AI·ONE

SECURITY PAPER

ContextOne Security

Shaun Laurens
Chief Technology Officer, AI One

PUBLISHED
November 2026

VERSION
1.0

IMPLEMENTATION REVIEW
30 October 2026

Contents

01 **Introduction** 03

02 **Foundational Security** 06
Encryption, communication between services, and the mechanisms used to issue and enforce access grants.

03 **Sandbox Security** 15
Isolation, governance, and auditing for agent use of Python, Bash, and GNU command-line utilities.

04 **Ontology Security** 27
Namespaces, guardrails, roles, grants, taints, and attribute-based access controls that govern access to ontology resources.

05 **Tool Interface Security** 36
Controls governing agent access to tools exposed through MCP and other interfaces.

06 **Structured Data Security** 44
Authorization, governance, and auditing for agent access to enterprise structured data sources.

07 **LLM Interface Security** 51
Runtime guardrails, rate limits, and cost controls for interactions with language models.

08 **Data Room Security** 57
Fine-grained access to unstructured information held in systems such as Microsoft SharePoint and Atlassian Confluence.

01

Introduction

ContextOne enables AI agents to work with enterprise information and services within defined security boundaries. Its security architecture draws on AI One's experience in regulated industries, particularly financial services, where access controls, isolation, and auditability are essential requirements.

Agents need useful capabilities to perform useful work. They may need to run command-line utilities, execute Python code, process documents, and query structured data. ContextOne's security approach is to make these capabilities available through governed interfaces, with controls over which resources an agent can access and which actions it can perform.

The architecture addresses five security concerns:

- **Agent authority:** A model's output is not a trusted authorization decision. Agents must operate within explicitly granted permissions, including when a model produces unsafe instructions or encounters malicious content.
- **External threats:** Interfaces exposed to users, agents, and connected systems need protection against unauthorized access and misuse.
- **Shared resources:** Enterprise use requires controlled sharing across agents, teams, organizational units, and systems. Access boundaries must remain explicit as resources are shared.
- **Containment:** If a component is compromised, isolation and restricted network access should limit the resources an attacker can reach and the actions they can take.
- **Auditability:** Security-relevant activity needs to be recorded so organizations can investigate events and review how access was exercised. Audit records should support retention requirements and remain usable in open formats.

These controls must also support the ways agents work effectively. Familiar command-line utilities, shell scripts, and Python allow agents to carry out a wide range of tasks. Agents also need access to enterprise file formats including text, Microsoft Office documents, PDFs, and Parquet. Access to structured data allows them to work with information in its existing form. The security objective is to govern these capabilities while preserving their usefulness.

This document explains ContextOne's security architecture and the controls available to organizations deploying it. It also describes how those controls support an organization's own access policies, operational procedures, and audit requirements.

The document covers the following areas:

02 **Foundational security**	Encryption, communication between services, and the mechanisms used to issue and enforce access grants.
03 **Sandbox security**	Isolation, governance, and auditing for agent use of Python, Bash, and GNU command-line utilities.
04 **Ontology security**	Namespaces, guardrails, roles, grants, taints, and attribute-based access controls that govern access to ontology resources.
05 **Tool interface security**	Controls governing agent access to tools exposed through MCP and other interfaces.
06 **Structured data security**	Authorization, governance, and auditing for agent access to enterprise structured data sources.
07 **LLM interface security**	Runtime guardrails, rate limits, and cost controls for interactions with language models.
08 **Data Room security**	Fine-grained access to unstructured information held in systems such as Microsoft SharePoint and Atlassian Confluence.

02

Foundational Security

ContextOne combines enterprise identity, authenticated service communication, and explicit access grants to govern how users and agents interact with enterprise resources. The Session Router connects these controls: it establishes the caller's identity, resolves their permissions, and issues scoped credentials for downstream operations. Services independently validate those credentials and apply the access rules for the resources they manage.

Encryption and service identity

In the secure runtime, the Session Router uses TLS to protect internal communication and client certificates to authenticate workloads requesting delegated credentials. Certificates carry a SPIFFE service identity, which the router checks against a workload registry. That registry defines which services a workload may contact, which scopes it may request, and which resource bindings are required. Possession of a trusted certificate therefore establishes a workload's identity; its configured policy determines what it can request.

Delegation tokens are digitally signed using RS256. The router retains the private signing key, while receiving services obtain public verification keys through a JSON Web Key Set, or JWKS. Tokens identify the signing key used, allowing services to verify their authenticity. The keyring supports rotation by retaining retired public keys so that previously issued tokens can remain verifiable during their remaining lifetime. Signing protects the integrity and authenticity of a token; TLS protects its confidentiality in transit.

Enterprise identity

ContextOne integrates with enterprise identity providers such as Microsoft Entra ID. In the Entra browser flow, the Session Router uses OpenID Connect authorization code exchange with PKCE, state, and a nonce. It validates the returned identity token's signature, issuer, audience, validity period, and nonce before accepting the identity. The router extracts the directory tenant and user identity alongside the asserted application roles and groups.

Identity mapping connects these external claims to a ContextOne tenant, principal, and permissions. Configured rules can translate application roles and groups into ContextOne grants. Entra access-tag mappings can add permission-specific resource scopes after the underlying identity has been successfully resolved. This allows organizations to express access policy using their directory structure while retaining explicit control over what those identities can do within ContextOne.

Entra monitoring includes handling for disabled or deleted accounts and invalidated sign-in sessions.

Short-lived delegated credentials

The Session Router issues short-lived credentials for downstream access. A workload requesting a token must authenticate to the router, and its requested audience, scope, subject, and resource binding must satisfy the workload policy. The resulting token identifies its intended recipient and delegated authority. This allows a service to receive the authority needed for an operation without receiving the user's original identity-provider credential.

The generalized Security Token Service currently issues tokens with a 60-second lifetime. The separate router-to-NodeOne delegation path defaults to 300 seconds, with configuration limited to a maximum of 300 seconds. These lifetimes limit how long an issued credential remains usable. Obtaining another token requires another issuance decision.

Browser sessions have a separate lifecycle. The router uses a server-side session referenced by a cookie marked `Secure`, `HttpOnly`, and `SameSite=Lax`. Session renewal is governed by an idle window and an absolute lifetime limit. These controls distinguish the duration of a user's sign-in from the much shorter lifetime of credentials issued for service access.

Grants and resource scope

A ContextOne grant associates a permission with its applicable resource scope. Permissions describe allowed actions, such as reading or writing ontology information. Access tags restrict the resources to which that permission applies.

Scopes are maintained per permission. For example, a principal may be allowed to read information tagged `finance` and `hr`, while being allowed to write only information tagged `hr`. The broader read scope does not expand the write scope. Delegation can further narrow these grants through agent inheritance limits. Where an administrator grants access to all registered tags, the router expands that scope into explicit tags when preparing the token.

Enforcement in Agent Ontology Server

Agent Ontology Server illustrates how these controls are applied at the service boundary. For router-issued credentials, AOS verifies the token's signature, trusted issuer, AOS audience, and expiry. It then checks both the principal's permission and the token's delegated scope against the requested operation. A valid token alone does not authorize every operation.

AOS evaluates resource access using the tags associated with the permission that authorized the request. It does not combine all of the caller's tags across unrelated permissions. For ordinary tag-based visibility, the caller must hold every access tag attached to the record.

