The Nexus Protocol: Standardizing Identity and Connection Orchestration for Autonomous Agents

Sangalo Mwenyinyo
mwenyinyo@gmail.com
CTO @ Prescott Data

Google Developer Expert; Modern & Enterprise Infrastructure

Abstract: In the emerging era of Autonomous Agents, software is moving from Chat to Action. Agents must interact with a multitude of external APIs, SaaS platforms, and cloud services to perform useful work. However, the existing authentication infrastructure is designed for interactive humans (SSO, 1Password) or static servers (hardcoded secrets) which is fundamentally ill-suited for dynamic, long-running agent fleets. This mismatch leads to the "N+1 Problem," where every new integration requires bespoke authentication logic, creating security risks and development bottlenecks. This paper introduces the Nexus Protocol, a standardized open protocol for Agent Identity and Connection Orchestration. Nexus decouples the *Authentication Mechanics* (headers, signatures, tokens) from the *Agent Logic*. By delegating identity management to a central Authority, agents become Universal Adapters, capable of connecting to any service without code changes, driven entirely by server-side policy.

1. Introduction

The shift towards autonomous microservice architectures has exposed a critical infrastructure gap: There is no standard for Agent Identity. While AI models have become capable of reasoning and planning, they lack a secure, trusted way to prove "who they are" to the outside world. We are trying to build autonomous fleets on top of authentication infrastructure designed for interactive humans (SSO) or static servers (hardcoded secrets) [5]. Unlike general workload identity standards that provide short-lived credentials via attestation [9][10], agents require more dynamic delegation and provider-agnostic orchestration. This Identity Crisis manifests in three critical ways that prevent agents from moving from prototype to production: integration complexity, fragile autonomy, and zero trust compliance [1].

Because there is no standard identity layer, every new integration requires bespoke authentication logic. A single business process may require an agent to interact with Google (OAuth2), AWS (SigV4), and a legacy CRM (API Key). For an agent to talk to “N” services, developers currently write “N” different authentication implementations. This "N+1 Problem" results in a combinatorial explosion of maintenance overhead and security surface area. This architectural rigidity directly compromises agent autonomy. While best practices advocate externalizing raw secrets (e.g., via vaults or secret managers) to enable seamless rotations without redeployment, the provider-specific authentication mechanics, such as OAuth refresh flows, request signing algorithms, or custom header injections are frequently hard-coded into the agent’s runtime. Changes to these mechanics (e.g., a provider updating its auth scheme) or adding new integrations still often require code modifications and full redeployments, forcing manual intervention on supposedly autonomous agents [5]. Beyond fragility lies a systemic security risk: as secrets and logic are decentralized across agent fleets, the attack surface expands linearly with every deployment. In the current landscape, leaked environment variables or compromised processes remain the primary vulnerability in agent frameworks. To survive in production, agents must transition away from 'Permanent Ownership' of master secrets and toward a model of Leased Identity where credentials are granted dynamically, scoped narrowly, and subject to instantaneous revocation [2][6]. Nexus Protocol solves this Crisis of Trust.

2. Agent Identity Orchestration

The first pillar of the Nexus Protocol is Identity Orchestration, the specialized process of securely acquiring, verifying, and isolating an agent's permission to act. Its primary objective is to resolve the Delegation Problem: the secure transfer of authority from a Human Principal to an Autonomous Agent. Unlike traditional service accounts, which often possess permanent and over-privileged access, an Agent Identity under the Nexus Protocol must be granularly scoped, time-bound, and strictly attributable to a specific user context. Identity Orchestration defines the rigorous cryptographic entities and workflows required to mint these ephemeral identities, ensuring that autonomy never comes at the cost of accountability. This architecture is best understood through the definition of three core system roles:

1) The Authority (Nexus): A centralized service that holds the master secrets, manages OAuth flows, and acts as the source of truth for all connections.
2) The Agent (Client): The autonomous runtime. It is dumb regarding authentication; it acts as a container for identity but does not manage the lifecycle itself.
3) The Connection: An opaque, persistent handle (e.g., UUID) representing an authorized link between a User and a Provider. It is decoupled from any specific Agent instance, allowing a single Connection to be shared across a fleet of agents subject to the Authority's policy. It moves through a strict lifecycle:

   PENDING → ACTIVE ↔ ATTENTION → REVOKED | EXPIRED | FAILED

2.1 The Nexus Handshake (Control Plane)

The Nexus Handshake is the foundational Control Plane interaction of the protocol. It is an asynchronous, three-party exchange involving the Agent (Initiator), the Authority (Mediator), and the User (Delegator).

