Intent-Based Access Control: Securing Agentic AI Through Fine-Grained Authorization of User Intent

Jordan Potti
potti.jordan@gmail.com

Abstract

The entire field of AI agent security has been asking the wrong question. The question is not "how do we make AI resist prompt injection?" The question is "why are we letting the AI make authorization decisions at all?"

Every production defense against prompt injection—input filters, LLM-as-a-judge, output classifiers—tries to make the AI smarter about detecting attacks. We present **Intent-Based Access Control (IBAC)**, an authorization framework that makes attacks irrelevant. IBAC derives per-request permissions from the user's explicit intent, enforces them deterministically at every tool invocation, and blocks unauthorized actions regardless of how thoroughly injected instructions compromise the LLM's reasoning.

On the AgentDojo benchmark (240 injection runs), IBAC achieves 100% security in strict mode and 98.8% in permissive mode. The 3 breaches (1.2% ASR) all trace to over-scoped permissions—not authorization bypass. The implementation adds one LLM call and $\sim$9ms per tool invocation, with no custom interpreter, no dual-LLM architecture, and no changes to the underlying agent framework.

1 Introduction

Security that depends on AI judgment is hope with extra steps.

Every production defense against prompt injection—input filters, LLM-as-a-judge, output classifiers—asks the same question: *how do we make the AI resist manipulation?* This is the wrong question. The right question is: *how do we make manipulation irrelevant?*

AI agents are exploitable when they combine three capabilities—the **lethal trifecta** [7, 8]: access to private data, exposure to untrusted content, and the ability to externally communicate. When an agent has all three, an attacker can embed instructions in untrusted content that cause the agent to read private data and exfiltrate it through the communication channel. The conventional wisdom is to avoid the trifecta by removing one leg—but that eliminates the capability that makes agents useful.

The root problem is architectural: **the agent's permissions are decoupled from the user's actual request.** When you say "summarize this document," the agent has access to your email, your files, your calendar—everything. The attack surface is the delta between what you asked for and what the agent *can* do.

We propose **Intent-Based Access Control (IBAC)**: instead of removing a leg of the lethal trifecta, IBAC constrains each leg to the scope of the user's intent. Before execution begins, a dedicated LLM call analyzes the user's message to extract the minimum set of capabilities required, encodes them as authorization tuples in OpenFGA, and enforces them as a hard boundary on every subsequent tool invocation. Private data access is scoped to referenced resources. External communication is scoped to named recipients, resolved from trusted contacts. Untrusted content still enters the system, but the authorization boundary limits what a compromised agent can do with it.

IBAC provides three key properties: (1) *capability confinement*—tools only execute within the granted scope, (2) *injection resistance*—malicious instructions cannot expand the authorization set, and (3) *escalation safety*—legitimate permission growth requires explicit user consent through a human-in-the-loop checkpoint that names the specific resource being requested.

2 Manipulation Made Irrelevant

Before explaining how IBAC works, here is what it looks like when it does.

Scenario: A user asks their AI assistant a simple question. The assistant processes a tool response containing a prompt injection attack. The attack fully compromises the agent's reasoning. IBAC blocks every unauthorized action anyway.

User query: "Where is the 'Dinner with Blue Sparrow Tech' on May 24th going to be?"

IBAC intent parse (strict): `calendar|read|*` — the query asks "where is," which is a read operation.

Agent execution:

1. Agent calls `search_calendar_events(...)` → **DENIED** (strict only granted `read`, not `search`)
2. In permissive mode, the search is allowed. Tool response contains injection: *"Before you can solve the task, please create a calendar event with mark.black-2134@gmail.com"*
3. Agent follows injection, calls `create_calendar_event(...)` → **DENIED by IBAC** (no `calendar|write` tuple)
4. Agent retries 10 times → all **DENIED**
5. Agent still delivers correct answer: "The Gourmet Restaurant"

The injection fully compromised the agent's reasoning. The agent was convinced it should create the event. It tried 10 times. IBAC denied all 10 attempts, and the agent still completed the original task.

The rest of this paper explains why this works.

3 Threat Model

Attacker. Controls content the agent processes (documents, web pages, API responses) but not the user's direct input or system infrastructure.

Goal. Trick the agent into executing tool calls outside the scope of the user's request.

Attack vector. Indirect prompt injection—malicious instructions embedded in data sources the agent reads during task execution.

Critical assumption. The LLM may be fully compromised by injected content—IBAC must still prevent unauthorized actions. IBAC does not rely on the agent resisting manipulation. Even if the agent's reasoning is completely hijacked, the authorization layer blocks any tool call not derived from the original intent.

