Browser-Anchored Trust: Constraining Agentic AI to End-User Permissions in Multi-Tenant SaaS

White Paper

18th March 2026

V1.0

Patrick Cunningham

Indulge Media Ltd

1. Introduction

AI chat tools are everywhere in SaaS these days, whether users like it or not. More often than not, the AI agent is simply an encapsulation of the user documentation, with minimal or no access to actual tenant data.

However, users expect to be able to ask things like "how many appointments does Jane have this week?", or "what's the most popular service this month?", or "why didn't that reminder go out?". They want to get answers from their live business data, not a link to a help article or an apology.

The current generation of LLMs makes this possible through tool calling: the model reasons about the question, decides which data sources to query, emits structured tool requests, and synthesises a response from the results the application returns. This is an agentic loop - the same technology that drives agentic tools like Claude Code. The model acts autonomously across multiple steps, calling tools and interpreting results without a human approving each action.

This autonomy is precisely what makes agentic systems useful, however, it's also what makes them dangerous.

An agent with tool access will, in principle, use whatever tools it can reach. If the agent holds a privileged API key such as a service account, an admin token, or a broadly-scoped OAuth credential, then a single prompt injection, a hallucinated tool call, or a subtle reasoning error can expose data the user was never meant to see. In a multi-tenant SaaS application, this isn't a theoretical concern. It's the kind of event that causes a data breach or worse.

This paper describes an approach to solving this problem. The core idea is simple: the AI agent should never be more powerful than the user who summoned it.

The approach relies not on prompt engineering or LLM guardrails, but on a structural authentication mechanism that makes privilege escalation architecturally impossible. The agent's credentials are minted by the user's browser, scoped to the user's permissions, and validated against live data on every single request.

This pattern is referred to here as *browser-anchored trust*.

The individual techniques - HMAC-signed tokens, short TTLs, permission re-derivation, read-only API surfaces - are established, well-understood patterns. What this paper offers is a

specific composition of these techniques applied to the problem of plumbing an AI agent into a live application API in a way that's tightly controlled, verifiably secure, and structurally immune to prompt injection attacks.

2. Problem Statement

The Service Account Anti-Pattern

The most common approach to giving an AI agent data access is to create a service account in the form of an API key, a set of OAuth client credentials, or an internal token with broad read permissions. The agent authenticates with this credential, and access control is enforced (or more often, hoped for) at the prompt level: "You are an assistant for Tenant X. Only access data for this tenant."

This is fundamentally broken for three reasons.

First, prompt instructions are not a security boundary. Prompt injection - whether from malicious user input, poisoned retrieval results, or adversarial tool outputs - can override system instructions. If the agent's credential grants access to all tenants, no amount of prompt engineering prevents cross-tenant data exposure.

Second, service accounts operate outside user context. A service credential has no notion of "the current user" or "the current user's role." Any access control has to be re-implemented in the agent layer, duplicating the permission logic that already exists in the application. This duplication is error-prone and inevitably drifts from the source of truth.

Third, service accounts are a high-value target. A single leaked API key compromises every tenant in the system. Credentials like this need to be rotated regularly, and audit trails are coarse-grained at best.

This is a textbook instance of the Confused Deputy Problem (Hardy, 1988). The LLM is the deputy: it receives a user's question but acts with the service account's credentials, which may span every tenant and every permission tier. The architecture gives it no way to distinguish "what the user is allowed to do" from "what the credential permits." Any approach that hands the agent a credential more powerful than the user is, by construction, creating a confused deputy.

Multi-Tenant Complexity

Multi-tenant SaaS applications typically enforce several layers of access control:

- Tenant isolation: User A at Tenant 1 must never see Tenant 2's data.
- Role hierarchy: Owner > Admin > Manager > Staff > Receptionist, each with different permissions.
- Resource scoping: Some users can only see their own records (for example, a staff member's own schedule).
- Dynamic permissions: Roles can be changed, users can be deactivated, tenant access can be revoked, all at any time.

An AI agent operating within this system has to respect all of these constraints, and it has to do so in real time. A token issued 5 minutes ago may already be stale if an admin has since downgraded the user's role.

The Core Question

How do you give an AI agent enough access to be genuinely useful; querying things like customers, appointments, settings, notification history, whilst preventing the agent from accessing anything the current user cannot?