The handshake’s primary objective is to securely establish a persistent Connection; a cryptographically bound link between a User entity and an external Provider without exposing long-term secrets to the Agent or requiring the Agent to implement provider-specific authorization flows. The Handshake is defined as a linear state transition moving a connection from PENDING to ACTIVE. This sequence of events happens in 3 phases:

A. Phase 1: Initiation (Intent Binding)

The sequence begins with the Agent declaring its intent to establish a connection. This phase cryptographically binds the request to a specific tenant context (e.g Workspace ID, User UUID) to prevent cross-tenant injection attacks. The steps to run through this phase are as follows:

1) The Agent issues a Connection Request to the Authority.

    a) Payload: Provider Identifier, Requested Scopes, Tenant/User, Context, and Return URL.

2) The Authority generates a secure state parameter. Unlike traditional OAuth, where state is often a random nonce, the Nexus Protocol mandates a Signed Context Payload:

    a) payload = base64({ tenant_id, provider_id, timestamp, nonce })
    b) signature = HMAC-SHA256(payload, AUTHORITY_MASTER_KEY)
    c) state = payload + "." + signature

3) The Authority creates a pending Connection record and returns:

    a) connection_id: An opaque, persistent identifier (UUID).
    b) auth_url: The targeted URL for user delegation.

B. Phase 2: Delegation (User Consent)

The Agent delegates the complexity of user interaction to the Authority. The User interacts only with the Authority or the upstream Provider, never directly with the Agent's credential handling logic. This phase proceeds as follows:

1) The User is directed to the auth_url.
2) The Authority executes the capture flow appropriate for the provider type:

    a) OAuth 2.0 / OIDC: The Authority acts as the relying party, redirecting the user to the Provider's authorization endpoint [3].
    b) Static Credentials (Schema-Driven): For API Keys or non-standard flows, the Authority renders a dynamic capture interface based on the provider's defined JSON Schema (e.g., prompting for api_key and region).

3) Upon completion (via callback or form submission), the Authority validates the state parameter.

   a) Integrity Check: The HMAC signature is verified to ensure the context has not been tampered with.
   b) Binding Check: The nonce is validated against the pending connection to prevent replay attacks.

C. Phase 3: Activation (Credential Vaulting)

Once consent is verified, the Authority finalizes the secure link as follows:

1) The Authority exchanges any temporary artifacts (Authorization Code) for long-lived artifacts (Refresh Tokens, Access Tokens).
2) All secrets are serialized into a generic credential map and encrypted at rest (e.g., using AES-GCM) within the Authority's Vault.
3) The Connection status transitions to ACTIVE
4) The User is redirected to the Agent's return_url with the connection_id and a success status, signaling that the Agent may now enter the Data Plane and request credentials.

The Nexus Handshake yields three foundational security benefits that reinforce the protocol's integrity through architectural design. First, it ensures the structural decoupling of the Agent from sensitive provider artifacts; by interacting exclusively with a pre-validated connection_id, the Agent is never exposed to user passwords or temporary authorization codes [4]. Second, the enforcement of a Signed State provides a cryptographic guarantee that a connection initiated by one tenant cannot be hijacked or fulfilled by another. By verifying the integrity of the HMAC-SHA256 signature, the Authority mathematically mitigates Login CSRF and session fixation attacks, ensuring strict tenant isolation within multi-tenant environments. Finally, the handshake establishes a uniform abstraction layer that ensures total protocol agnosticism. This allows the interface to remain identical for the Agent regardless of whether the underlying provider utilizes OAuth 2.0, OIDC, SAML, or static credentials captured via JSON Schema.

Figure 1: Nexus Handshake Sequence

This diagram illustrates the flow between the Agent, Authority, User, and Provider. It highlights the Control Plane separation and the specific point where the Signed Context Payload is generated.

