or compromised, downstream operations proceed as legitimate. Because from the receiving server's perspective, they are.

```
THE SESSION PROBLEM IN MULTI-HOP WORKFLOWS:

  Alice authenticates
          │
          ▼
    Session Token ───────────────────────────────────────────────►
          │               │               │               │
     Hop 1: LLM      Hop 2: CRM      Hop 3: DWH      Hop 4: Notify
    (token valid)   (token valid)   (token valid)   (token valid)
          │
          ◉  Prompt injection here produces a valid token
               for all downstream hops.
               No per-invocation proof exists.
               No audit trail distinguishes
               legitimate from injected.
```

This is not a theoretical attack surface. It is the current production posture of every agentic deployment that authenticates with sessions rather than per-invocation signed proofs.

2. Two Ways to Connect External Services to a Trust Layer

The correct response to this gap is a cryptographic trust layer: a system that issues signed, policy-evaluated, per-invocation authorizations and maintains a tamper-evident chain across every hop. The architectural question is where that trust layer lives relative to the calling application. There are two meaningful options.

Option A — The Centralized Gateway

A gateway sits between the application and external services. All outbound calls — to LLM APIs and MCP servers — are routed through it. The gateway authenticates, evaluates policy, logs, and forwards.

```
OPTION A — CENTRALIZED GATEWAY:

  Application
       │  outbound call
       ▼
  ┌───────────────────────┐
  │       Gateway         │
  │  ───────────────────  │
  │  Auth + Policy        │
  │  Logging + Enforce    │
  └──────────┬────────────┘
             │
      ┌──────┴──────┐
      ▼             ▼
  Service-A     Service-B
  (LLM / MCP)  (LLM / MCP)

  Latency:         ~50ms per hop (network round-trip)
  Blast radius:    shared — one misconfiguration affects all traffic
  Configurability: coarse — gateway-level rules, not per-user
```

Gateways work. They are a legitimate architectural choice for deterministic, low-hop, network-layer traffic — the workloads they were designed for. Where they struggle is agentic traffic: multi-hop, user-specific, context-sensitive. Policy evaluated at the network edge has lost the user context it needs. Governance in the gateway is governance about routing — not about what any specific user was authorized to do at a specific moment.

Option B — The Inline Gateway

The inline gateway instantiates trust layer enforcement at the point of the outbound call, inside the calling application. The macawClient is created dynamically, wraps the call — LLM API or MCP server — and evaluates policy, signs the invocation, and records lineage inline, in-process, without a network round-trip.

```
OPTION B — INLINE GATEWAY:

  Application
       │  Dynamic macawClient instantiation
       ▼
  ┌──────────────────────────────────────────┐
  │   Inline at the outbound call:           │
  │                                          │
  │   ┌──────────────────────────────────┐   │
  │   │  macawClient                     │   │
  │   │  ──────────────────────────────  │   │
  │   │  Identity propagation            │   │
  │   │  Policy evaluation (full context)│   │
  │   │  Per-invocation signature        │   │
  │   │  Lineage recording               │   │
  │   └─────────────────┬────────────────┘   │
  └──────────────────────────────────────── ┘
                        │  signed, policy-evaluated call
                 ┌──────┴──────┐
                 ▼             ▼
           Service-A     Service-B
          (LLM / MCP)   (LLM / MCP)

  Latency:          ~2ms per hop (in-process)
  Blast radius:     per-adapter — scoped to this application
  Configurability: per-user, per-invocation, per-context — full MAPL
```

Why Inline Is Architecturally Superior

Property	Gateway	Inline Gateway
Latency per hop	~50ms (network RT)	~2ms (in-process)
User context at enforcement	Flattened to service identity	Full user identity + claims
Policy granularity	Gateway-level rules	Per-user, per-invocation, per-context
Blast radius	Shared across all traffic	Scoped to this adapter
Single point of failure	Yes — outage blocks all	No — each adapter independent
Covers LLM APIs + MCP	Two separate gateways needed	Yes — natively, same pattern
Prompt injection protection	Network-level detection	Cryptographic: modified params → no valid sig

The inline gateway is not a tradeoff. It provides the same security guarantees as a centralized gateway — with better latency, lower blast radius, more expressive policy, and uniform coverage across both LLM and MCP surfaces.