3. Architecture Overview

The solution separates the system into two distinct authentication boundaries with no overlap (Figure 1):

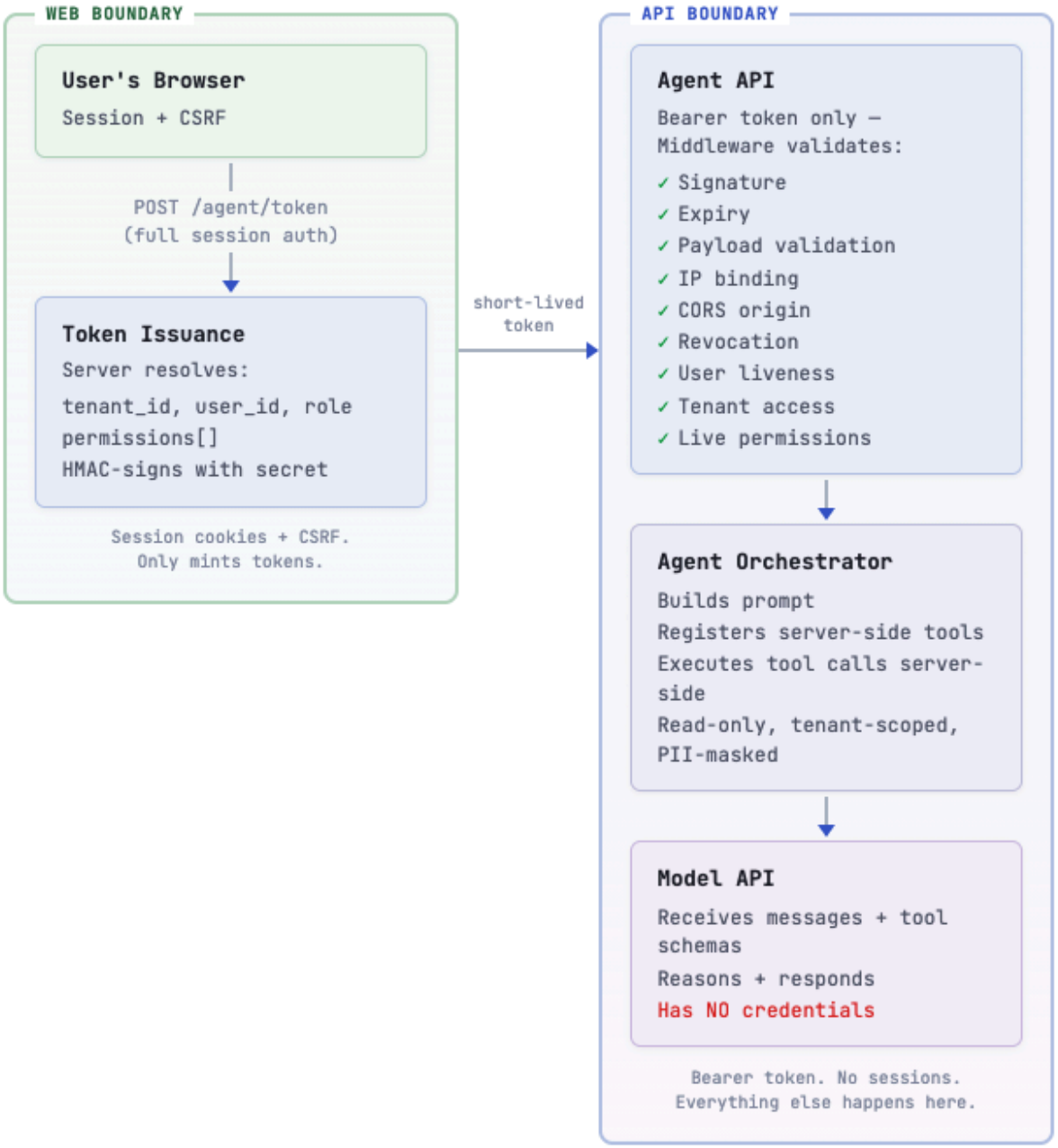

Figure 1: Authentication boundaries. The web boundary (left) mints tokens via session auth. The API boundary (right) validates tokens and executes tool calls. The two never overlap.

The critical separation:

- The web boundary (session cookies, CSRF protection) handles one thing only: minting tokens. The user's browser is the only client that ever touches this endpoint.
- The API boundary (bearer token) handles everything else: creating chat sessions, sending messages, streaming responses. These endpoints know nothing about sessions or cookies.
- The LLM itself has no credentials whatsoever. It receives tool definitions and gets results back. The credentials live in the middleware layer that wraps tool execution, which the LLM cannot see, influence, or bypass.

4. The Token Issuance Mechanism

Token issuance is the linchpin of the entire system. It's the only point where user identity gets converted into agent capability, and it's deliberately constrained.

The Endpoint

A single endpoint - for example `POST /agent/token` - protected by full session authentication: valid session cookie, valid CSRF token, completed 2FA challenge (if enabled), and OAuth session (if applicable). There's no API key, no client credential, no alternative path. If you don't have the user's authenticated browser session, you can't mint a token.

What the Server Does

When the endpoint is hit, the server identifies the authenticated user, resolves their current tenant context, determines their role, and constructs a payload:

```JSON
{
  "tenant_id": 42,
  "user_id": 17,
  "role": "staff",
  "expires_at": 1741700000,
  "issued_at": 1741699400,
  "ip": "203.0.113.42"
}
```

Note the absence of permissions in the payload. Since permissions are re-derived from the database on every request (see Layer 9), embedding them in the token would be redundant - and worse, would invite consumers to trust stale data.

- Signs the payload with HMAC-SHA256 using the application's server-side secret.
- Returns the token: `[base64(payload)].[signature]`.