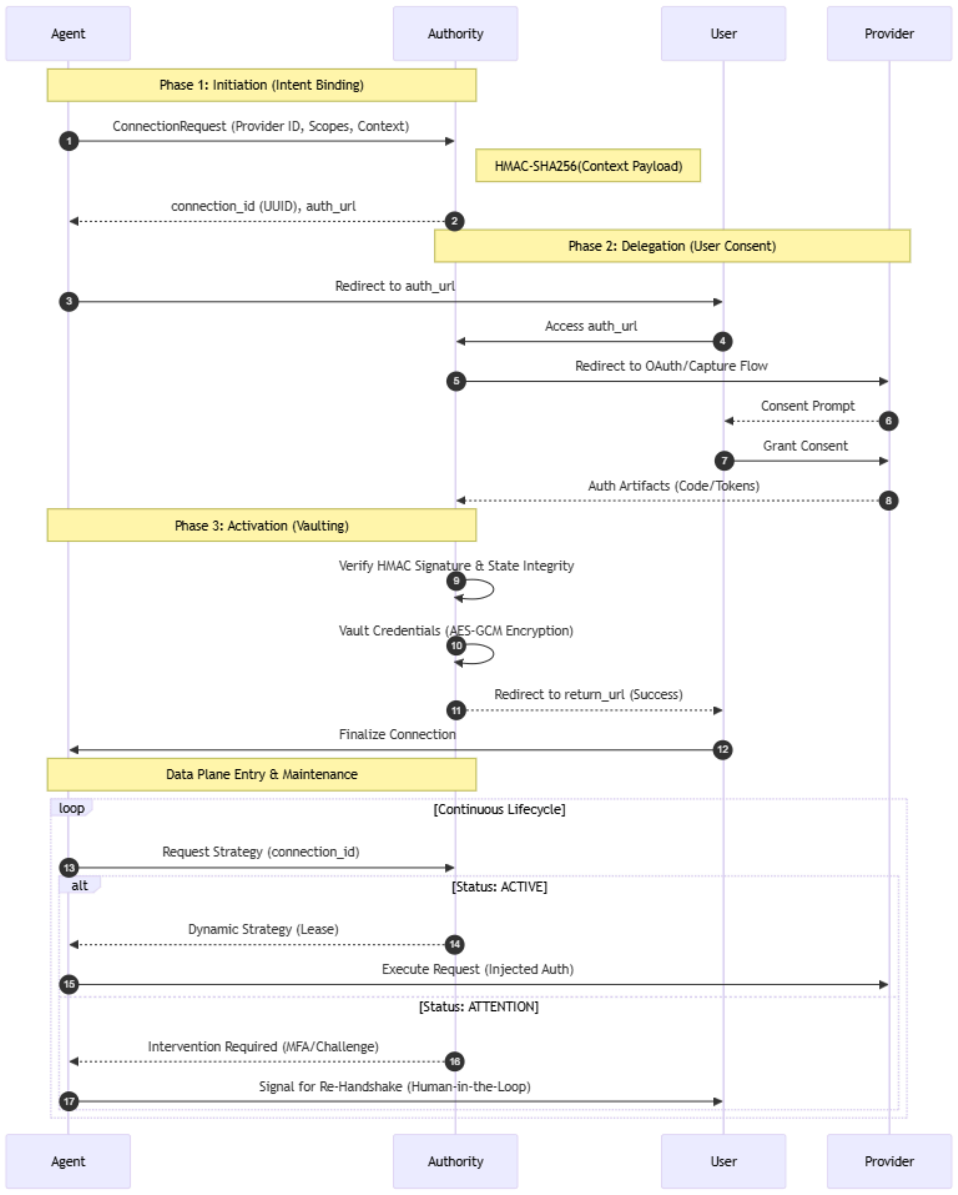
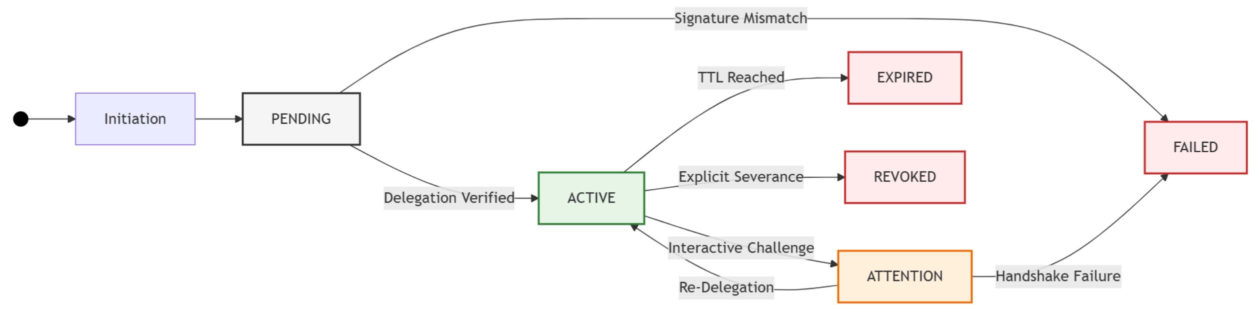