Namespaces can enforce access tags across their contents. Child namespaces inherit their ancestors' requirements, and the server stamps enforced tags onto entities and relationships at the mutation boundary. Writers must hold the namespace's required tags, and an update cannot remove those requirements by omitting them from its payload. This makes the restriction a server-enforced property of the namespace.

Policy-based authorization

ContextOne uses Cedar as a policy language for runtime authorization within the NodeOne harness. Policies evaluate the principal, requested action, target resource, and relevant execution context. This allows organizations to express restrictions independently of the agent's instructions and have the harness apply them before an operation proceeds.

For tool invocation, policies can consider the session's accumulated security labels, the selected tool or tool group, the dispatch backend, and tool arguments. A policy can prohibit an operation or require operator approval. For example, an organization can prevent a session that has accessed personal information from invoking an external messaging tool, or require approval before sending information to a particular destination.

NodeOne also provides Cedar authorization surfaces for virtual filesystem operations, mounted Data Rooms, ontology operations, and key-value access. These controls complement the permissions and resource-level ABAC checks enforced by downstream services.

Access rights and information exposure

ContextOne distinguishes between the information a principal is authorized to access and the information an agent session has encountered.

Access grants define authority. They associate permissions with resource scopes, including access tags. Session labels—called taints in the implementation—record classifications acquired during execution. Acquiring a label does not grant additional access. It provides information that policy can use to constrain subsequent actions.

This distinction addresses a limitation of access control alone: permission to read sensitive information does not necessarily include permission to disclose it through another tool or save it in a less restricted location.

Structured data access

Structured Data Gateway applies ABAC requirements across catalogs, schemas, relations, and fields. Requirements are additive: a field inherits applicable tags from its parent objects, and ontology overlays can contribute further requirements.

The SQL analyzer identifies the relations and columns accessed by a query, including references reached through common table expressions and subqueries. It checks the effective requirements against the caller's access attributes. Unauthorized references are rejected; the check is based on the analyzed query rather than a description supplied by the agent. Relation-level checks also cover accesses that do not reference individual columns.

SDG responses carry effective access tags through the tool-result contract. The ATI adapter preserves those tags, and NodeOne merges the returned labels into the session before continuing with the result. This connects the controls applied at the data source to the controls applied during subsequent agent execution.

The one-way session ratchet

NodeOne maintains an accumulating set of security labels for each session:

```
labels after a read = existing session labels ∪ labels acquired from the
read
```

The set grows as the session encounters classified information. Reading unclassified information later does not remove an earlier classification. NodeOne records provenance for acquired labels, including the tool or ontology record that introduced them, and includes the label set in session snapshots so that restoring a session restores its classifications.

Labels can originate from returned access metadata, ontology reads, an agent's declared access classification, or configured Cedar taint-application policies. The latter allow an organization to classify a session after particular tool operations, including tools whose results do not supply their own classification metadata.

The ratchet is deliberately conservative. Once sensitive information has entered a session, later output may be influenced by it even when that output does not reproduce the source verbatim. The harness therefore retains the classification rather than asking the model to decide whether it has forgotten the information.

Controls on a labelled session

Before dispatching a tool, NodeOne evaluates applicable Cedar invocation policies against the session's current labels and the proposed call. Configured controls can:

- Block particular tools or groups of tools after sensitive information has been accessed.
- Restrict destinations or other arguments, such as an external messaging channel.
- Require operator approval for an otherwise restricted operation.
- Constrain the actual dispatch backend, so a tool name alone is insufficient to satisfy a backend-specific rule.

NodeOne resolves the effective tool behind wrappers such as `call_tool`, allowing policy to evaluate the underlying operation. A hard denial takes precedence over an approval requirement.

An approval authorizes an exception to a particular policy condition; it does not clear the session's labels. Approval grants are bound to the policy identifier and hash, tool, and covered labels. A changed policy or an uncovered label can therefore require a new decision.

Preserving restrictions in the ontology

The same classifications follow agent-generated information into AOS. When NodeOne requests an AOS delegation token, it supplies the session labels computed for that dispatch. The Session Router accepts this attestation only from the owning NodeOne workload through the supported session-bound flow, validates the labels against the permitted scope, and signs them into the AOS token. The claim identifies the session and label revision.

At the mutation boundary, AOS derives an agent-written record's labels from:

```
session labels ∪ existing record labels ∪ namespace-enforced labels ∪
explicitly supplied tags
```

An agent cannot remove an existing restriction by omitting a tag from an update. Its write must also be permitted under the access tags associated with the permission authorizing that operation. If the session's classification exceeds that write authority, AOS refuses the write.

An absent session-label claim is treated as unknown, rather than as an empty classification. By default, AOS rejects agent-driven writes whose context labels are unknown.

REQUIRED CONFIGURATION

Production deployments must enable the secure service paths, configure trusted issuers and workload policies, and restrict delegated audiences, scopes, and resource bindings. Development bypasses must be disabled. Signing-key rotation, replay protection where required, and policy-dependency failures must be tested under the deployed configuration.

REVOCATION AND PROPAGATION

Token expiry, identity-provider change detection, service-policy refresh, and source-permission synchronization have different timelines. A short-lived token limits credential reuse; it does not refresh an old SharePoint ACL or invalidate data already delivered to a caller. Operators must document each propagation path and its failure response. Retained credentials, claims, and authentication diagnostics must be covered by the retention inventory.

03

Sandbox Security

Command-line utilities and Python give agents practical ways to search, transform, and analyze enterprise information. They allow an agent to process data outside its context window and return selected results, reducing the amount of source material that must pass through the model. For suitable tasks, this can improve accuracy and reduce token consumption.

ContextOne provides these capabilities through a governed execution environment. Agent-generated code is treated as untrusted. The agent can express a computation or request an operation, but the trusted host determines which resources it may access, which effects may occur, and which results may leave the sandbox.

Threat model

The sandbox assumes that generated code may attempt actions beyond the agent's authorization. This can arise from model error, malicious instructions in retrieved content, or deliberate attempts to misuse available capabilities.

The controls address several related threats:

- Access to files, credentials, services, or host resources outside the authorized scope.
- Disclosure of restricted information through network requests, files, tool calls, or returned output.
- Attempts to manufacture identity, approvals, or resource authority inside a script.
- Resource exhaustion through excessive computation, allocation, output, or repeated operations.
- Duplicate or unauthorized effects following cancellation, retries, process failure, or policy changes.
- Interpreter vulnerabilities that could allow code to escape the intended language interface.

The trusted boundary includes the host integration, policy plane, resource adapters, storage implementation, worker broker, and underlying operating system. Interpreter isolation reduces the authority available to a compromised worker; it does not eliminate reliance on those components.

Virtual filesystem and session storage

NodeOne provides each session with a virtual filesystem for working files, tool results, and generated artifacts. Agents access these files through governed filesystem operations and sandbox utilities. File writes are checked against path permissions, and reserved internal paths are protected from ordinary agent writes. This gives agents a familiar file-based workspace while keeping storage access under runtime control.

Content-addressed storage

The filesystem separates a file's path and metadata from its stored contents. Ordinary file contents are identified by a SHA-256 digest, allowing files with identical contents to reference the same stored blob. Repeated copies therefore need not duplicate the underlying bytes.