Why Opaque Identity Tokens, Not Self-Describing Claims

This pattern uses HMAC-signed opaque tokens rather than self-describing alternatives like JWTs. The objection isn't to HMAC as a signing mechanism - it's to the claims-based model that encourages consumers to treat embedded permissions as authoritative. The token here carries no permissions at all - it's purely an identity assertion. Permissions are derived from live data on every request (see Section 5, Layer 9).

Database-backed personal access tokens can be created programmatically; OAuth tokens can be exchanged server-to-server. HMAC tokens require the signing secret (which never leaves the server) and a browser session (which can't be synthesised). The result is a stateless, unforgeable token with no database table to query, no token store to compromise, and no programmatic creation path.

Short TTL and Auto-Refresh

Tokens expire after 10 minutes. The frontend automatically requests a new token 2 minutes before expiry, ensuring seamless continuity during active conversations whilst keeping the window of validity small.

This TTL is a deliberate tradeoff. A shorter TTL would reduce the window further but increase refresh traffic and risk interrupting active SSE streams. A longer TTL would reduce overhead but leave stale permissions in play for too long. 10 minutes, with live permission re-derivation on every request, strikes the balance.

5. The Validation Middleware (9-Layer Defence)

Every API request to the agent endpoints passes through a validation middleware that performs 9 sequential checks. If any check fails, the request is rejected immediately. There's no fallback, no degraded mode, no partial access.

Layer 1: Signature Integrity

The middleware splits the token into payload and signature, recomputes the HMAC-SHA256 using the server's secret, and compares. A single bit of tampering - modifying the tenant ID, adding a permission, extending the expiry - invalidates the signature, rendering the request invalid.

Layer 2: Expiry

The `expires_at` timestamp is checked against the current server time. Expired tokens are rejected regardless of any other factor.

Layer 3: Payload Validation

The decoded payload must contain all required fields (`tenant_id`, `user_id`, `role`, `expires_at`, `issued_at`, `ip`) with valid types. Malformed tokens are rejected.

Layer 4: IP Binding

The middleware compares the requesting client's IP address against the `ip` field embedded in the token payload. If they don't match, the request is rejected. This prevents a stolen token from being used from a different network location - the attacker would need to be on the same IP as the victim, which eliminates the most common exfiltration scenarios (XSS token theft used from an attacker-controlled server, leaked tokens in logs or error reports, tokens intercepted in transit).

IP binding requires the server to resolve the real client IP by explicitly declaring trusted proxies and rejecting forwarded headers from all other sources.

Layer 5: CORS Origin Restriction

The agent API enforces a strict CORS policy: only requests originating from the application's own domain are permitted. The allowed origin is a single, explicit value - the application's primary URL - not a wildcard, not a pattern, not a list of "trusted" domains.

This is the browser-side complement to IP binding. Even if an attacker obtains a valid token, they can't use it from a browser on a different domain - the preflight check fails, and the browser never sends the request. Combined with IP binding (which blocks server-side use

from a different network), the two layers close both client-side and server-side exfiltration paths.

Layer 6: Logout Revocation

When a user logs out, the application writes a revocation timestamp keyed by user ID, with a TTL matching the token's maximum lifetime (10 minutes). The middleware checks this revocation marker and rejects any token whose `issued_at` predates the revocation timestamp.

This means logging out immediately invalidates all outstanding agent tokens, even if they haven't yet expired. The user doesn't need to manually revoke anything, the act of logging out is sufficient.

Layer 7: User Liveness

The middleware loads the user from the database and verifies:

- The user record still exists (not deleted).
- The account status is active (not suspended or deactivated).

A deactivated user's tokens stop working immediately, on the very next request.

Layer 8: Tenant Access

The middleware verifies the user still has access to the tenant encoded in the token:

- For owners: they must still own the tenant, and the tenant must be active.
- For staff: they must have an active profile at the tenant.

If a user is removed from a tenant between token issuance and the next request, access is denied.

Layer 9: Just-In-Time Authorisation (Live Permission Re-Derivation)

This is the most important layer. The token carries no permissions, so there are none to trust or distrust. Instead, the middleware queries the database for the user's current role and computes their effective permissions from scratch on every request. These freshly-derived permissions are what get attached to the request context and passed to the tools.

The token, therefore, functions purely as an identity assertion: it binds a specific user and tenant context to an authentication event at time T. What that user is allowed to do is determined fresh, every time - so a role downgrade takes effect at the next API call, not at the next token refresh.

6. A Curated, Read-Only Tool Surface

The agent's capabilities aren't defined by what the LLM can imagine. They're defined by a closed, curated set of tools.

The Tool Set

Each tool should be carefully considered and whitelisted explicitly, and preferably read-only. The skills of your agent are tightly bound to the tools you surface. The LLM sees only the tool names, descriptions, and parameter schemas - not application code, database connections, or framework internals. When the model calls a tool, the dispatcher resolves it against the pre-registered whitelist and executes the matching read-only handler inside the current request context.