Agent
Authority
User
Provider
Phase 1: Initiation (Intent Binding)
1 ConnectionRequest (Provider ID, Scopes, Context)
HMAC-SHA256(Context Payload)
2 connection_id (UUID), auth_url
Phase 2: Delegation (User Consent)
3 Redirect to auth_url
4 Access auth_url
5 Redirect to OAuth/Capture Flow
6 Consent Prompt
7 Grant Consent
8 Auth Artifacts (Code/Tokens)
Phase 3: Activation (Vaulting)
9 Verify HMAC Signature & State Integrity
10 Vault Credentials (AES-GCM Encryption)
11 Redirect to return_url (Success)
12 Finalize Connection
Data Plane Entry & Maintenance
loop
[Continuous Lifecycle]
13 Request Strategy (connection_id)
alt
[Status: ACTIVE]
14 Dynamic Strategy (Lease)
15 Execute Request (Injected Auth)
[Status: ATTENTION]
16 Intervention Required (MFA/Challenge)
17 Signal for Re-Handshake (Human-in-the-Loop)
Agent
Authority
User
Provider

Figure 2: Nexus Handshake State Lifecycle

This diagram defines the State Transitions during the Nexus Handshake. It shows how the connection transitions from its initial intent to its final terminal state.



2.2 The Universal Provider Model

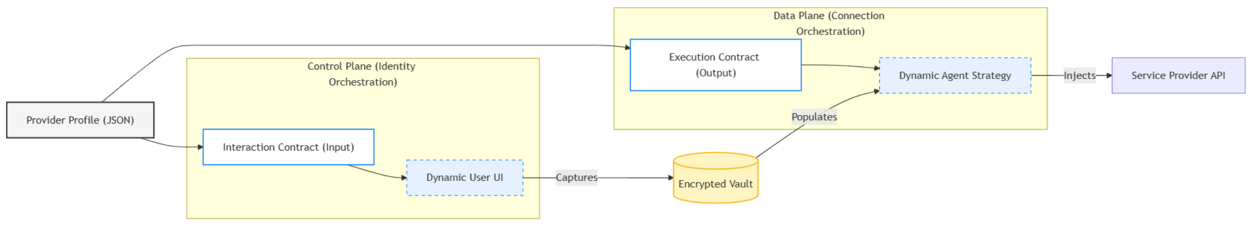
The Nexus Protocol achieves universality by abstracting the definition of an external service into a Provider Capability Model. This model enforces a strict decoupling between the acquisition of authority and the exercise of authority. By standardizing these two phases, the protocol eliminates the need for provider-specific code within the Agent runtime [4].

A valid Provider Definition within the Nexus ecosystem must explicitly declare two contracts:

1) The Interaction Contract (Input): This defines the interface required to establish authority. For standard providers (OAuth), this is defined by IETF standards. For custom or legacy providers, this is an explicit Capture Schema describing the data (keys, secrets) the Authority must solicit from the User during the Handshake.
2) The Execution Contract (Output):Defines the mechanism required to exercise authority. It serves as the blueprint for the Dynamic Strategy, mapping the stored credentials to transport-level instructions (e.g., Header Injection, Request Signing) for the Agent.

Figure 3: The Universal Provider Definition

The diagram depicts how Nexus uses a JSON Provider Profile to separate user-facing input capture (Interaction Contract, via dynamic UI to vault) from agent runtime actions (Execution Contract, generating strategies for API calls). This enables flexible, code-free integrations.