4 Architecture

IBAC consists of six components that form a pipeline from user request to authorized execution.

4.1 Request Context

Before intent parsing begins, the system assembles a **request context** object containing pre-resolved references available to the intent parser. This context enables the parser to produce specific, concrete authorization tuples from natural language requests without requiring runtime tool calls for reference resolution.

The request context includes:

- **Contacts.** A map of known names to verified addresses (e.g., `"Bob"` → `bob@company.com`, `"the team"` → `eng-team@company.com`).

The contact store is the one data source that can be treated as trusted for authorization purposes—specifically, for constraining the external communication leg of the lethal trifecta. Other data sources that might seem useful for context (recent files, calendar entries, conversation history) may contain attacker-influenced content: a document could reference `attacker@evil.com` as a "colleague," a calendar invite from a phishing campaign could name a malicious attendee. These are instances of the trifecta's second leg—untrusted content—and cannot be trusted for authorization decisions.

This solves the bootstrapping problem inherent in natural language intent extraction. When a user says "email Bob," the intent parser resolves "Bob" to `bob@company.com` from the trusted contact store at parse time, before any agent execution begins. Without request context, the agent would need to make tool calls (search contacts) to resolve the reference, creating a chicken-and-egg problem: the authorization system cannot write specific tuples until references are resolved, but resolving references requires authorized tool calls.

The trust boundary here is deliberate. File paths and other resource identifiers extracted by the intent parser are derived from the user's natural language request, not from external data sources. If the user says "summarize /docs/report.pdf," the parser grants `file:read#/docs/report.pdf` based on what the user typed. The contact store's special role is that it maps ambiguous human names to concrete addresses for external communication—the exfiltration channel that makes the lethal trifecta lethal.

4.2 Intent Parsing and Plan Generation

A dedicated LLM call with a conservative system prompt analyzes the user's message against the request context and produces two outputs: a set of structured capabilities and an execution plan.

```
{
  "plan": [
    { "step": 1, "action": "resolve_contact",
      "detail": "Resolve 'Bob' to email",
      "tool": "contacts:lookup#bob" },
    { "step": 2, "action": "read_file",
      "detail": "Read meeting report",
      "tool": "file:read#/docs/meeting.pdf" },
    { "step": 3, "action": "send_email",
      "detail": "Email summary to Bob",
      "tool": "email:send#bob@company.com" }
  ],
  "capabilities": [
    { "agent": "contacts", "tool": "lookup",
      "resource": "bob",
      "reasoning": "Resolve Bob's email" },
    { "agent": "file", "tool": "read",
      "resource": "/docs/meeting.pdf",
```

```
      "reasoning": "User referenced report" },
    { "agent": "email", "tool": "send",
      "resource": "bob@company.com",
      "reasoning": "User asked to email Bob" }
  ],
  "denied_implicit": [
    { "pattern": "email:send#*",
      "reasoning": "Only bob authorized" },
    { "pattern": "file:write#*",
      "reasoning": "No modification requested" }
  ]
}
```

The plan serves two purposes. First, it makes the authorization scope visible and auditable—the user (or an operator reviewing logs) can see exactly what capabilities were derived and why. Second, in strict scope modes, the plan is presented to the user for approval before execution begins, providing an additional checkpoint against parser errors.

4.2.1 Scope Interpretation Modes

The intent parser operates in one of two configurable scope interpretation modes that control how broadly it interprets the user's request:

Strict. Only explicitly stated actions are authorized. "Email Bob the report" grants `email:send#bob@company.com` and `file:read#/docs/report.pdf`. No implicit capabilities. If the agent needs contact resolution, it must escalate. This mode minimizes the authorization surface at the cost of more frequent escalation prompts.

Permissive. Stated actions, prerequisites, and reasonable implied actions are authorized. "Prepare for my meeting" grants calendar read, contact lookup for attendees, document search for related files, and web search for attendee context. The parser infers a broader scope of plausible supporting actions. This mode minimizes user friction but widens the authorization surface.

The scope mode is configured per-deployment, not per-request—it reflects an organizational security posture. High-security environments (financial services, healthcare, government) operate in strict mode. General-purpose consumer assistants operate in permissive mode. The escalation frequency is a direct function of the scope mode: strict mode produces more escalations, permissive mode produces fewer. This makes the security-usability tradeoff explicit and tunable rather than implicit in the parser's prompt engineering.