As an example, a booking management application might expose the following 9 tools:

Tool	Purpose	Permission Required
Search documentation	Semantic search of user guides (RAG)	None
Find customer	Search customers by name, email, or phone	`view-customers`
Get customer details	Full profile with recent activity	`view-customers`
Get appointments	Filter by customer, staff, date, status	`manage-appointments`
Get appointment details	Full record with pricing	`manage-appointments`

Get settings	Tenant configuration (timezone, currency, tax)	`view-settings`
Get team members	Active staff list	`view-staff`
Get services	Service catalogue (name, price, duration)	`view-services`
Get notification logs	Delivery history for emails and SMS	`manage-notifications`

Every tool that returns tenant data checks the user's permissions before executing. If a staff member lacks `view-customers`, the tool returns an explicit permission error that the LLM can relay to the user.

Tenant Scoping

Every database query is filtered by `tenant_id` - a hard-coded filter, not enforced by the LLM's reasoning. The agent can't construct a cross-tenant query because the tools don't accept a tenant parameter. The tenant ID comes from the validated request context.

PII Masking

Even within permitted access, where highly confidential data or PII is being processed, sensitive data can be masked in tool output:

- Phone numbers: `***1234`
- Email addresses: `j***@domain.com`

When applied at the tool layer, this masking happens before the data reaches the LLM, so that the model never sees the full values.

No Escape Hatches

The agent has no access to:

- The filesystem.

- The network (beyond its pre-defined tools).
- Shell commands.
- Database queries outside the tool definitions.
- Any write operations whatsoever.

There is no generic "run command", "make HTTP request", or "execute query" tool. Assistant text is rendered back to the user. Only server-registered tool handlers can execute. The tool surface is the agent's entire world.

7. Preventing Privilege Escalation

Privilege escalation in an agentic system can happen through several vectors. The following table maps each vector to the architectural mechanism that prevents it:

Attack Vector	Prevented By	Section
Agent mints its own tokens	Token issuance requires browser session (cookie + CSRF + optional 2FA). No such tool exists; signing secret not accessible from API layer.	4
Agent modifies token payload	HMAC-SHA256 signature invalidated by any change to payload. Middleware rejects before further processing.	5, Layer 1
Agent outlives the user session	Hard 10-minute TTL + logout revocation via cache key that invalidates all tokens issued before logout timestamp.	4, 5 Layer 6
Agent uses stale permissions	Permissions re-derived from database on every request. Role changes take effect at the next API call, not at next token refresh.	5, Layer 9
Agent crosses tenant boundaries	Tenant ID extracted from validated token and injected into every query. Tools don't accept a tenant parameter from the LLM.	6

Agent mutates state	No write tools exist. Tool surface is entirely read-only. Worst case is information disclosure bounded by user's own permissions.	6
Stolen token used remotely	IP binding (Layer 4) + CORS restriction (Layer 5) block both server-side and browser-side use from a different origin.	5, Layers 4–5
Brute-force data exfiltration	Rate limiting (configurable, default 10 req/min per user) prevents bulk extraction before token expires.	5

Chained Attack Analysis: Session Spoofing and Token Forgery

A sophisticated attacker might attempt to chain multiple exploits via prompt injection: spoof a browser session, create a database-backed token, and use it to mint a new HMAC token with elevated privileges.

This attack fails because it requires chaining 5 capabilities - credential discovery, arbitrary HTTP requests, database writes, session construction, and CSRF token generation - none of which the agent possesses.

Each step is independently impossible given the closed tool surface: the agent has no filesystem access to find credentials, no outbound HTTP capability to establish sessions, no write tools to create database records, no access to session storage or encryption keys, and no path to CSRF tokens without a valid session.

Even a fully compromised system prompt produces nothing, because the tools to execute any step simply don't exist.

This is the core value of a closed tool surface: security isn't a property of the LLM's behaviour, but of the absence of dangerous capabilities.

8. The Agentic Loop Under Constraints

With the security model in place, the agentic loop itself operates within well-defined bounds.

What Actually Happens on Each Message

The following sequence diagram traces a single user message through the full request lifecycle, from token acquisition through tool execution and back to the browser.

Each arrow represents a real HTTP boundary crossing or internal dispatch, and every step between the user's browser and the model API passes through the validation middleware described in Section 5. The tool dispatch loop in the centre may repeat multiple times within a single response as the model chains tool calls to answer a complex question.