Provider Profile (JSON)
Control Plane (Identity Orchestration)
Interaction Contract (Input)
Dynamic User UI
Captures
Encrypted Vault
Populates
Data Plane (Connection Orchestration)
Execution Contract (Output)
Dynamic Agent Strategy
Injects
Service Provider API

Example: A Provider Capability Definition

The following JSON structure illustrates how these two contracts are represented in an Implementation. It links the User's input requirements directly to the Agent's runtime behavior.

```
{
  "provider_profile": {
    "name": "internal-data-lake",
    "interaction_contract": {
      "credential_schema": {
        "type": "object",
        "properties": {
          "api_key": { "type": "string", "title": "API Key" },
          "region":  { "type": "string", "title": "Region" }
        },
        "required": ["api_key"]
      }
    },
    "execution_contract": {
      "auth_strategy": {
        "type": "header",
        "config": {
          "header_name": "X-Data-Lake-Auth",
          "credential_field": "api_key"
        }
      }
    }
  }
}
```

By formalizing these contracts, the Protocol ensures that any service, regardless of its native authentication mechanics can be represented uniformly to both the User (during Handshake) and the Agent (during Usage).

3. Agent Connection Orchestration

The second pillar is Connection Orchestration. In traditional systems, authentication is often treated as a static setup step. However, for autonomous agents running indefinitely, authentication is a dynamic, continuous process [6]. Tokens expire, keys are rotated, and network policies change. Connection Orchestration defines the lifecycle that an Agent must implement to transform a static `Connection ID` into a resilient, self-healing communication channel.

3.1. Standard Execution Strategies

To achieve the universal adapter capability, the Nexus Protocol mandates that all Agents support a foundational set of execution strategies. These primitives allow the Agent to interact with the vast majority of existing web services without requiring provider-specific code.

1) Header Injection (“header”): The Agent injects a specific value into an HTTP header, supporting OAuth 2.0 Bearer tokens and standard API Keys.
2) Query Parameter Injection (“query_param”): The Agent appends key-value pairs to the request URL, a requirement for WebSocket handshakes and specific legacy GET interfaces.
3) Basic Authentication (“basic_auth”): The Agent encodes credentials using the standard Authorization: Basic <base64> scheme, ensuring compatibility with legacy enterprise infrastructure and message brokers.
4) Cryptographic Signing (“hmac” / “aws_sigv4”): The Agent uses a leased secret key to compute a cryptographic signature of the request. For high-security environments, such as those utilizing AWS SigV4.

3.2. The Active Lifecycle

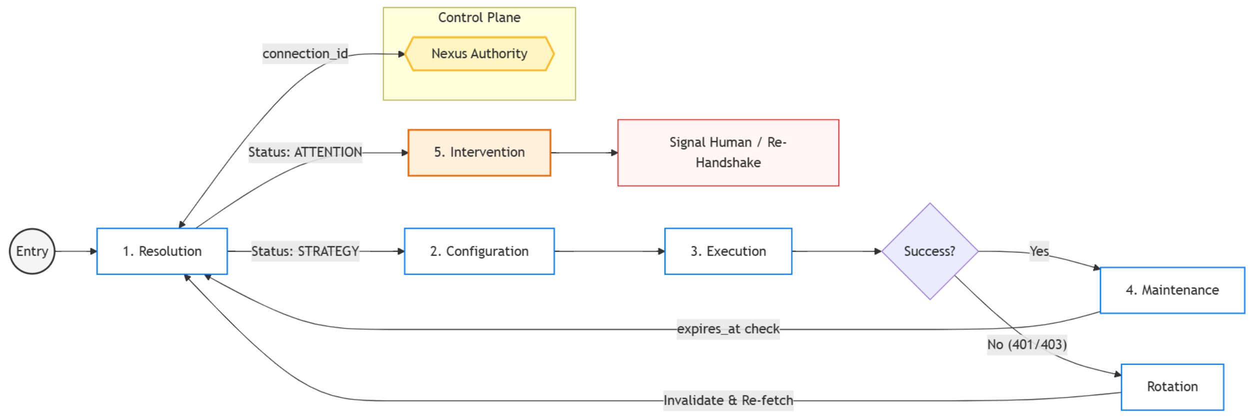
To solve the problem of fragile autonomy, the Agent must implement a resilient maintenance loop. This loop ensures that the Agent’s "Leased Identity" remains valid without manual intervention. Below is how this is achieved:

1) Resolution: The Agent fetches the latest Dynamic Strategy from the Authority using the connection_id.
2) Configuration: The agent applies the Strategy to the transport layer (e.g., injecting headers into HTTP requests or metadata into gRPC streams). The Agent must cache this strategy locally until “expires_at” is reached or a terminal error occurs.
3) Maintenance: The Agent proactively monitors the expires_at timestamp provided in the strategy, initiating a re-resolution before the lease expires.