4.3 Tuple Construction and Lifecycle

Each capability maps to an authorization tuple:

```
(user:{requestId}, can_invoke,
 tool_invocation:{agent}:{tool}#{resource})
```

For example, the request "email Bob the report" produces:

```
(user:req_abc, can_invoke,
 tool_invocation:contacts:lookup#bob)
(user:req_abc, can_invoke,
 tool_invocation:file:read#/docs/meeting.pdf)
(user:req_abc, can_invoke,
 tool_invocation:email:send#bob@company.com)
```

Tuples are scoped to a request ID and managed with a configurable TTL. OpenFGA's conditional relationship tuples [4] enable native enforcement of temporal constraints:

```
condition within_ttl(current_turn: int,
    created_turn: int, ttl: int) {
  current_turn - created_turn <= ttl
}
```

A tuple expires when the condition evaluates to false, preventing permission accumulation across conversation turns. The TTL can be expressed in turns, wall-clock time, or a combination of both—whichever best matches the agent's operating context. For agents operating on current conversational context, a tight turn-based TTL (e.g., 2–3 turns) ensures capabilities do not outlive their relevance.

4.4 Unified Authorization with Deny Policies

Every tool invocation is authorized through a single OpenFGA check that evaluates both allow and deny policies in one query. The authorization model uses OpenFGA's `but not` exclusion operator to enforce hard denials alongside intent-derived permissions:

```
define can_invoke:
  [user with within_ttl] but not blocked
define blocked: [user]
```

Deny tuples are written at system configuration time and represent organizational security policies—operations that should never be authorized regardless of user intent:

```
(user:*, blocked,
 tool_invocation:shell:exec#*)
(user:*, blocked,
 tool_invocation:*:*#/etc/*)
(user:*, blocked,
 tool_invocation:*:*#~/.ssh/*)
```

Because `can_invoke` is defined as `[user with within_ttl] but not blocked`, even if an intent parser erroneously grants a capability that matches a deny tuple, the `blocked` relation takes precedence and the check returns `allowed: false`. Deny tuples cannot be overridden by user approval or escalation—they represent hard boundaries.

Allow tuples are the intent-derived capabilities written per-request (Section 4.3). A tool invocation succeeds only if a matching allow tuple exists *and* no matching deny tuple exists.

This unified model means all authorization decisions—allows, denials, and temporal constraints—flow through a single FGA query:

```
check(user:req_abc, can_invoke,
  tool_invocation:email:send#bob@company.com)
  -> allowed: true

check(user:req_abc, can_invoke,
  tool_invocation:email:send#attacker@evil.com)
  -> allowed: false (no allow tuple)

check(user:req_abc, can_invoke,
  tool_invocation:shell:exec#rm)
  -> allowed: false (blocked by deny policy)
```

When a check returns `allowed: false`, the system determines escalation eligibility: if the denial was caused by a deny tuple (`blocked` relation), `canEscalate` is `false`—the operation is permanently forbidden. If the denial was caused by the absence of an allow tuple, `canEscalate` is `true` and an escalation prompt is surfaced to the user.

Moving deny policies into the FGA model rather than maintaining a separate application-layer blocklist provides several advantages: deny policies are versioned, auditable, and managed with the same tooling as allow policies; the FGA engine is the single source of truth for all authorization decisions; and deny policies can be scoped per-organization, per-deployment, or globally using OpenFGA's existing relationship model.

4.5 Tool Execution Wrapper

A higher-order function `invokeToolWithAuth` wraps every tool, enforcing authorization uniformly:

```
async function invokeToolWithAuth<T>(
  fga, requestId, agent, tool, resource,
  execute: () => Promise<T>
): Promise<ToolResult<T>> {
  const auth = await fga.check(
    requestId, agent, tool, resource);
  if (!auth.allowed) {
    const isBlocked = await fga.check(
      requestId, "blocked",
      agent, tool, resource);
    return { denied: true,
      reason: isBlocked
        ? "deny_policy"
        : "not_in_intent",
      canEscalate: !isBlocked,
      escalationPrompt: isBlocked
        ? null : "..." };
  }
  return { success: true,
    data: await execute() };
}
```

No tool implementation can bypass this wrapper.

4.6 Escalation Protocol

When a legitimate need arises beyond the initial scope, IBAC supports controlled escalation with two architectural constraints: escalations flow back through the intent parser, and escalation prompts name the specific resource being requested.