Shared blobs are not modified in place. Writing different contents creates a different content reference, preserving other files that reference the original blob. New blobs are staged and synchronized before installation, preventing partially written content from becoming visible through shared references. Append-heavy files use a separate mechanism to avoid rewriting their entire contents on every append.

History and recovery

NodeOne records file-operation lineage and persists session checkpoints. Lineage records can associate a file operation with its tool call, model, turn, timestamp, and content hash, providing evidence of how an artifact was produced or changed.

Incremental checkpoints preserve file paths and content references in a manifest linked to the preceding checkpoint. Content already present in the session's archive is reused; new content is stored as compressed blobs. This reduces duplication across successive checkpoints while supporting reconstruction of retained session states. Archive objects are organized by session, so deduplication does not require a global content store shared across sessions.

Recovery follows the checkpoints and history retained by the persistence policy. A lineage event records an operation; it does not by itself guarantee that every intermediate file version remains recoverable indefinitely.

Per-session storage limits

The session manifest defines a storage quota. The current default is 100 MiB, with a maximum configurable allowance of 100 GiB. A zero configuration value resolves to that finite maximum.

Quota accounting measures the logical size of live session files, rather than the physical size of deduplicated blobs. Two identical files can share storage while each contributes its full size to the session's allowance. Deduplication therefore improves storage efficiency without allowing an agent to bypass its logical quota.

Writes that increase usage reserve the additional capacity before proceeding. Deleting a file or replacing it with a smaller file releases logical capacity. When a session resumes, usage is reconstructed from its live file records and active upload reservations.

The quota constrains ordinary agent writes. Trusted internal system writes are counted but may exceed it. Historical archives and physical backing-store consumption also require separate capacity and retention controls; the session quota is not a ceiling on every byte retained for that session.

Storage visibility through familiar utilities

NodeOne's built-in utilities read the virtual filesystem's metadata and quota accounting. `du` reports the logical sizes of files and directory contents, helping the agent identify where its working data is concentrated. `df` reports the session's configured capacity, current usage, available allowance, and percentage used.

These commands expose the storage information relevant to the agent's workspace. An agent can inspect usage, remove temporary files, and choose a smaller output before attempting another write. Removing a live file releases logical quota immediately; reclaiming its underlying bytes depends on remaining references and storage-retention behavior.

A shared intermediate language

Bash and Python use a common, typed intermediate language (IL) for supported external effects. A file operation or HTTP request becomes a structured proposal to the execution plane. The plane validates the proposal and evaluates the operation under trusted identity, policy, resource bindings, and execution limits.

The language frontends retain their own computation and control flow. They are not unrestricted host shells or general-purpose Python environments with ambient operating-system access. The IL contains no arbitrary host-command execution, policy-editing, credential-access, or declassification instruction. Resource references are created within the plane and cannot be supplied by a script as assertions of authority.

This places governance at the operation boundary. A policy does not need to infer every possible effect from the source text of a Bash command or Python program. It evaluates the concrete resource and operation when the frontend requests it. Structural validation establishes that a proposal is well formed; it does not authorize execution.

Native command implementations

ContextOne implements its supported shell builtins and command-line utilities in Rust. Commands execute within the sandbox runtime, with filesystem and other external effects routed through the governed execution plane. They do not launch the corresponding operating-system executables. The hardened container the sandbox operates in has no shell, and no installation of any executables offered to the agent via Bash.

Compatibility is checked against Bash and GNU utilities in a Debian reference environment pinned by container digest and package snapshot. The conformance suite compares command output, diagnostics, exit status, and behavior across successive shell invocations. This helps preserve the familiar behavior agents rely on when composing commands and processing their results.

The tests check exact output for the covered cases, including formatting and whitespace, with defined normalization for selected diagnostics. Unsupported behavior and intentional differences are recorded explicitly. Conformance applies to the tested command surface; it is not a claim of complete GNU compatibility.

Memory-safe implementation

Rust provides memory-safety controls through ownership, borrowing, and lifetime checking, supported by runtime checks where necessary. These mechanisms prevent classes of defects such as use-after-free, double-free, and unchecked out-of-bounds memory access in safe Rust. Its concurrency rules also prevent data races, although application-level race conditions still require explicit design and testing. These guarantees depend on the soundness of underlying libraries and their safe interfaces.

ContextOne makes this an enforced development rule. The sandbox workspace sets `unsafe_code = "forbid"`, causing compilation to reject unsafe code in the governed crates. A CI check verifies that every first-party runtime crate inherits or explicitly declares the prohibition, with additional checks covering the maintained vendored components. The prohibition cannot be relaxed by a local `allow(unsafe_code)` annotation.

Memory safety complements authorization and isolation. It does not establish that a policy is correct, prevent resource exhaustion, or eliminate vulnerabilities in dependencies, native libraries, or the operating system. The prohibition applies to the checked source components; it does not mean the complete dependency graph contains no unsafe or native code.

The harness retains authority

The trusted host binds the tenant, principal, session, storage, output destination, and budgets. Model-supplied arguments cannot replace that context or provide approval credentials.

For each effect, the plane constructs an authorization request containing the action, destination, current security labels, invocation identity, and payload digest. The host policy returns allow, require approval, or deny. Resource resolution and subsequent use have separate checks, and an approval wait does not freeze authority: policy is checked again when execution resumes.

NodeOne supplies the product-specific authority: routed permissions, filesystem and Data Room access checks, tool resolution, and approval routing. The reusable runtime supplies execution, effect mediation, receipts, and worker lifecycle management. Together, these components arbitrate what happens; the interpreter does not make authorization decisions.

Cedar fits into this architecture through the host's authorization controls. The standalone runtime exposes a policy interface; it does not itself ship a complete application-specific Cedar policy for every IL operation.

Governing files, network access, environment, and time

CAPABILITY	GOVERNANCE
Files	Paths refer to the virtual filesystem and configured resources. Reads, listings, metadata observations, creation, replacement, append, and removal have explicit authorization boundaries. Version checks detect conflicting changes. Backend checks continue to enforce mounted-resource permissions and supported mutation semantics.
Network	Supported Bash `curl` and Python `sandbox.http_request` calls request typed HTTP effects. The trusted host performs the network operation after origin, policy, address, label, and budget checks. Workers receive neither a general network client nor host credentials.
Environment variables	Worker processes start with a cleared operating-system environment. Shell-local variables are auditable interpreter state.
Time and generated values	The IL includes governed generated-source operations, including recorded clock and UUID sampling.

File operations preserve classification as well as permissions. Metadata and directory listings can reveal sensitive information, so they are governed observations rather than unrestricted alternatives to content reads. Changing virtual mode bits does not create new policy authority.

For HTTP, the runtime requires explicit host configuration. It checks each redirect, validates and pins DNS results, disables environment proxies and automatic transport retries, and prevents HTTPS-to-HTTP redirects. Outbound requests carry the session's accumulated labels into policy evaluation; responses add their configured classifications before release. An origin allowlist must therefore be accompanied by appropriate information-flow policy.

Information flow and output

The execution plane maintains a security-label set that grows as the sandbox encounters classified information. Subsequent effects and output decisions use that accumulated set. Transforming data, producing a summary, or constructing a new byte string does not remove its inherited restrictions.