4) Rotation: Upon encountering a “401 Unauthorized” error, the Agent handles this by invalidating local caches and re-fetching the Strategy from the Authority. This step automatically heals "Configuration Drift" if a Provider changes its authentication scheme (e.g., key rotation, algorithm update), the Agent automatically fetches and applies the new Strategy upon the next failure.
5) Intervention: If a refresh fails due to an interactive challenge (e.g., MFA, CAPTCHA, or password change), the Authority transitions the connection to “ATTENTION”. The Agent receives this status and must pause operations, signaling the need for human re-delegation (a new Handshake) rather than endlessly retrying.

Figure 4: Agent Connection Lifecycle

This flowchart shows the agent's runtime loop for an active connection: resolving strategy from the Authority (Control Plane), configuring and executing calls, checking success, and entering maintenance. On failure (e.g., 401/403 errors), it triggers rotation (invalidate & re-fetch) or intervention (signal human/re-handshake) for resilience.



4. Security Model: The ‘Least Privilege’ Agent

The Nexus Protocol enforces a security model designed for zero-trust environments, prioritizing containment and revocation over permanent trust. To achieve this, the protocol employs the Principle of Leased Identity. The core tenet of Nexus security is that Agents are Guests, not Owners; they hold "Leases" to an identity, not the "Deeds". This leased model aligns with emerging approaches for agentic workloads that emphasize short-lived, just-in-time credentials without stored secrets [8], but extends it with centralized vaulting and instant revocation. In this model, Master Secrets (the Deed), including

Refresh Tokens and Client Secrets are stored exclusively within the Authority's encrypted vault and never leave the Control Plane. The Agent receives only short-lived Usage Secrets (the Lease), such as access tokens with restricted TTLs, necessary to execute immediate tasks. This centralized approach contrasts with decentralized frameworks for agent-blockchain integrations, which rely on DIDs and key delegation for peer-to-peer trust but face challenges in revocation and auditing [11].

4.1 Blast Radius Containment

In a standard library-based implementation, the Agent would hold secrets in its process memory, creating a vulnerability. In such an implementation, if an Agent process is compromised (e.g., via RCE), an attacker can dump the memory and retrieve current secrets. However, because these are Leased Identities, the compromise is Time-Bounded. The attacker cannot refresh the lease without the Authority's permission. Once the lease expires or is explicitly revoked, the stolen credentials become useless, offering a significantly smaller blast radius than static configuration files. This containment is made possible by a Centralized Control Plane where security policy is enforced at the Authority rather than the edge. This centralization enables instant revocation; an administrator can sever a connection_id at the source, ensuring that the next time an Agent attempts a lifecycle heartbeat, access is denied immediately [7]. Furthermore, because the Authority mediates all identity acquisition, it provides granular auditing. Every strategy resolution and token refresh is a recorded event, offering a complete, immutable audit trail of agent activity that is impossible to achieve when secrets are decentralized across a fleet.

Even so, for environments requiring the highest security assurance, the Protocol supports a Sidecar Deployment Model. In this model, the Agent holds *zero* secrets. It sends unauthenticated requests to a local sidecar proxy. The sidecar intercepts the traffic, resolves the execution contract from the Authority, injects the credentials, and forwards the request. This eliminates the shared memory risk entirely; even a fully compromised Agent process yields no credentials to an attacker. While this implementation is the standard for zero secrets autonomy, its implementation necessitates a more complex infrastructure footprint; a small price to pay in mission critical enterprise environments.