3. The Inline Gateway Pattern in Practice

The Unified Model: LLMs and MCP Servers Are Semantically Equivalent

The key architectural insight: in MACAW's model, everything is an agent with tools. An LLM API endpoint — OpenAI, Anthropic, LiteLLM — is not categorically different from an MCP server. Both receive a structured request, execute against it with real user data, and return results. Both need per-invocation proof of authorization. SecureOpenAI and SecureAnthropic are inline gateways for LLM endpoints — the same pattern as SecureMCPProxy for external MCP servers. One trust layer covers the entire agentic surface.

```
MACAW UNIFIED SERVICE MODEL:

  ┌──────────────────────────────────────────────────────────────┐
  │                      MACAW Trust Layer                       │
  │   Identity   │  Policy (MAPL)  │  Signature  │  Lineage     │
  └──────────────┴─────────────────┴─────────────┴──────────────┘
         │                │                │                │
         │                │                │                │
   LLM Endpoint      LLM Endpoint     MCP Server       MCP Server
  (openai-finance)  (openai-eng)    (snowflake-     (salesforce-crm)
   SecureOpenAI      SecureOpenAI     analytics)
                                   SecureMCPProxy   SecureMCPProxy

  Each is a named, registered instance — own policy, own audit
  stream, own rate limits, own control plane visibility.
```

Three Deployment Modes, One Pattern

The pattern deploys differently depending on what you control. The trust layer is identical in all three modes.

Mode 1 — Embedded Inline Gateway (you control the source)

When you own the Python source — applications, internal MCP servers, LLM integrations — the inline gateway is a drop-in import. One line changes. No architecture changes. No new infrastructure.

```
# LLM endpoint — before
from openai import OpenAI
client = OpenAI()

# LLM endpoint — after (embedded inline gateway)
from macaw_adapters.openai import SecureOpenAI
client = SecureOpenAI(app_name="finance-agent")

# MCP server — before
from mcp.server import MCPServer
server = MCPServer("analytics-tool")

# MCP server — after (same pattern, different endpoint type)
from macaw_adapters.mcp import SecureMCP
server = SecureMCP("analytics-tool")
```

Available: SecureOpenAI, SecureAnthropic, SecureLiteLLM, SecureMCP, SecureLangChain. Open source, Apache 2.0: github.com/macawsecurity/secureAI

Mode 2 — Sidecar Inline Gateway (you run it but cannot modify it)

When source modification is impractical — TypeScript orchestrators, closed-source agents, third-party frameworks — the sidecar provides the same trust layer guarantees at the process boundary. **SecureOpenClaw** implements this for OpenClaw-compatible orchestrators. **SecureClaudeCode** implements this for Claude Code, Anthropic's coding agent — closed source, communicates natively over MCP. The inline gateway wrapped the MCP calls at the client side: every tool invocation carries signed, policy-evaluated authorization with no source modification and no architecture change to the agent. The same pattern applies to any code agent that speaks MCP natively.

Mode 3 — Proxy Inline Gateway (you call it externally)

The case most enterprises are actively working through: external MCP servers. Salesforce. Snowflake. Databricks. Slack. GitHub. Endpoints you call, not endpoints you build or run.

```
# Standard outbound MCP call — no trust layer
response = mcp_client.call_tool("salesforce", "query_records", params)

# Proxy inline gateway — macawClient at call time
from macaw_adapters.mcp import SecureMCPProxy

proxy = SecureMCPProxy(
    target="salesforce-mcp-endpoint",
    user_context=current_user.jwt,
    app_policy="crm-read-only"
)
response = proxy.call_tool("query_records", params)
```

The outbound call now carries a signed, policy-evaluated invocation. The user's identity is bound to the operation. Parameters are cryptographically bound — a prompt injection that modifies the query produces a signature mismatch and is rejected before reaching the external server. Lineage is recorded: Alice authorized this query, at this time, with these parameters, as part of this workflow.

4. A Cryptographic Chain, Not a Session

What distinguishes the MACAW trust layer from a policy enforcement wrapper is that security is expressed as mathematical properties, not configuration. Four properties hold for every operation through every deployment mode — Embedded, Sidecar, or Proxy — across every service type, LLM or MCP.

Identity Propagation, Not Impersonation. When Alice initiates a workflow, her identity propagates through every hop. Downstream services receive a claim chain cryptographically anchored to Alice's original authentication — not a service account token with her name attached. No vault lookups. No session state. The identity is in the signature.

Parameter Binding. Every invocation is signed over its parameters. A prompt injection that modifies a query, tool argument, or model instruction produces a signature mismatch. The modified call is rejected at enforcement. This is prevention by construction — not detection.