Output is a governed boundary. The runtime authorizes delivery to the host-bound destination, including delivery of a previously recorded result. The worker's protocol output is not connected directly to the model or user as an unchecked stream.

This supports a consistent rule: permission to compute on information does not automatically authorize its publication. A permitted write may also remain committed when final output is withheld; the runtime must report that distinction without releasing the protected result.

Isolation in a sidecar

The Linux sidecar separates interpreter processes from the trusted service. The host retains policy, approvals, storage, and credential provisioning. The sidecar contains the worker broker and language workers, communicating with the host through an authenticated Unix-socket connection.

Before processing untrusted source, workers install a restrictive syscall filter. The allowlist permits computation and bounded protocol communication while denying operations such as opening files, creating sockets, launching processes, and inspecting other processes. Workers also receive address-space and CPU limits, a restricted file-descriptor limit, and disabled core dumps.

The deployment contract adds container controls: separate non-root identities, a read-only root filesystem, dropped capabilities, disabled privilege escalation, explicit resource limits, and narrowly scoped mounts. Database files, cloud credentials, approval credentials, Kubernetes API tokens, and container-runtime sockets must remain outside the worker container.

A sidecar in the same pod has important limits. Containers share the pod's network namespace, including localhost, and share the host kernel. Kubernetes NetworkPolicy alone does not separate the worker from services listening within the pod. The worker syscall restrictions are therefore part of the security boundary.

This is process and container isolation, not a separate virtual machine or a guarantee against kernel exploits and timing side channels. Container configuration and the trusted broker remain material to its effectiveness.

Failure, recovery, and resource limits

The runtime binds execution to an immutable invocation identity and records durable effects for recovery. Replaying a recorded result requires current authorization. An uncertain remote effect is not automatically repeated, because a lost response does not establish that the operation failed.

Worker termination is treated similarly. A disconnected socket or restarted broker is not sufficient evidence that an earlier worker has stopped. Uncertain execution retains its reservation until ownership and termination can be established.

Instruction, byte, output, process, memory, CPU, and lifetime limits bound different parts of execution. These limits complement one another; they do not constitute a complete deployment-wide storage quota or eliminate resource contention.

REQUIRED CONFIGURATION

Deploy the worker isolation, syscall restrictions, narrowly scoped mounts, and resource limits described above. Configure explicit network destinations and information-flow policies; an origin allowlist alone does not authorize disclosure. Test cancellation, dependency failures, and uncertain execution without automatically repeating effects whose outcome is unknown.

RETAINED SESSION DATA

Saved tool results, transcripts, virtual files, lineage, checkpoints, and exports may contain sensitive content. Production operations must identify the users and support roles permitted to inspect, restore, or export sessions, and restrict direct archive access separately. Exact support and export entitlements require deployment verification.

Deleting a live file or freeing logical quota does not delete retained checkpoints, transcripts, exports, or backups. Set retention and deletion procedures for each store, including removal of content that is no longer referenced. Do not infer an archive deletion deadline from the session quota.

04

Ontology Security

ContextOne's ontology holds information that agents use to interpret enterprise resources and perform work, including concepts, relationships, catalog metadata, skills, and tool definitions. Protecting this information requires controls over who can discover it, who can use it, and how changes become available to other users and agents.

Agent Ontology Server combines operation-level permissions, attribute-based access control, namespace boundaries, and publication governance. Structured Data Gateway applies complementary controls when ontology information is used to discover or query structured data.

Permissions and resource visibility

AOS checks the permission required by each operation and, for delegated requests, the scope carried by the delegation token. It then evaluates resource access using the tags associated with the permission that authorized the request.

Access tags restrict visibility of entities and relationships. A caller must hold all required tags for the resource. A broad read grant does not expand the scope of a separate write grant, and possession of a valid token does not authorize every ontology operation.

Namespaces and inherited restrictions

Namespaces provide an explicit structure for organizing and governing ontology content. User-facing writes must target a registered namespace. Namespace discovery and error responses respect access restrictions, so a caller is not given a list of protected namespaces merely because a write failed.

A namespace can require access tags across its subtree. Child namespaces inherit their ancestors' requirements and may add further restrictions. The server applies the effective tags to entities and relationships at the mutation boundary; a writer cannot evade those requirements by leaving tags out of a request.

Namespace access requirements and publication policies serve different purposes. Access tags govern visibility and permitted resource use. Publication policies govern who may introduce or change shared content. Both can apply to the same namespace.

AOS and Structured Data Gateway

SDG owns the structured-data catalog and its source access requirements. Those requirements can be assigned at catalog, schema, relation, and field level, with additive inheritance. SDG checks the effective requirements during discovery and SQL analysis.

When SDG publishes catalog metadata into AOS, it carries the relevant restrictions onto the ontology records. AOS tracks SDG-owned restrictions separately from restrictions added within the ontology. Catalog reconciliation preserves independently owned AOS restrictions, including classifications inherited from agent sessions. A refresh of source metadata therefore does not simply replace every access restriction on the record.

The authorization boundaries remain distinct:

BOUNDARY	RESPONSIBILITY
AOS	Controls access to ontology records, including catalog metadata, and governs changes to shared knowledge and policy.
SDG	Controls structured-data discovery and query execution against the resources referenced by a request.
NodeOne	Retains classifications acquired from reads and applies runtime policy to subsequent agent actions.

Seeing a table's description in the ontology does not itself authorize querying that table. Likewise, permission to query a dataset does not confer authority to change its ontology definition or access policy.

Ontology-managed row policies

AOS can also act as the source of published row-policy manifests for SDG. This is separate from metadata visibility and tag-based ABAC.

When configured to use this integration, SDG retrieves the applicable policy manifest for a tenant and catalog, resolves the required facts, and applies the resulting row restrictions during query preparation. Policies can deny access, permit unrestricted access, or require filtering. For catalogs configured to require policy coverage, an uncovered relation causes rejection rather than unrestricted execution.

The integration uses bounded caching. It supports a maximum age for serving a last-known-good manifest when AOS is unavailable. Deployment configuration determines whether row policies are enabled and whether their source is AOS or SDG's local store.

Enterprise publication governance

Enterprise mode adds a controlled path for changing shared, live ontology content. Governance duties are scoped to namespaces and distinguish owners, curators, proposers, approvers, and publishers.

Policies can require a minimum number of approvals, independent reviewers, representation from specified review groups, and a publisher separate from both contributors and reviewers. They can also determine whether agent review is acceptable. Independence is evaluated using authenticated identities, including delegated human identity; presenting different credentials for the same person does not establish independent review.

Each applicable ancestor policy is evaluated. A child namespace cannot weaken an inherited approval quorum, independence requirement, or required reviewer-group representation.

Approval applies to a particular candidate and its governance context. It is not a reusable authorization to publish different content. The server checks the approved change at the final mutation boundary, after ordinary access-tag and retention processing. A change that no longer matches the reviewed content is rejected.

Governance records themselves are protected from ordinary graph mutation. Internal maintenance also has constrained capabilities: a system writer does not receive a general exemption to publish semantic changes.