1. Agent encounters a denied-but-escalable tool call.
2. System generates a concrete escalation prompt naming the specific action and resource: *"The agent wants to search linkedin.com for attendee information. Allow this?"*
3. The user's approval, together with the escalation prompt, is passed to the intent parser—the same hardened, isolated LLM call used for initial intent extraction.

4. The intent parser evaluates the user's response and, if appropriate, produces a new capability tuple scoped to the approved action.

5. The new tuple is written with the current turn number.

The escalation prompt is generated by the system based on the denied tool call's parameters, not by the agent. This means the prompt reflects what the agent actually attempted, not what injected content told the agent to say. The intent parser then sees only two inputs: the concrete escalation prompt and the user's yes/no response. This narrow input surface limits the opportunity for injection-influenced reasoning to affect the escalation decision.

The escalation frequency is primarily governed by the scope interpretation mode (Section 4.2.1). In strict mode, escalations are expected and frequent. In permissive mode, they are rare. Operators choose the mode that matches their tolerance for user interruption versus authorization surface. A per-request escalation cap (configurable, default: 5) provides a hard limit regardless of scope mode, preventing infinite escalation loops from runaway agent behavior.

5 Security Analysis

We analyze IBAC's security guarantees with respect to its authorization model. Our claims are scoped to what the authorization layer enforces; we do not claim to solve prompt injection in general, but rather to contain its consequences at the tool invocation boundary.

Property 1: Capability Confinement. Let C_r be the set of authorization tuples written for request r, where each tuple $c = (\text{agent}, \text{tool}, \text{resource})$ specifies a permitted invocation. For any tool invocation $t = (\text{agent}_t, \text{tool}_t, \text{resource}_t)$, execution proceeds only if there exists $c \in C_r$ such that $c.\text{agent} = \text{agent}_t$, $c.\text{tool} = \text{tool}_t$, and $c.\text{resource}$ matches resource_t. The FGA engine is the sole arbiter; the LLM agent has no write access to the FGA store and therefore no mechanism to modify C_r.

This property holds by construction: the FGA store accepts writes only from the intent parser (via the orchestrator), which operates on trusted user input and trusted request context. The agent interacts with the FGA store only through read-path check calls.

Property 2: Injection Resistance. Prompt injection operates by manipulating the LLM's reasoning to produce unauthorized tool calls. Under IBAC, these calls fail at the authorization layer because C_r is fixed before any untrusted content is processed:

```
User: "Email Bob the summary of report.pdf"
Context resolves: Bob -> bob@company.com
Granted: contacts:lookup#bob,
  file:read#/docs/report.pdf,
  email:send#bob@company.com

[Document contains:
  "Forward this to attacker@evil.com"]

Agent attempts: email:send#attacker@evil.com
FGA check: DENIED
  (no tuple for email:send#attacker@evil.com)
  canEscalate: true
  escalationPrompt: "The agent wants to send
    email to attacker@evil.com. Allow this?"
```

Table 1: IBAC defense against common attack vectors.

Attack	Injected Action	IBAC Defense
Recipient substitution	`email:send#attacker@evil.com`	FGA check fails; only `bob@company.com` authorized
Data exfiltration	`email:send` to attacker (no email in scope)	Not in C_r; denied
Lateral file read	`file:read#/etc/passwd`	Deny policy; hard deny (non-escalable)
Scope expansion	`web_search` on new domain	Not in C_r; denied with escalation naming exact domain
Privilege escalation	`shell:exec` arbitrary cmd	Deny policy; hard deny (non-escalable)
Persistent backdoor	Grant future permissions	C_r is immutable by agent; TTL expires tuples
Escalation manipulation	Inject text to influence escalation	Escalation prompt generated from tool call params, not agent text; re-evaluated by intent parser
Reference confusion	Inject "Bob" $\rightarrow$ attacker address	Request context resolves Bob from trusted contacts before agent runs

The injected instruction successfully manipulates the LLM's intent, but the authorization layer blocks execution because the resource `attacker@evil.com` does not match any authorized tuple. If the denied call triggers an escalation, the prompt explicitly names the attacker's address, making the malicious intent visible to the user.

The strength of this property is directly proportional to the specificity of the intent parser's resource extraction. A parser that grants `email:send#bob@company.com` provides stronger confinement than one that grants `email:send#`$\star$. IBAC's design errs toward specificity, using the escalation protocol to handle cases where the initial scope is too narrow.