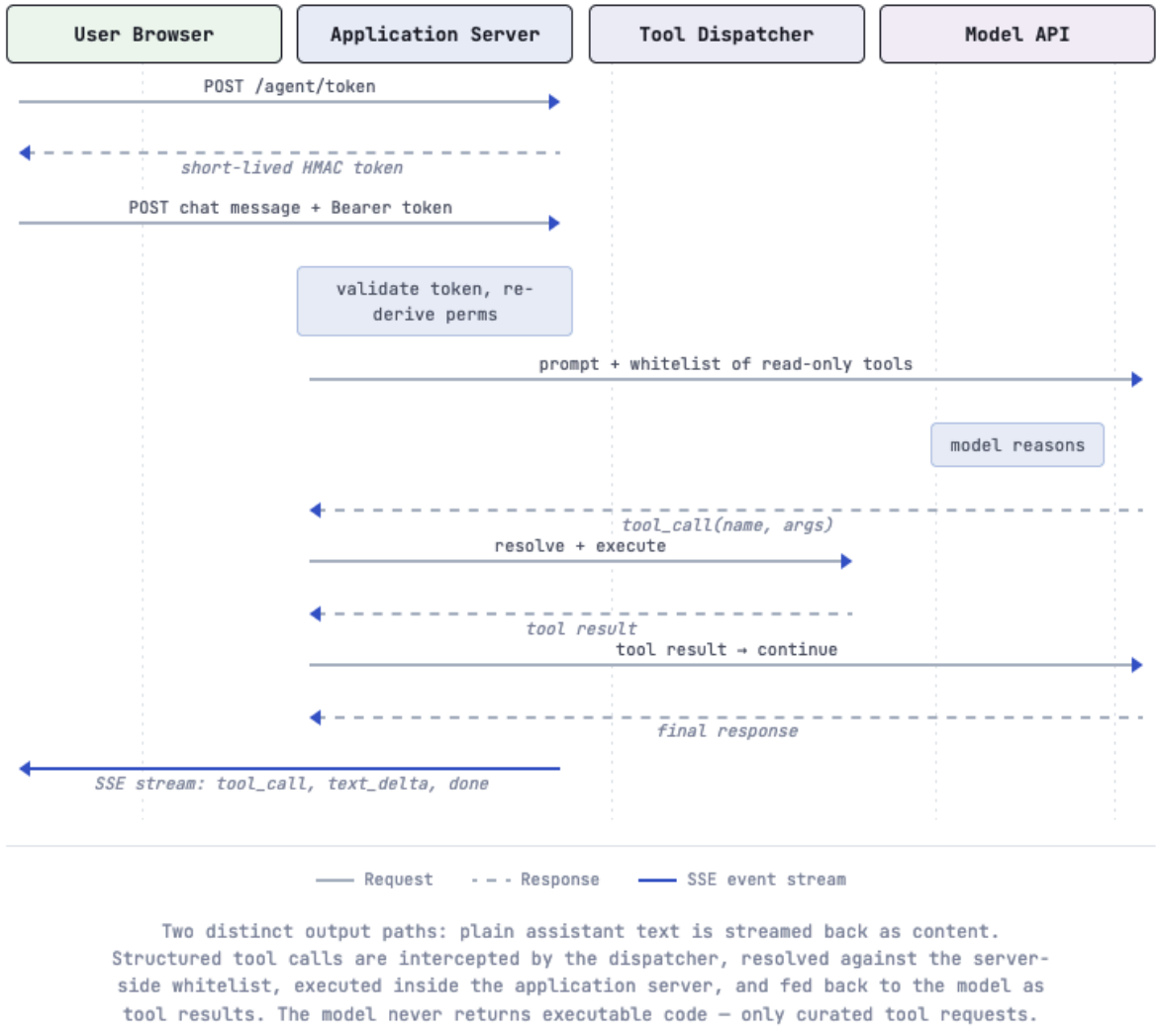

Figure 2: The agentic loop sequence. Token validation and permission re-derivation occur before every tool execution. The model never holds credentials.

Two distinct output paths exist from the LLM. Either plain assistant text is streamed back to the browser, or structured tool calls are intercepted by the dispatcher, resolved against the server-side whitelist, executed inside the application server, and fed back to the model as tool results. The model can only request pre-registered tools. If no matching tool exists, nothing executes.

Bounded Computation

The LLM is configured with a maximum step count (e.g., 10 tool calls per message) and a maximum token output limit. These prevent runaway loops and excessively verbose output.

Structural Enforcement, Not Prompt Reliance

The system prompt instructs the agent to be helpful, to search documentation before querying live data, and to never fabricate information. But these instructions aren't the security boundary. Every constraint that matters - permission checks, tenant scoping, read-only access, token validation - is enforced in application code that the LLM can't see, influence, or bypass.

If the system prompt were completely removed, the security properties would remain intact. The agent might behave less helpfully, but it could not access data the user isn't permitted to see.

Real-Time Visibility

Responses are streamed to the frontend via Server-Sent Events (SSE) with typed event categories:

- `tool_call`: The agent is invoking a tool (visible to the user in real time).
- `text_delta`: Incremental text output.
- `tool_result`: A tool has returned results.
- `message_done`: The response is complete.
- `error`: Something went wrong.

This gives the user confidence that the agent is querying real data and creates an implicit audit trail. It also allows the frontend to render each event type with structured, consistent UI components - tool calls, results, and errors can each have their own visual treatment rather than being flattened into plain text or LLM-generated markdown.