Content guardrails

Information retrieved from the ontology can enter an agent's context directly through tool responses. Descriptions, properties, skills, and other retrieved material may therefore influence both the agent's answer and its subsequent actions. Access controls determine whether the caller may retrieve that information; content guardrails address what the information contains.

AOS applies native content guardrails at authoring boundaries to reduce the persistence and subsequent retrieval of prohibited material. The configurable checks cover supported personal and financial identifiers, credential patterns, and profanity. They use deterministic recognizers, validation rules, structured-field checks, and locale-specific dictionaries. Protection is enabled by default, with deployment settings controlling the enabled categories.

Protecting stored content

For ordinary graph content, detected values are replaced with typed markers before mutation checksums, persistence, graph writes, and subsequent indexing. Entity names and descriptions are treated as prose; data-bearing properties receive structured inspection. Sanitization is permanent: AOS does not retain a reversal map that can restore the original value.

The processing distinguishes content from control information. Identity, authorization, routing, and other typed protocol fields are preserved rather than indiscriminately rewritten. Changes to indexed text mark the corresponding embeddings for refresh, and draft-similarity inputs are sanitized before leaving AOS.

Executable and immutable artifacts require different treatment. Silently editing a script or skill package could change its behavior or invalidate the relationship between reviewed content and stored bytes. AOS therefore rejects protected artifact content when sanitization would alter it. Hosted packages are checked before artifact objects are written, and dedicated approval operations recheck the package under the current policy. Transfer imports also preflight supported artifact content.

Prompt-injection checks

Hosted skills receive additional checks because their content is intended to guide agent behavior. The skill guard examines names, descriptions, and UTF-8 text files for recognized credential patterns, invisible Unicode characters, and selected prompt-injection phrases.

Detected credentials and invisible-text findings block the package. Recognized injection phrases produce warnings for review. These checks identify specific suspicious patterns; they do not establish that a skill is free of malicious instructions.

Enterprise publication governance provides a further control over shared content. Organizations can require independent review and evaluation before a proposed skill or ontology change becomes live. Content screening and review address different risks: automated checks identify supported patterns, while governance establishes who is accountable for approving the material.

Protection at the agent boundary

NodeOne also applies native content guardrails to supported tool-result and conversation-content boundaries. Structured results can be inspected before serialization removes useful field context. For example, a sensitive value in an SDG result can be recognized using its column or field name. A processing failure blocks forwarding rather than falling back to the original content.

Guardrails complement the controls that govern agent actions. Retrieved text does not grant permissions, supply an approval, or alter the trusted policy. If malicious content persuades a model to request an unauthorized operation, the harness and downstream services must still enforce their grants, Cedar policies, ABAC requirements, and sandbox effect controls.

Coverage and limits

Content recognition is necessarily bounded. The native checks do not identify every secret, personal identifier, prohibited statement, or prompt-injection technique. Binary media and arbitrary nested archives are not interpreted as text, and enabling protection does not automatically rescan historical AOS content.

The current prompt-injection phrase checks apply to the hosted-skill path; they should not be described as comprehensive screening of every ontology tool response. The security design combines content checks, controlled publication, and independent action authorization so that no single detector carries the full responsibility for protecting the agent.

Validation before publication

The enterprise implementation extends review beyond approval counts. Namespace-scoped type definitions and relationship constraints validate the resulting graph. Declared dependency contracts identify changes that would invalidate dependent content and can block publication or require evaluation.

Evaluation requirements can specify the evaluator, version, environment, minimum score, freshness, and independence. AOS retains and validates the evidence; evaluator code runs in external workers.

Field-level trust policies address conflicting information from different sources. They define scoped source authority and freshness requirements. Equal-priority disagreement can remain unresolved and block publication, rather than being silently resolved by whichever update arrived last.

Private work and shared publication

AOS provides an identity-bound personal namespace through the `self` alias. Its physical namespace is allocated by the server rather than selected by the caller.

In Enterprise mode, authenticated owners can edit their private content without applying the shared-publication workflow to every personal change. This exception checks both the existing and proposed record location. It cannot be used to move shared content into private storage or publish private content into a shared namespace without the applicable controls.

A separate compliance resolver can map an exact user identity to its namespace. It requires a reason and records the request before returning; resolving the namespace does not grant access to its contents.

Preserving classification and retention

Agent-generated ontology content retains the classifications of the session that produced it. AOS combines session labels, existing record labels, namespace requirements, and explicit tags at the write boundary. Omitting a tag from an update does not declassify the record.

Retention controls govern expiry and deletion separately. Active legal holds can block deletion or relocation, and retention execution rechecks the applicable controls under the mutation gate.

REQUIRED CONFIGURATION

Configure namespace access requirements and publication governance explicitly. Where SDG consumes ontology row policies, enable the intended policy source, require coverage for protected catalogs, and configure the maximum age of last-known-good manifests.

POLICY FRESHNESS AND REVOCATION

Publishing a restriction and enforcing it in a consuming service are separate events. Consumer caches and refresh behavior determine when the new policy becomes effective. Verify refresh intervals, acceptable stale-manifest age, and rejection when required policy cannot be established; do not assume immediate propagation from publication.

RETAINED INFORMATION

Live records, drafts, revisions, publication evidence, audit events, and backups have separate lifecycles. Restrict review and export to authorized roles and document the effective permissions for each store. Sanitizing live content does not remove earlier revisions or copies. Retention procedures must honor legal holds and verify deletion across applicable stores; no single platform-wide duration is established here.

05

Tool Interface Security

Agent Tool Interface (ATI) provides a governed point of access to enterprise tools. It brings registered gateway tools, MCP integrations, and approved REST API operations into a common interface for discovery and execution. Administrators control the available tool definitions, access requirements, usage policies, and connections to upstream services.

These controls address both sides of a tool call: the action an agent requests and the information returned to its context. Access controls determine whether a call may proceed. Usage limits constrain demand on connected systems. Argument rules and risk classifications support decisions by the agent runtime. Approved transformations can further restrict the request sent upstream and the result returned to the agent.

Authorization at the tool boundary

In its secure service configuration, ATI authenticates the calling workload and verifies its delegated credential against the requested operation. The authentication path checks workload identity, token authority and scope, and replay protection. Tool execution also checks the access tags required by the tool against the caller's grants for tool invocation.

These checks apply when a tool is called. Knowing a tool's name does not grant access to it. A call that fails the tool's access-tag requirements is treated as though the tool were unavailable, limiting disclosure of restricted tools. The execution path performs this check before consulting the result cache or selecting a production gateway.

For REST API tools, administrators own the connection, authentication configuration, and approved operation. Credentials are resolved by ATI from configured secret references; ordinary tool arguments cannot supply or replace the connection's authentication. Execution, live testing, mutating HTTP methods, and scripted transformations have separate deployment controls. Agent proposals remain drafts until they pass the applicable review and activation process.

Argument constraints and risk classification

A tool may be permitted while particular uses of it require additional restrictions. Argument rules express conditions on a call's inputs—for example, requiring a change-ticket reference when an operation targets a production environment. Rules can inspect nested fields, compare values, test membership, and apply conditions across arrays.

ATI validates these rules against the tool's input schema and versions them with a schema fingerprint. A schema change marks existing rules stale, making the change visible for revalidation. Rules support observation and rejection modes, allowing organizations to examine their effect before enabling rejection.