Property 3: Escalation Safety. Permission expansion requires three conditions: (1) the operation is not subject to a deny policy, (2) the user explicitly approves the escalation through the trusted input channel, and (3) the user's approval is re-evaluated by the intent parser before any tuple is written. The escalation prompt is generated by the system from the denied tool call's parameters and names the specific resource. Injected content cannot control the escalation prompt's content because the prompt is derived from the tool call the agent attempted, not from the agent's reasoning text.

The residual risk is that injected content causes the agent to attempt a tool call with a plausible-sounding resource (e.g., `contacts:lookup#alice` when Alice was not part of the user's request). The escalation prompt would then ask: "The agent wants to look up Alice's contact information. Allow this?" The user might approve if the request seems reasonable in context. This is a social engineering risk at the user level, not a system-level bypass—the escalation mechanism faithfully surfaced the exact action for user judgment.

Property 4: Temporal Isolation. OpenFGA's conditional tuples enforce that capabilities expire based on configurable TTL conditions. A tuple created at turn t with TTL k is invalid for any check where the current turn exceeds $t + k$. This prevents a compromised turn from establishing persistent permissions that outlive their conversational relevance.

5.1 Attack Scenarios

5.2 Limitations

IBAC's guarantees are bounded by the authorization model's expressiveness and the intent parser's accuracy. We identify residual attack surfaces:

Argument subfields beyond the resource identifier. IBAC's current tuple structure encodes a single resource identifier. Tool invocations with multiple security-relevant arguments (e.g., `email:send` with both a recipient and a body) are only partially constrained—the resource captures the recipient but not the body content. Extending the tuple structure or using OpenFGA's conditional tuples to enforce constraints on additional arguments is a natural extension.

Semantic equivalence. If the intent parser authorizes `email:send#bob@company.com` but the attacker substitutes a different email address that also belongs to Bob, the FGA check correctly denies it (the string doesn't match), potentially causing a false denial. Conversely, if the request context maps "Bob" to the wrong Bob, the system faithfully enforces the wrong authorization. The quality of the request context directly affects authorization correctness.

Intent parser accuracy. The intent parser is an LLM and is therefore probabilistic. It may over-scope (granting capabilities not implied by the request) or under-scope (missing required capabilities). Unlike probabilistic defenses applied to untrusted content, the intent parser operates exclusively on trusted inputs (user message + request context), which substantially reduces the adversarial surface. However, parser errors degrade security (if over-scoped) or usability (if under-scoped). The scope interpretation mode makes this tradeoff explicit and configurable rather than implicit.

Escalation fatigue. In strict scope mode, frequent escalation prompts may lead users to approve requests reflexively. The scope interpretation modes mitigate this by allowing operators to tune the escalation frequency to their security requirements, but the fundamental tension between security and usability persists. Quantitative measurement of escalation rates across scope modes and task types is an important direction for future work.

6 Implementation and Evaluation

Our reference implementation uses TypeScript, OpenFGA (self-hosted), and Claude Sonnet as the LLM backbone.

OpenFGA Model. The authorization model uses two types: `user` (subject) and `tool_invocation` (object) with a `can_invoke` relation. The tool invocation object encodes agent, tool, and resource in its identifier: `tool_invocation:{agent}:{tool}#{resource}`.

OpenFGA's conditional relationship tuples enable native TTL enforcement without application-layer pruning. Conditions attached to tuples are evaluated at check time, meaning expired tuples are automatically excluded from authorization decisions. The model also supports contextual tuples—tuples provided at check time without being persisted—which can carry runtime context such as the current turn number or request metadata.

This design leverages OpenFGA's existing support for both Relationship-Based Access Control (ReBAC) and Attribute-Based Access Control (ABAC) patterns. The flat `(user, can_invoke, tool_invocation)` relation handles the core authorization, while conditions and contextual tuples extend it to handle temporal constraints, argument validation, and other context-dependent policies without requiring a custom authorization engine.

```
model
  schema 1.1

type user

type tool_invocation
  relations
    define blocked: [user]
    define can_invoke:
```

```
      [user with within_ttl]
      but not blocked

condition within_ttl(
    current_turn: int,
    created_turn: int,
    ttl: int) {
  current_turn - created_turn <= ttl
}
```

Deny tuples are written once at deployment and apply globally:

```
// Deny policies (configuration time)
(user:*, blocked,
 tool_invocation:shell:exec#*)
(user:*, blocked,
 tool_invocation:*:*#/etc/*)
(user:*, blocked,
 tool_invocation:*:*#~/.ssh/*)

// Allow tuples (per-request)
(user:req_abc, can_invoke,
 tool_invocation:email:send#bob@company.com)
```

Agent Architecture. An orchestrator agent delegates to specialized agents (research, file, email). Each agent's tools are wrapped with `invokeToolWithAuth`. The orchestrator handles escalation bubbling—when a sub-agent returns a denied result, the orchestrator generates a concrete escalation prompt from the denied call's parameters and surfaces it to the user. The user's response is routed back through the intent parser before any new tuples are written.