Monotonic Restriction. Permissions narrow through the delegation chain. An agent authorized to read CRM records cannot grant a sub-agent permission to write them. The permission intersection is computed at each hop, across every service boundary:

```
    Effective Permission =
        User Intent
        ∩  Application Policy
        ∩  Service Policy         ← applies to LLM endpoints and MCP servers equally
        ∩  System Policy
        ∩  Context State

    → Always a strict subset of the invoking principal's permissions.
    → Delegation cannot escalate privilege.
```

Blast Radius Containment. Each adapter is an independent enforcement point. A compromise of the Salesforce MCP connection does not affect the Snowflake or OpenAI connections. There is no shared session state to laterally pivot through.

```
  DELEGATION CHAIN — CRYPTOGRAPHIC ANCHORING:

    Alice authenticates
            │
            │  σ_alice  (signed JWT, Ed25519)
            ▼
    ┌────────────────────┐
    │   Orchestrator     │  σ_hop1 = Sign(alice_claims ∩ app_policy, privkey)
    └──────────┬─────────┘
               │  σ_hop1
               ▼
    ┌────────────────────┐
    │   LLM: OpenAI      │  σ_hop2 = Sign(σ_hop1_claims ∩ llm_policy, privkey)
    └──────────┬─────────┘
               │  σ_hop2
               ▼
    ┌────────────────────┐
    │   MCP: Salesforce  │  σ_hop3 = Sign(σ_hop2_claims ∩ tool_policy, privkey)
    └──────────┬─────────┘
               │  σ_hop3
               ▼
    ┌────────────────────┐
    │   MCP: Snowflake   │  σ_hop4 = Sign(σ_hop3_claims ∩ context_state, privkey)
    └────────────────────┘

    At any point: reconstruct the full chain.
    At any point: verify Alice authorized this.
    At any point: prove no permission was added — at any hop, to any service.
```

This is the difference between a session and a chain of custody. A session tells you who authenticated. A chain of custody tells you what they authorized, at each step, across every service boundary, and proves no step deviated from their original intent.

5. Governance Without a Governance Layer

The governance fragmentation enterprises experience with agentic AI has a structural cause: governance in a gateway is governance about where traffic is routed. It cannot answer the question that matters at audit time: who authorized this specific operation, with what intent, and does the record prove it? MACAW governance is a property of every operation, not of the network path. The audit trail is hash-chained, tamper-evident, and integrates directly with SIEM systems — Datadog, Grafana, Splunk — without a translation layer.

Named Service Instances: Governance at Semantic Granularity

Every service endpoint is a named, registered instance in the MACAW control plane. Governance is expressed at the level that actually matters — not "traffic to OpenAI" but "traffic from the finance agent to the OpenAI endpoint operating under the finance policy."

```
NAMED SERVICE INSTANCES IN THE CONTROL PLANE:

  openai-finance      → policy: finance-policy      rate-limit:  10/min
  openai-engineering  → policy: eng-policy           rate-limit:  50/min
  snowflake-analytics → policy: analyst-data         rate-limit:  20/min
  snowflake-executive → policy: executive-data       rate-limit:   5/min
  salesforce-crm      → policy: crm-read-only        rate-limit:  30/min
  databricks-ml       → policy: ml-compute           rate-limit:  15/min

  Each instance:
    ✓  Own policy (MAPL)      — enforced per-invocation
    ✓  Own audit stream       — per-instance in the control plane
    ✓  Own rate limits        — independently configurable
    ✓  Own observability      — distinct in dashboards and alerts
    ✓  Independent lifecycle  — tighten one without touching others
```

Tightening snowflake-executive does not affect snowflake-analytics. Compliance questions — who accessed executive-tier data via Snowflake, when, with what parameters — are answerable from a single named instance's audit stream. Named instances are also composable: the same underlying API serves as many governance surfaces as the organization needs.

Policy Hierarchy: MAPL in Practice

Policy is expressed in MAPL — MACAW Access Policy Language. Policies compose through inheritance: user policy extends team policy, which extends company policy. The effective permission at any call site is the intersection of the full hierarchy.

```
// Company-level — outer boundary
{
  "policy_id": "company:FinTech Corp",
  "resources": ["tool:**/generate","tool:trading/*","tool:financial_analysis"],
  "denied_resources": ["*.secret","*.password","*.key","*.pem"],
  "constraints": {
    "parameters": {
      "tool:*/generate": {
        "model": ["gpt-3.5-turbo","gpt-4","claude-3-haiku","claude-opus"],
        "max_tokens": { "max": 4000 }, "temperature": { "max": 1.0 }
      }
    }, "rate_limit": 100
  }
}
```

```
// Alice — entry-level analyst (inherits team:Reporting, adds restrictions)
{
  "policy_id": "user:alice", "extends": "team:Reporting",
  "denied_resources": ["data:executive/*","data:confidential/*"],
  "constraints": {
    "parameters": {
      "tool:*/generate": {
        "model": ["gpt-4","gpt-3.5-turbo","claude-3-haiku"],
        "max_tokens": { "max": 500 }, "temperature": { "max": 0.5 }
      },
      "tool:trading/execute_trade": { "amount": { "max": 20000 } }
    }, "rate_limit": 10
  }
}
```

```
// Bob — manager (same inheritance, broader permissions)
{
  "policy_id": "user:bob", "extends": "team:Reporting",
  "constraints": {
    "parameters": {
      "tool:*/generate": {
        "model": ["gpt-3.5-turbo","gpt-4","claude-3-haiku","claude-3-sonnet","claude-opus"],
        "max_tokens": { "max": 2000 }, "temperature": { "max": 0.8 }
      }
    }, "rate_limit": 30
  }
}
```

At call time: Alice calling openai-finance is constrained to 500 tokens, three models, temperature ≤ 0.5, rate-limited to 10 req/min — regardless of what the company policy permits at the ceiling. The policy follows Alice, not the service. Neither Alice nor any agent acting on her behalf can grant downstream permissions that exceed her own.