ATI distributes these rules to the agent runtime, which evaluates and enforces them before dispatch. ATI does not itself reject an invocation because it violates an argument rule. The protection therefore depends on the calling runtime implementing the rule contract.

Risk classifications provide a separate input to runtime policy. ATI supports five ordered classes: `minimal`, `low`, `moderate`, `high`, and `critical`. An unspecified classification is not treated as minimal risk. The effective classification combines the runtime's baseline for the tool kind, the gateway declaration, administrator overrides, and rules matching the actual arguments. The highest applicable classification wins.

This allows a tool to receive stronger treatment for a particular operation or target. For example, a recursive deletion can carry a higher classification than a normal lookup through the same interface. Classification supplies the signal; the runtime's configured policy determines whether the call needs confirmation, approval, or rejection.

Usage and execution limits

ATI can limit tool calls within fixed minute, hour, or day windows. Policies may cover every call to a tool or only calls whose arguments match configured conditions. Every matching enabled policy applies. Rejected calls identify the blocking limit and provide retry information.

The scope of enforcement depends on deployment configuration. Local counters apply independently to each ATI replica. Redis-backed counters provide shared counting across replicas. If Redis becomes unavailable, the configured fallback may use local counting or fail open; the operational interface exposes the active mode. The production profile must therefore specify the counting mode and its failure behavior.

API connections also impose request and response byte limits. The current creation defaults are 256 KiB for request bodies and 512 KiB for retained response bodies. A response exceeding its cap is marked as truncated. A transformation can reject a truncated response when completeness is necessary for safe interpretation. These connection limits are separate from the limits applied to Lua execution.

Controlled result caching

ATI can cache successful results from suitable read-only operations to reduce latency and load on upstream systems. Caching requires both an enabled cache engine and a policy for the tool. Policies define expiry, argument conditions, result-size limits, and whether entries may be shared globally, within a tenant, or with a single caller.

Tenant and caller identities used in cache keys come from verified authentication context. ATI checks tool access before looking up a result. A cache hit avoids upstream execution and does not consume the tool's rate-limit quota. Failed and timed-out results are not stored.

Cache scope must reflect the sensitivity and personalization of the underlying result. Tool authorization alone does not establish that every authorized caller should receive the same cached data. A globally shared result, for example, must be suitable for every caller eligible to retrieve it.

Expiry bounds reuse but does not guarantee freshness. Changes to upstream data, credentials, permissions within a scope, or response transformations require cache lifecycle management. Before tightening redaction, operators must disable caching, allow running calls to finish, flush previous results, and verify a fresh transformed response before re-enabling it.

Shaping requests and responses with Lua

Scripted API tools allow administrators to define the interface presented to the agent independently of the upstream API's data format.

The request hook receives the agent's arguments and limited invocation context. It can validate, normalize, remove, or construct fields before ATI binds the result to the configured API operation. The transformed output must satisfy the approved wire contract. ATI retains responsibility for request dispatch and connection authentication.

The response hook receives a bounded response representation, including HTTP status, body content, timing, and a truncation indicator. It can select approved fields, remove sensitive values, normalize structures, and reject unsuitable responses. Its output must satisfy the approved agent-facing output schema. Both hooks use the same approved version captured for the invocation.

For redaction, constructing a new result containing only approved fields provides a useful default. An upstream service can then add fields without automatically exposing them to the agent. An output schema that rejects additional properties reinforces that boundary. The script should also define how to handle missing fields, unexpected types, error responses, and truncation.

This controls the data returned to the agent at that boundary. It does not erase the original response retained in ATI's request records, remove information already present in a conversation, or redact request arguments automatically. Those surfaces require their own controls and retention policies.

Handling large tool results

NodeOne separates the data returned by a tool from the amount placed directly into the model's context. Large results are saved to the session's virtual filesystem, and the agent receives a file reference and a bounded preview. This preserves access to the retained data while preventing a single response from consuming an excessive share of the context window.

The agent can then use NodeOne's governed Bash environment to process the saved result. Utilities such as `jq`, `grep`, `awk`, `sort`, and `head` allow it to select fields, locate records, aggregate values, and extract relevant passages. Only the information needed for the next reasoning step needs to return to the model. These operations use the session's virtual filesystem and the sandbox's permitted execution capabilities.

Storage and context delivery remain bounded. The current general tool-result path replaces results larger than 40 KiB with a file reference and preview, and retains at most 5 MiB per result. Results exceeding the storage limit are explicitly marked as truncated; if file storage is unavailable, NodeOne returns a bounded inline result with a truncation notice. Saving a result to a file controls context consumption—it does not redact its contents. Required redaction must therefore occur before sensitive information reaches either the saved result or its preview.

Approval and containment of transformations

Lua source and its associated contracts are stored as immutable versions. Validation, approval, and activation are separate steps, and authors cannot approve their own versions. Execution requires an approved, active, non-revoked version whose digest and definition match the current tool and API revision. Changes require another version and review.

Lua runs in dedicated worker processes, with a fresh Lua state for each invocation. Scripts receive restricted libraries and bounded JSON inputs. ATI does not expose network or filesystem APIs, connection credentials, or host callbacks through the scripting interface. Limits cover source size, input and output size, nesting depth, memory, instruction count, and execution time. Workers that time out or fail are discarded and replaced.

This provides resource controls and crash isolation. The workers share ATI's container namespaces; they do not provide a separate operating-system sandbox. Independent review of the exact executable version remains an essential trust boundary.

Extending transformations across tool interfaces

ContextOne is extending this Lua capability across tool surfaces so organizations can apply consistent request shaping and response redaction wherever agents use tools. The verified implementation described here applies to scripted ATI API tools; uniform coverage across other surfaces is part of the ongoing rollout.

For that rollout, the security contract needs to remain consistent: transformations must run before protected data reaches the agent, transformed outputs must be validated, and transformation failures must not expose the original payload as a fallback. Cache reuse, error responses, and retained diagnostic records must be included in each surface's design. This allows organizations to apply a common redaction policy while preserving the authorization and execution controls of each tool interface.

REQUIRED DEPLOYMENT SETTINGS

Enable secure ATI authentication and replay protection. Restrict connection credentials and enable only the API execution, live-test, mutating-method, and scripted-transformation capabilities required by the deployment. Activate only approved Lua versions and require the calling runtime to enforce argument rules and risk policy.

Enable tool-limit policies and select shared Redis accounting where quotas span replicas. Local fallback and fail-open operation do not preserve a strict shared quota. If strict admission is required, verify a control that rejects new calls when shared enforcement is unavailable. Set request, response, worker, and execution limits explicitly.

Configure cache scope to match the result's authorization boundary. Permission, credential, or transformation changes require invalidation procedures; expiry alone does not establish current authorization.

RETAINED INFORMATION

Treat the transformed result delivered to an agent, original API request and response records, execution-job results, cached results, saved session files, and Lua test fixtures as separate record classes. Response redaction does not erase the original upstream response retained by ATI or redact request arguments automatically.

Restrict operational payload access and export to designated administrators under verified service permissions. The operational-log sweeper covers named operational tables and script test records; its setting must not be assumed to delete all API payloads. The deletion schedule and exact reader/exporter permissions for retained API request and response bodies require verification before production sign-off.