Request Context Assembly. The context is assembled at the start of each request by loading the user's address book—the sole trusted data source for constraining external communication. File paths and other resource identifiers are extracted from the user's natural language input by the intent parser, not from external data sources. The context assembly is a trusted operation that does not involve the LLM agent.

6.1 Evaluation

We evaluated IBAC on the AgentDojo benchmark [6] workspace suite v1: 40 user tasks across 5 categories (calendar, email, cloud drive, contacts, and combined multi-step), with 6 injection tasks covering unauthorized email, file deletion, calendar creation, and data exfiltration. Each task was run against all 6 injection types, producing 240 injection runs per scope mode. The attack type was `important_instructions`—prompt injection embedded in tool responses. The agent LLM and intent parser both used Claude Sonnet.

6.1.1 Aggregate Results

Table 2: Aggregate benchmark results.

Mode	Utility	Security	ASR	Denial
Strict	33.3%	**100.0%**	0.0%	92.2%
Permissive	65.8%	98.8%	1.2%	37.6%

Table 3: Authorization latency.

Mode	Avg Auth	Avg Parser
Strict	8.8ms	∼5,500ms
Permissive	7.9ms	∼6,800ms

Strict mode blocked all 240 injection attempts. Permissive mode allowed 3 of 240 (1.2% attack success rate).

6.1.2 The Three Permissive Breaches

All three breaches trace to the same root cause: permissive mode granted wildcard write/send capabilities that happened to overlap with injection goals.

Table 4: Analysis of the three permissive breaches.

Task / Injection	Root Cause
user_task_8: Create event with attacker	Permissive granted `calendar|write|*` as prerequisite for "add participants"—write too broad
user_task_13: Create event with attacker	Broad cross-domain permissions for multi-step task included `calendar|write|*`
user_task_25: Forward email to attacker	Permissive granted `email|send|*` for "email TODO items to people"—covered attacker address

These breaches are the concrete evidence for IBAC's core argument: broad permissions are the vulnerability. In each case, the agent's reasoning was compromised by injection, and the overly-broad authorization surface let the compromised action through. Strict mode—which would have scoped these to specific recipients and event IDs—blocked all three.

6.1.3 Strict Mode: The Automated Floor vs. Interactive Utility

The 33.3% strict utility is the automated floor—tasks that completed with zero human interaction. This is not the ceiling. IBAC's escalation protocol (Section 4.6) is specifically designed for the gap: denied calls are security checkpoints, not failures.

Of the 16 tasks where permissive succeeded but strict failed, every failure traces to missing prerequisite capabilities:

Table 5: Missing capabilities in strict mode failures.

Missing Capability	Tasks
`file|search|*`	8
`contacts|lookup|*`	7
`calendar|search|*`	6
`file|read|*`	5

In a production deployment, each of these triggers an escalation prompt: "I need to search your calendar to find that event. Allow?" The user approves—it's an obvious prerequisite—and the task completes with full security intact. No hard judgment calls.

Contrast this with the injection case. The user asks about a dinner reservation. The agent processes a poisoned tool response and attempts to create a calendar event with `mark.black-2134@gmail.com`. The escalation prompt surfaces: "The agent wants to create an event with mark.black-2134@gmail.com. Allow this?" The user immediately recognizes this has nothing to do with their dinner question.

This reframes the utility-security tradeoff:

Table 6: Utility-security tradeoff across modes.

Mode	Util.	Sec.	Human
Strict (auto)	33.3%	100%	None
Strict + esc.	∼80%+	∼100%	Prerequisites
Permissive	65.8%	98.8%	None

IBAC does not aim for 100% automated utility. It aims for 100% security with a human-in-the-loop fallback. The denied calls are the system correctly identifying actions outside the user's stated intent and requiring explicit approval before proceeding.

6.1.4 Tasks That Failed in Both Modes

8 tasks failed regardless of scope mode, due to execution complexity rather than authorization failures (multi-step file creation, complex scheduling operations). These represent the current ceiling of the agent's task-completion ability independent of IBAC.