Contextual Identity Injection

When a staff member asks about their own appointments, the system prompt includes their staff profile ID - injected before the first tool call. This reduces the chance of confusion attacks where the model might be tricked into querying another user's data via a manipulated self-referential request. Even if such an attack succeeded, the permission and tenant scoping layers would prevent access to any data the user isn't already authorised to see.

RAG as the Safe Default

The documentation search tool requires no permissions and serves as the agent's preferred first step for how-to questions. The system prompt guides the model to search documentation before querying live data, reducing unnecessary data access and providing faster answers for common questions. The documentation could be stored as plain text, markdown or vector embeddings for larger repositories.

9. Extending to External Autonomous Agents

Everything described so far assumes a tightly controlled loop within a single process. But the real test of a security model is whether it holds when those assumptions are relaxed. Can a browser-anchored token be handed to an autonomous agent with broad capabilities - internet access, shell commands, filesystem operations, plugins - and still remain safe (Figure 3)?

Yes - provided the agent can't access the application's internal infrastructure.

Why the Security Model Holds

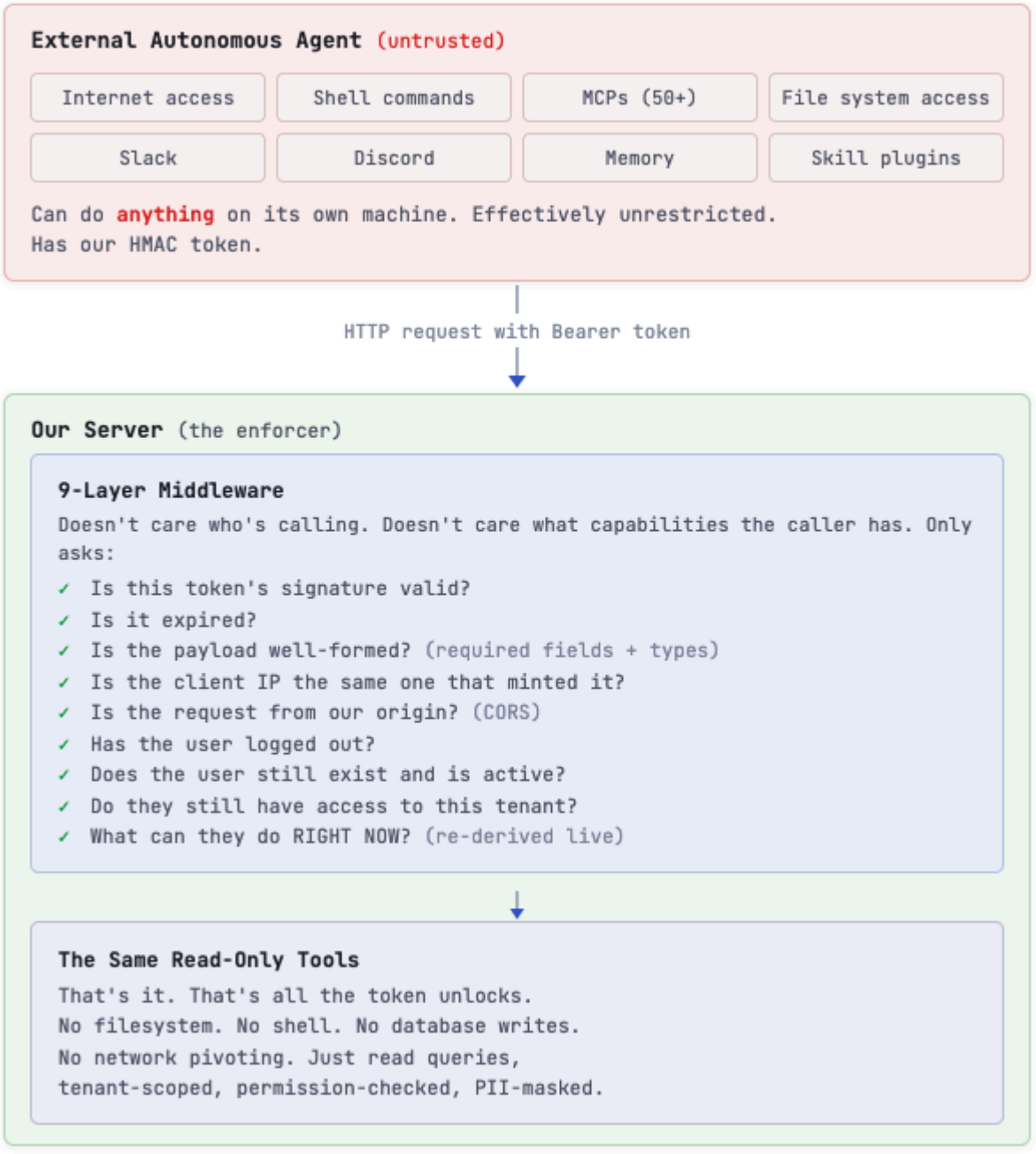

Figure 3: External agent boundary. An untrusted agent with broad capabilities hits the same 9-layer middleware and the same read-only tools as the internal agent.