06

Structured Data Security

ContextOne separates the authority to access structured data from the machinery that executes a query. Structured Data Gateway (SDG) governs discovery, validation, access policy, and query submission. Data Access (DA) runs as a distinct sidecar process, plans and executes approved queries against connected sources, and returns results to SDG.

Agents work with logical catalog, table, and column names. Database connections, credentials, physical mappings, and security policies remain under administrative control. This allows agents to formulate useful queries without receiving direct database credentials or authority to change the controls applied to them.

Authenticated identity and attribute-based access

Agent tool calls reach SDG with a short-lived delegation token issued by the Session Router. The token binds the verified caller and tenant to the requested operation and a hash of its arguments. SDG verifies the delegation and trusted calling peer before processing the request. Names or identities supplied in a prompt or SQL statement do not replace this authenticated context.

SDG applies attribute-based access control (ABAC) to catalogs, schemas, relations, and columns. Requirements inherit additively through that hierarchy. Effective requirements also combine local SDG policy with approved AOS metadata, allowing either source to add restrictions.

For query operations, SDG uses the caller's grants associated with `sdg:read`. Permissions for administration or configuration do not automatically grant data-reading authority. The caller must satisfy the effective requirements for every referenced object, including columns used in predicates, joins, and expressions—not only columns returned in the result. Queries that reference inaccessible objects are rejected.

Validate, then execute

The normal agent workflow separates preparation from execution:

1. **Discover and describe.** The agent identifies accessible logical relations and their schemas.
2. **Validate.** The agent submits SQL to `validate_query`. SDG parses it, resolves its references, checks permissions and supported functions, and prepares the applicable row-security plan. Successful validation returns a reference to the stored plan.
3. **Submit.** The agent submits that validation reference through `submit_query`. SDG checks that it remains usable, rechecks access, verifies policy compatibility, and resolves the caller's current row entitlements.
4. **Execute and retrieve.** SDG creates a governed job, dispatches it to DA, tracks its progress, and returns a bounded result or an authorized result artifact.

Validation does not run the requested data query, although it can request planning information from the execution engine. It provides an opportunity to correct a query before execution and creates a record against which the subsequent execution can be compared.

A validation reference is not a permanent access grant. Submission checks its scope and validity and re-enforces ABAC. The submission interface can also validate supplied SQL itself; omitting a separate validation call does not bypass the authorization pipeline.

Structural query validation

SDG validates the parsed structure of a query. The supported agent-facing language permits read-only queries within a defined SQL profile. References must resolve against the logical catalog, and functions must satisfy the configured allowlist.

These checks extend into nested query expressions. A write hidden inside a common table expression, an unresolved reference, or a construct the validator cannot safely support is rejected before execution. Logical references are then compiled into the physical identifiers used by DA. Agents cannot use arbitrary physical names to step outside the governed catalog.

Row-level security controlled by the service

ABAC determines which tables and columns a caller may access. Row-level security determines which records within those objects the caller may receive.

For example, two advisers may have identical access to an accounts table while being entitled to different accounts. Both agents can submit the same query without a `WHERE` clause. SDG adds the appropriate restriction using each adviser's verified identity and assignments.

The agent does not supply the authoritative identity, entitlement values, policy selection, or security parameters. SDG obtains policies from configured governance sources and resolves entitlement facts through trusted providers. User-authored filters can narrow a query, but cannot remove the service-applied restrictions. Naming another person in a prompt does not change whose entitlements are evaluated.

SDG supports direct bindings between a column and an allowed-value set, as well as relational policies that follow relationships between tables. A contact, for example, may be visible because it is linked to an account assigned to the caller.

For relational policies, SDG wraps each protected base-table occurrence in a filtered subquery before it participates in the surrounding query. A separate guard checks that required filters remain present, including in nested expressions. Unsupported combinations fail closed. On catalogs configured to require relational policy coverage, a table without an applicable policy or explicit governed exemption causes rejection.

Entitlement values are resolved server-side at submission and bound through reserved policy parameters. An empty allowed-value set produces no matching rows. Provider failures, malformed facts, and unresolved required entitlements block submission rather than producing an unrestricted query.

These restrictions govern the execution plan before results reach the agent. DA may execute filters at the source or retain exact local enforcement where source semantics require it; row security does not depend on the model removing unauthorized records afterward.

Governance of security policy

For AOS-governed catalogs, row policies are published as immutable revisions in versioned bundles. Publication validates policy dependencies and activates the bundle atomically. SDG consumes a verified manifest under a configured freshness policy.

Policy definitions and personal entitlement values have separate lifecycles. AOS governs what the rule means and which facts it requires. The entitlement system supplies the current values for the verified caller. Querying data does not grant authority to edit either source.

Sealed execution and the DA sidecar

SDG seals the execution job using HMAC-SHA-256. The seal binds the job identity, compiled SQL, bound parameter values, schema fingerprint, execution limits, result mode, and connector execution configuration. This ties the executable request to the work SDG prepared.

SDG's worker adapter verifies the seal and checks the relevant live schema against the recorded fingerprint before dispatch. A failed integrity check or incompatible schema change prevents execution through that path.

DA runs separately from the SDG control plane as a sidecar reached over the private in-pod connection. It owns query planning, source connections, and execution; SDG owns the authorization decision.

SDG creates and verifies the job seal; DA does not independently verify it on receipt. DA's listener must therefore remain private and protected by configured runtime authentication. The separate process provides an execution boundary, but does not by itself prevent direct requests from bypassing SDG.

Audit and execution evidence

SDG retains linked records for validations, execution jobs, and query denials. Query History lets authorized administrators follow an attempt from its submitted SQL through validation, policy application, execution, and outcome.

The evidence includes:

- Caller, tenant, session, validation reference, and execution identity.
- Logical SQL, referenced objects, access requirements, and policy-version information.
- The row-policy rewrite, SQL sent to DA, and captured SQL sent to downstream sources.
- Submitted parameters, with server-controlled policy values handled separately.
- Job transitions, timings, returned row and byte counts, limits, failures, and denial stages.
- Available validation plans and execution evidence, including explicit indications when capture is missing, partial, or bounded.

The value-free row-policy rewrite is copied onto the durable job record, preserving evidence of the enforced restriction after the validation reference expires. Administrators can compare what the agent requested with the policy-applied statement and the work actually dispatched. Agent-visible diagnostics remain restricted.

The query log records execution evidence and outcome metadata rather than duplicating result rows. Result content remains in its authorized response or artifact.

Denial recording is deliberately non-fatal. If persistence fails, SDG still denies the request and reports the logging failure. The implementation therefore supports detailed auditing across validation and execution, but does not establish an unconditional guarantee that every rejection is durably recorded during a storage failure.

REQUIRED CONFIGURATION

Keep DA's listener private and enable its runtime authentication. Configure SDG's trusted delegation path, source credentials, query and result limits, and required row-policy coverage. A separate process or job seal does not replace protection of the DA endpoint. Test rejection when required entitlements, policy, or integrity checks cannot be established.

RETAINED INFORMATION

Validation records, denials, execution jobs, logical and compiled SQL, submitted parameters, source plans, and result artifacts may reveal sensitive information. Query History is an administrative surface; agent-visible diagnostics and artifact retrieval remain separately authorized. Production operations must verify the exact administrative reader and exporter roles.