5. Future Vision

The Nexus Protocol lays the foundation for a much larger vision: The Autonomous Capability Fabric [5]. By solving the Identity problem first, we unlock the ability for agents to self-discover and self-onboard tools. In this paradigm, the agent is responsible for the entire process of tool generation, connection, and communication with external providers. An Agent identifies a new tool definition or service interface via standards such as MCP or OpenAPI, generates the client code to use that tool, and uses the Nexus Protocol to securely acquire the necessary connection, all without human intervention. This transforms static tool registries into living ecosystems where agents can organically expand their capabilities, secured by the Nexus Protocol.

6. Conclusion

The transition to autonomous software requires a fundamental rethinking of our security infrastructure. We can no longer rely on authentication patterns designed for interactive humans or static servers. To support fleets of transient agents, we must adopt a dynamic, leased identity model that scales with the software itself.

The Nexus Protocol formalizes this model. By abstracting the complexity of diverse authentication standards into a unified Provider Capability Model and a polymorphic Execution Contract, Nexus Protocol treats authentication as data, not code. This creates a universal language for Agent-to-World communication, allowing developers to build agents that are inherently secure, easily integrated, and capable of long-running autonomy without becoming entangled in the specifics of provider logic.

Just as HTTP standardized how clients fetch resources, Nexus standardizes how agents acquire authority. It provides the missing infrastructure layer necessary to move AI from a conversational novelty to a trusted, autonomous economic actor.

Acknowledgements

I thank Muyukani Kizito for sparking the development of this work by identifying the scalability limitations of existing tool-use protocols, specifically the need for bespoke integrations for every service. His vision for agents capable of creating their own tools led directly to my realization that standardized Identity and Connection Orchestration must serve as the prerequisite infrastructure for such autonomy.

I also thank Victor Ashioya for his peer review; his insights into interactive security challenges led to the formalization of the ATTENTION state and the Intervention stage.

References

1. Okta. What is AI agent identity? Securing autonomous systems [Internet]. Okta; 2025 Oct [cited 2026 Jan]. Available from: https://www.okta.com/identity-101/what-is-ai-agent-identity.
2. Strata Identity. What is AI agent identity management? 2026 guide [Internet]. Strata Identity; 2025 Dec [cited 2026 Jan]. Available from: https://www.strata.io/glossary/ai-agent-identity-management.
3. Microsoft Entra. Agent OAuth flows - on-behalf-of flow [Internet]. Microsoft; 2025 Nov [cited 2026 Jan]. Available from: https://learn.microsoft.com/en-us/entra/agent-id/identity-platform/agent-on-behalf-of-oauth-flow.
4. IETF Draft. OAuth 2.0 extension: on-behalf-of user authorization for AI agents [Internet]. IETF; 2025 May [cited 2026 Jan]. Available from: https://www.ietf.org/archive/id/draft-oauth-ai-agents-on-behalf-of-user-00.html.
5. South T et al. Identity management for agentic AI: the new frontier of authorization [Preprint]. arXiv; 2025 Oct [cited 2026 Jan]:9 p. Available from: https://arxiv.org/abs/2510.25819.
6. South T et al. Authenticated delegation and authorized AI agents [Preprint]. arXiv; 2025 Jan [cited 2026 Jan]:12 p. Available from: https://arxiv.org/abs/2501.09674.

7. HashiCorp. Zero trust for agentic systems: managing non-human identities at scale [Internet]. HashiCorp; 2025 Dec [cited 2026 Jan]. Available from: https://www.hashicorp.com/blog/zero-trust-for-agentic-systems-managing-non-human-identities-at-scale.
8. Aembit. Agentic AI and workload identity & access management [Internet]. Aembit; 2025 [cited 2026 Jan]. Available from: https://aembit.io/.
9. SPIFFE Project. SPIFFE concepts [Internet]. SPIFFE Project; [date unknown] [cited 2026 Jan]. Available from: https://spiffe.io/docs/latest/spiffe-about/spiffe-concepts.
10. SPIRE Project. SPIRE concepts [Internet]. SPIRE Project; [date unknown] [cited 2026 Jan]. Available from: https://spiffe.io/docs/latest/spire-about/spire-concepts.
11. Alqithami S. Autonomous agents on blockchains: standards, execution models, and trust boundaries [Preprint]. arXiv; 2026 Jan [cited 2026 Jan]:30 p. Available from: https://arxiv.org/abs/2601.04583.

Licence & Usage