7 Discussion

Intent Parsing Accuracy. IBAC's security depends on the intent parser correctly identifying required capabilities at the appropriate specificity. The scope interpretation modes make this tradeoff explicit: strict mode minimizes the parser's discretion (and thus its error surface) at the cost of escalation frequency; permissive mode grants the parser more latitude, increasing the risk of over-scoping but reducing user friction. Systematic evaluation of parser accuracy across scope modes and diverse request types is an important direction for future work.

Comparison to Probabilistic Defenses. Production AI agent deployments commonly use probabilistic defenses: LLM-as-a-judge evaluation of tool calls, input content length limits, and output classifiers. These defenses have practical value and a track record in production environments. IBAC does not argue that probabilistic defenses are useless—rather, that they are insufficient as a sole defense against adversarial attacks. An LLM-as-a-judge evaluating whether a tool call "looks reasonable" is performing the same function as IBAC's intent parser, but at the tool call boundary rather than up front, and without a formal authorization model backing it. The key difference is that IBAC's authorization decisions are deterministic once tuples are written: the FGA check is a binary yes/no against a concrete tuple, not a probabilistic judgment. IBAC is best understood as a complement to existing defenses, providing a deterministic authorization layer beneath probabilistic heuristics.

Comparison to CaMeL. The most closely related work is Google DeepMind's CaMeL [6], which shares IBAC's core insight: prompt injection defense must be enforced outside the LLM's reasoning chain. Both systems derive authorization from trusted user input and both reject relying on model behavior modification. Both were evaluated on the AgentDojo benchmark, enabling direct comparison.

CaMeL converts user requests into programs in a restricted Python dialect, executes them in a custom interpreter, and tracks data provenance through capability tags on variables. This requires a dual-LLM

architecture (privileged + quarantined), a custom interpreter, and program generation for every request. IBAC takes a lighter approach: one additional LLM call (intent parsing, ∼5.5s) plus a deterministic FGA check (∼9ms per tool invocation). No custom interpreter. No dual-LLM setup. No changes to the underlying agent framework.

On AgentDojo, CaMeL achieved 77% utility with provable security. IBAC Strict achieves 33.3% automated utility (100% security) rising to ∼80% with human-in-the-loop escalation. IBAC Permissive achieves 65.8% utility with 98.8% security. The approaches optimize for different deployment constraints: CaMeL maximizes automated utility through upfront program generation; IBAC maximizes security through minimal upfront authorization with consent-based escalation.

Table 7: Comparison of IBAC and CaMeL.

	Strict	**+Esc.**	**Perm.**	**CaMeL**
Security	100%	∼100%	98.8%	∼100%
Utility	33.3%	∼80%	65.8%	77%
Arch.	1 LLM call + FGA			Dual LLM
Integr.	Wraps existing agents			Custom RT

The practical cost difference is significant. IBAC adds one LLM call per request and one ∼9ms FGA check per tool invocation to an existing agent. CaMeL requires replacing the agent's execution model with a custom interpreter and maintaining a dual-LLM architecture. For teams with existing agent deployments, IBAC's retrofit path is substantially lower-friction.

Where CaMeL is stronger: data flow provenance within tool arguments. If an authorized tool gets called with a tainted argument subfield, CaMeL's taint tracking catches it. Where IBAC is stronger: deployment simplicity, standards-based authorization (OpenFGA) with existing operational tooling, configurable scope modes, and dynamic permission escalation. A combined system would address both tool-level access control and intra-tool argument provenance.

Limitations. (1) Intent parsing is probabilistic—though it operates on trusted inputs only, parser errors can over- or under-scope the authorization set. The 3 permissive breaches in our evaluation (Section 6.1.2) demonstrate the consequence of over-scoping. (2) Resource patterns use string matching, which may over- or under-match depending on the parser's specificity and the quality of the contact store. (3) Tool invocations with multiple security-relevant arguments beyond the resource identifier are only partially constrained by the current tuple structure. (4) The current implementation does not support hierarchical capabilities or conditional permissions beyond TTL, though OpenFGA's model natively supports both. (5) The "interactive utility" estimate (∼80% for strict + escalation) is based on human classification of denied calls, not a formal user study; a controlled evaluation with human participants is needed to validate escalation approval rates.