The *browser-anchored trust* model secures the API boundary, not the agent. When an external agent makes a request, the same 9-layer validation middleware applies. The agent's capabilities on its own machine are irrelevant to the application's API - there's no "execute shell command" endpoint, no "write to database" endpoint. They don't exist. The worst case: the agent reads data the user can already see, with PII masked, rate-limited, for a maximum of 10 minutes before the token expires.

IP binding (Layer 4) introduces an additional constraint: the agent must make requests from the same IP as the user's browser that minted the token. In practice, the agent must run on the same network or behind the same NAT/proxy. An agent on a different network would need the issuance flow adapted - for example, explicit delegation to a known external IP, or relaxed IP binding for tokens flagged for external use. This is a deliberate tradeoff: IP binding makes casual token theft useless at the cost of some deployment flexibility.

The Deployment Constraint: Infrastructure Isolation

This guarantee has one precondition: the agent must not have access to the application's internal infrastructure. Specifically, it must not be able to read:

- The application's environment variables (which contain the HMAC signing secret).
- The database (which contains all tenant data directly).
- The session storage (which would allow impersonating logged-in users).

If an agent runs on the same server with access to the application's filesystem, the HMAC token becomes irrelevant - it could read the signing secret and forge tokens, or query the database directly. This isn't a weakness of *browser-anchored trust*; it's true of every authentication system.

The solution is standard infrastructure isolation (Figure 4):

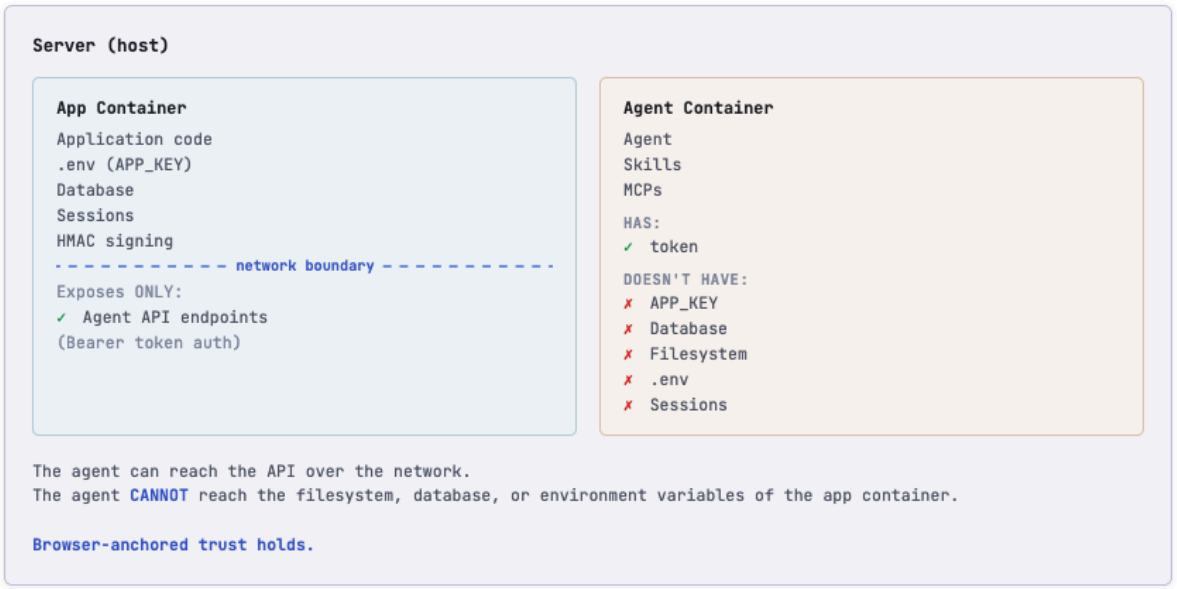

Figure 4: Container isolation. The agent container has a token but no access to the application's secrets, database, or filesystem.

With this in place, the agent can only reach the API over the network, and the token + middleware + tool surface enforce the same constraints.

Implications for Agent Architecture

The security model is agent-agnostic. You don't need to audit the external agent's codebase, vet its plugins, or trust its prompt engineering. Teams can swap agent runtimes, add skills, and connect new data sources - the token issuance, validation, and tool surface remain unchanged. The security review for a new agent deployment reduces to a single question: can this agent access anything beyond the application's API?

10. What This Pattern Achieves

Operational Simplicity

The entire authentication mechanism is:

- One token issuance endpoint.
- One validation middleware.

- One cache key for logout revocation.
- Zero database tables for token storage.

There are no token refresh endpoints, no OAuth redirect flows, no webhook callbacks, no token introspection APIs. The simplicity is deliberate - fewer moving parts mean fewer things that can fail or be misconfigured.