6. Deployment Mode Comparison

The inline gateway pattern has precedent in systems engineering. OpenSSL is linked into the application that needs TLS — not deployed as a sidecar. JDBC drivers are in-process libraries, not network proxies. JWT validation runs where the token is consumed. In each case, the security primitive lives closest to the operation it protects. The deployment mode varies by what you control. The trust layer does not.

	Embedded	Sidecar	Proxy	Gateway
Latency	~2ms	~5ms	~3ms	~50ms
Source access required	Yes	No	No	No
Per-invocation policy	Yes	Yes	Yes	Partial
User context	Full	Full	Full	Reduced
Blast radius	Per-adapter	Per-process	Per-proxy	Shared
LLM + MCP native coverage	Yes	Yes	Yes	Two gateways
Ops complexity	Low	Medium	Low	High

All three MACAW modes share the same identity propagation, cryptographic signing, policy evaluation, audit trail, and control plane visibility. Deployment mode is a function of what you control, not a security tradeoff.

7. Lessons Learned

The first version of the inline gateway pattern took approximately six months to build to production readiness. The architecture was clear early. What production revealed was different.

Multi-hop identity collisions. When two independent agents execute on behalf of the same user simultaneously, their delegation chains can produce conflicting identity claims at a shared downstream service. The resolution requires a monotonic nonce scheme tied to the session, not the user — a non-obvious distinction that only appears under concurrent load.

Key rotation under clock skew. Rotating signing keys where adapter instances have slightly different clocks produces a window where valid signatures are rejected and recently-expired ones pass. The fix requires bounded clock skew enforcement at key issuance, not at verification.

IdP normalization at scale. Okta, Azure AD, GitHub, and Keycloak each represent group membership differently in JWT claims. At scale with dynamic group changes, normalization needs to be real-time with invalidation — not pre-computed at startup.

Policy inheritance negative space. A child policy that explicitly denies an operation the parent is silent on creates a different enforcement outcome than a child policy that is also silent. The silence-vs-denial distinction requires explicit handling or it creates exploitable gaps.

The 10,000 requests-per-second cliff. At approximately 10K req/sec per adapter instance, cryptographic signing overhead becomes the bottleneck. The solution is batched signing with ordered commitment — borrowed from certificate transparency logs — that maintains per-invocation guarantees while distributing the work.

Revocation windows in long-running workflows. An agent workflow running for 20 minutes may hold a signed authorization whose underlying policy was revoked at minute 15. The inline gateway maintains a revocation check interval configurable per policy tier — a feature that appears unnecessary until the first mid-workflow policy change in production.

Named instance proliferation. As organizations register more named service instances, the control plane needs hierarchical grouping to remain navigable. Instance groups, inherited defaults, and bulk policy operations appear on the operations ticket after the sixth team onboards — not on the architecture diagram.

The first milestone is months. The rest is production. Every item above is now handled. None of it was visible from the architecture diagram.

8. Conclusion

Organizations deploying agentic AI today are making a consequential architectural choice, often without recognizing it as one. The choice between routing outbound calls through a gateway versus instantiating the trust layer inline — across LLM endpoints and MCP servers alike — is not a deployment preference. It is a decision about how much context, granularity, and cryptographic proof your security posture will have at every operation.

Gateways solve a real problem. For deterministic, low-hop, service-to-service traffic, they are appropriate. For agentic workflows that carry user identity across multiple LLM and MCP services, evaluate dynamic context-sensitive policy, and need tamper-evident lineage — the inline gateway is the correct architecture. And because the trust layer applies uniformly to LLM APIs and MCP servers, there is no second gateway to operate, no separate policy surface to maintain, and no seam in the audit record between model calls and tool calls.

The pattern generalizes. Any orchestrator that speaks MCP natively can be secured without source modification. Any external MCP endpoint, any LLM provider, any internal service can be registered as a named instance with its own policy and its own audit stream — independently configurable, independently observable, composable as the organization's needs evolve.

The security is not in where you place the wrapper. It is in whether the wrapper connects to a trust layer that travels with the request — one that carries the calling user's identity, binds it to the operation's parameters, enforces the intersection of applicable policies across every service boundary, and produces a cryptographically reconstructable record of every decision made. That is the pattern.

MACAW Security · macawsecurity.com · github.com/macawsecurity/secureAI · research@macawsecurity.com