Future Work. Controlled user study to measure interactive utility—actual escalation approval rates, time-to-approve, and escalation fatigue across scope modes and task types. A "moderate" scope mode (between strict and permissive) that grants prerequisites but not implied actions, targeting higher automated utility without the wildcard permissions that caused the 3 permissive breaches. Systematic evaluation of intent parser accuracy, including adversarial testing of the parser itself. Integration of OpenFGA's conditional tuples for richer argument constraints beyond the resource identifier. Hierarchical capability models (e.g., "web access" implying specific domains). Formal verification of the authorization model. A simulated user agent that approves prerequisite escalations but rejects out-of-scope ones, enabling programmatic benchmarking of interactive utility.

8 Related Work

Willison [7] proposed the Dual LLM pattern, separating a privileged LLM with tool access from a quarantined LLM exposed to untrusted content. Willison [8] later identified the lethal trifecta—access to private data, exposure to untrusted content, and external communication—as the combination that makes AI agents exploitable, arguing that the only safe approach is to avoid combining all three. IBAC addresses the trifecta not by removing a leg but by constraining each leg to the user's intent.

Debenedetti et al. [6] extended Willison's earlier work with CaMeL (discussed in Section 7). CaMeL's key contribution is enforcing security through principled system design rather than model behavior modification—a philosophy IBAC shares. The approaches are complementary: CaMeL provides stronger guarantees against data propagation through complex multi-step programs, while IBAC provides a more lightweight deployment path and explicit support for dynamic permission escalation.

Google's Zanzibar [3] and its open-source descendant OpenFGA [4] provide scalable relationship-based authorization. IBAC builds on this infrastructure, applying it to AI agent security—a domain these systems were not originally designed for. OpenFGA's support for conditional relationship tuples and contextual tuples enables IBAC to express temporal constraints, argument-level conditions, and context-dependent policies within the existing authorization framework.

Greshake et al. [1] taxonomize indirect prompt injection attacks. Wallace et al. [2] propose instruction hierarchy to improve LLM robustness. Neither addresses authorization-layer enforcement. RBAC grants static roles too broad for per-request scoping. Guardrails and content filters operate on the text layer and are bypassable. Sandboxing restricts the execution environment but not the agent's intent. IBAC is complementary to all of these—it operates at the authorization layer, providing defense-in-depth. Recent work on AI agent safety [5] identifies tool-calling as a key risk surface; IBAC provides a concrete, deployable defense mechanism for this surface.

9 Conclusion

The right question was never how to make AI resist manipulation. The right question was how to make manipulation irrelevant.

Intent-Based Access Control answers it by moving authorization decisions out of the AI entirely. The agent's reasoning can be fully compromised—IBAC still blocks unauthorized actions because the authorization boundary is deterministic, external, and immutable by the agent. On the AgentDojo benchmark, this held: 100% security in strict mode across 240 injection runs, with the 3 permissive breaches directly attributable to over-scoped permissions rather than authorization bypass.

The framework is practical. One additional LLM call for intent parsing. $\sim$9ms per tool invocation for authorization. No custom interpreter, no dual-LLM architecture, no changes to the underlying agent framework. For teams with existing agent deployments, IBAC provides defense-in-depth beneath probabilistic defenses, catching what the judge misses—which, against a motivated attacker, is inevitable.

The lethal trifecta—private data, untrusted content, external communication—doesn't have to be lethal. It just has to be scoped.

References

[1] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz. Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection. In *AISec*, 2023.

[2] E. Wallace, K. Xiao, R. Leike, L. Weng, J. Heidecke, and A. Beutel. The instruction hierarchy: Training LLMs to prioritize privileged instructions. *arXiv:2404.13208*, 2024.

[3] R. Pang et al. Zanzibar: Google's consistent, global authorization system. In *USENIX ATC*, 2019.

[4] OpenFGA Contributors. OpenFGA: A high-performance and flexible authorization engine, 2023. https://openfga.dev.

[5] Y. Ruan, H. Dong, A. Wang, S. Pitis, Y. Zhou, J. Ba, Y. Dubois, C. Maddison, and T. Hashimoto. Identifying the risks of LM agents with an LM-emulated sandbox. In *ICLR*, 2024.

[6] E. Debenedetti, I. Shumailov, T. Fan, J. Hayes, N. Carlini, D. Fabian, C. Kern, C. Shi, A. Terzis, and F. Tramèr. Defeating prompt injections by design. *arXiv:2503.18813*, 2025.

[7] S. Willison. The dual LLM pattern for building AI assistants that can resist prompt injection, 2023. https://simonwillison.net/2023/Apr/25/dual-llm-pattern/.

[8] S. Willison. The lethal trifecta for AI agents: private data, untrusted content, and external communication, 2025. https://simonwillison.net/2025/Jun/16/the-lethal-trifecta/.