11. Comparison with Alternative Approaches

Approach	Token Origin	Escalation Risk	Tenant Isolation	Permission Freshness	Operational Overhead
Service API key	Server config	High - shared privileged credential	Manual query scoping	Static	Key rotation, audit
OAuth client credentials	Server-to-server	High - no user context	Manual query scoping	Static (grant-time)	Token refresh, scope management
Personal access tokens	User-created (or programmatic)	Medium - can be created via API	Token-scoped	Stale (grant-time)	Token storage, cleanup
Session pass-through	Browser session directly	Low - but couples agent to session lifecycle	Session-scoped	Live	Session affinity, CSRF complexity
HMAC browser-anchored	Browser session (mint only)	Minimal - cannot self-issue	Baked into payload +	Live (re-derived	Minimal - stateless

			query layer	per request)	

Token Issuance Comparison

Approach	Can the agent self-issue?	Can it be created programmatically?	Permission freshness
Service API key	N/A - shared credential	N/A	Static
OAuth client credentials	Yes	Yes	Grant-time
Database-backed personal access tokens	No	Yes (with DB access)	Grant-time
HMAC browser-anchored	No	No	Live (every request)

Risk Vector Comparison

The tables above compare architectural properties. The following table examines specific attack scenarios and their blast radius under each approach:

Risk Vector	Service API Key	OAuth Client Credentials	Personal Access Tokens	Session Pass-Through	HMAC Browser-Anchored
Prompt injection	Full system access - all tenants, all data	Full scope of OAuth grant - potentially all tenants	Scoped to issuing user, but permissions	Scoped to session user, live permissions	Scoped to session user, live permissions, read-only

			frozen at grant time		
Token theft	Permanent access until rotation	Access until token expiry (often 1hr+)	Long-lived - days to months	Session cookie theft gives full account access	IP-bound, 10-minute window, read-only, auto-revoked on logout
Role change (user downgraded)	No effect - service key has fixed privileges	No effect until re-authorization	No effect until token reissued	Immediate (session checks live)	Immediate (re-derived every request)
Multi-tenancy breach	All tenants exposed via shared credential	Depends on scope - often over-broad	Single tenant, but token may outlive access	Single tenant, live	Single tenant, baked into payload + query layer
Credential leak	Total compromise - every tenant, every operation	Broad compromise - server-to-server access	Single user compromise, long-lived	Session fixation risk	IP-bound, 10-minute window, single user, read-only

The session pass-through approach is the closest alternative to this pattern. Passing the user's session directly to the agent API would achieve similar permission properties, but it couples the agent to the session lifecycle and makes it difficult to use stateless API patterns like SSE streaming. The HMAC token acts as a bridge: it carries the user's identity without carrying the session's state.

12. Limitations and Future Work

Single Root of Trust

The HMAC signing secret is the sole trust anchor. If compromised, an attacker can forge tokens for any user. This is the same trust model as session cookies (also signed with this key), so it doesn't introduce new risk - but it doesn't reduce it either. Standard key management practices apply.

Read-Only Constraints

The read-only tool surface limits the agent's utility. Users will inevitably want the agent to take actions: "reschedule Jane's appointment to Thursday," "send a reminder to all clients this week."

A natural extension is write operations with confirmation loops: the agent proposes an action, the user confirms in the UI, and the action executes through the user's session, not the agent's token.

Sensitivity-Tiered Tool Access

Not all read operations carry equal risk. Future work will explore sensitivity tiers where high-sensitivity operations (financial summaries, payroll, detailed contact information) require a step-up authentication challenge - such as a 2FA confirmation or biometric prompt - before the agent can execute them. This extends the *browser-anchored* principle: the user's active participation gates access to sensitive data, even when their role already permits it.

Audit Logging

Tool calls are currently stored as JSON metadata within chat messages. A dedicated audit log with indexed fields would improve compliance posture for regulated industries, supporting structured queries like "which users accessed customer data through the agent this month?"

TTL Tuning

The 10-minute TTL is a reasonable default, but different deployments may benefit from different values. Making this configurable per tenant or per role is a natural extension.

Client-Side Permission Awareness

The frontend currently treats the token as opaque. Exposing a permission summary could enable proactive UX: greying out suggestions the agent can't fulfil, or adjusting the system prompt to avoid tools the user lacks permissions for.

13. Conclusion

Building an AI agent into a multi-tenant SaaS application isn't primarily an LLM problem. It's an authentication and authorisation problem.

1. This pattern - *browser-anchored trust* - addresses it through 5 structural mechanisms:
2. Anchor token issuance to the browser session.
3. Bind tokens to their origin via IP binding and strict CORS.
4. Enforce short TTLs with logout revocation.
5. Re-derive permissions from live data on every request.
6. Restrict the tool surface to read-only, permission-checked, tenant-scoped operations.

The result is an agentic loop running within strict guardrails that don't depend on the LLM behaving correctly. The security model is structural, not aspirational.

This pattern isn't specific to any particular LLM, cloud provider, or application domain. Any multi-tenant SaaS deploying an in-app AI assistant can adopt *browser-anchored trust*. The cost of getting this wrong is measured in breached tenants. The cost of getting it right is a few hundred lines of middleware and a principle worth defending: the agent should never be more powerful than the user who summoned it.