Set separate retention periods for query evidence and result artifacts. Expiring a validation reference does not establish deletion of durable job evidence, and evidence-log cleanup does not establish deletion of result files or exports. Record and monitor deletion failures. A failed denial-log write must be observable even though the query itself remains denied.

07

LLM Interface Security

ContextOne governs model interactions at two boundaries. The NodeOne harness controls the content and capabilities available to an agent, selects its model route, and manages execution. The LLM Gateway authenticates model requests, controls access to provider credentials, coordinates shared capacity, and records usage.

The default configuration routes completions and remote embeddings through the gateway. Direct provider access from the harness is also supported through an explicit configuration choice. That choice changes where credentials and shared controls reside; the harness remains responsible for agent execution and routing policy.

Content guardrails in the harness

NodeOne applies native content guardrails at model-content boundaries. These cover incoming conversation content, tool results, generated tool arguments, and model responses. The implementation processes content fields separately from authentication, routing, and provider protocol metadata, preserving the integrity of the surrounding control information.

The default guardrail configuration enables identifier and credential redaction and profanity filtering. Recognized sensitive values are replaced with typed markers such as `[EMAIL_ADDRESS]` or `[CREDENTIAL]`. Structured processing preserves field context, including column names associated with tabular query results, so sensitive values can be recognized before their structure is lost through serialization.

Administrators can configure the enabled categories and language dictionaries or explicitly disable the guardrails. If enabled sanitization fails, the runtime blocks forwarding rather than falling back to the original content.

Streaming requires the same protection. A credential or identifier may span several model-output fragments. With guardrails enabled, NodeOne buffers model content for inspection before releasing it, preventing fragment boundaries from defeating recognition. Buffer limits bound this processing, and blocked or cancelled streams do not subsequently release their remaining content.

These are bounded, deterministic controls. They recognize supported patterns and structured fields; they do not guarantee detection of every sensitive value, harmful instruction, or prompt injection. Tool authorization, sandbox restrictions, and data-access policy remain necessary even after content has passed inspection.

Model output does not grant authority

The harness treats a model response as proposed content or action. A generated tool call still passes through the runtime's applicable argument checks, risk policy, approval requirements, and tool authorization.

The harness also owns the configured provider, model, and recovery route. The gateway transports the selected request; it does not independently grant an agent broader tools or permissions. Content returned by a model cannot change the deployment's credentials, access grants, or security configuration.

The current implementation places native content sanitization in the harness. The gateway's authentication and capacity controls are additional protections; they should not be described as a second, equivalent content-scanning layer.

Gateway authentication and credential custody

In gateway-backed deployments, provider credentials remain at the gateway. Harnesses submit logical credential references, and the gateway resolves those references to the configured secrets. This reduces the number of processes that need to hold provider credentials.

In secure mode, the gateway validates Session Router delegation for the gateway audience and uses authenticated transport. Configured client authorization rules restrict which credential references each principal may use. A caller's ability to reach the service therefore remains separate from its authority to spend a particular provider credential.

Credential restrictions must be configured explicitly: the current authorization implementation permits credential-reference access when its client-rule list is empty. Production deployments must configure and verify these restrictions.

Shared rate and capacity limits

The gateway can coordinate provider capacity across the harness fleet using Redis-backed accounting. Supported dimensions include request counts, input and output tokens, total tokens, and provider-specific limits such as image requests. Provider profiles map requests onto the relevant capacity scope, including shared model families or regional capacity where applicable.

Before dispatch, the gateway performs a capacity check and reserves capacity. A blocked request receives a retry indication without reaching the provider. After the attempt, the gateway reconciles the reservation with reported usage or failure information. Provider rate-limit feedback also informs subsequent admission decisions. This reduces the risk of multiple harnesses independently consuming the same shared allowance.

Rate limiting is configurable and disabled by default in the underlying configuration. Redis failures follow the configured degraded or fail-open behavior; fail-open is the configuration default. A deployment requiring strict capacity enforcement must explicitly select and verify its failure policy.

Bounded requests and recovery

The harness checks the estimated request size against the selected model's context budget, reserving space for output. Output-token limits bound individual generations. Context management and bounded tool-result delivery help prevent large histories or tool responses from producing uncontrolled request growth.

Recovery is also configured. Retry counts, delays, and capacity waits have explicit bounds, including limits on the number of waits and total waiting time. These controls prevent an unavailable or rate-limited provider from causing an indefinite retry loop.

For dispatches using NodeOne's agent execution protocol, the harness additionally reserves against the dispatch's token budget before issuing model requests. It reduces the permitted output to the remaining allowance and stops when that allowance cannot cover another request. This execution-budget mechanism is distinct from the model's context-window limit.

Cost visibility and spending controls

The gateway records model usage and provides cost breakdowns using configured catalog prices. Accounting distinguishes input, output, and cache-related tokens where the provider supplies them, avoiding double-counting cached input. Usage with insufficient pricing information remains visible without being presented as zero cost.

These figures support operational oversight, but are estimates rather than provider invoices. Cost reporting alone does not stop further requests.

Budget-bound Gym executions have an additional admission mechanism. The Session Router is the balance authority, and the gateway uses verified execution bindings to reserve budget before provider dispatch. Provider attempts are recorded durably and settled against reported usage. Attempts known not to have dispatched can release their reservations; uncertain outcomes remain explicitly unresolved rather than being assumed free.

This prevents concurrent work within that budgeted execution from relying only on after-the-fact cost totals. Its scope should remain explicit: the verified reservation path applies to budget-bound Gym work, and does not establish a universal dollar cap for every ordinary chat.

Observability and retained content

Both direct and gateway-backed calls use the shared provider layer and emit correlated model telemetry. Gateway accounting adds a central view of requests, usage, capacity decisions, and failures, allowing operators to distinguish requests blocked before dispatch from attempts that reached a provider.

Operational metadata and retained request bodies need separate treatment. The gateway includes bounded, best-effort capture of exact outbound request JSON. Such records may contain sensitive business content even after configured guardrails have run, and require appropriate access and retention controls. Content sanitization should not be described as a guarantee that telemetry and diagnostic storage contain no sensitive information.

REQUIRED PRODUCTION SETTINGS

Configure explicit credential-reference ACLs; an empty client-rule list permits access. Enable rate limiting, configure the required shared dimensions, and test the selected Redis failure policy. The underlying defaults disable rate limiting and select fail-open behavior. Strict shared limits require a verified admission path that rejects work when enforcement is unavailable.

Enable and test the required harness guardrails, including supported categories and streaming behavior. Record any direct-provider routes and their credential and capacity controls. Budget-bound Gym reservations do not establish a universal spending cap for ordinary chats.

RETAINED REQUEST CAPTURES AND USAGE RECORDS

When request capture is configured, the gateway can retain exact outbound request JSON, which may include prompts, conversation content, and tool results. The inspected configuration defaults `events_retention_days` to 14 days. An hourly cleanup deletes expired capture records and then unreferenced body blobs. Cleanup failures are logged; this is scheduled deletion, not an exact deletion deadline.

The inspected capture-export endpoint accepts the Session Router. The downstream human authorization and export path requires deployment verification. Restrict storage access and exported copies separately.

Usage rollups have a separate default retention of 800 days. Request-capture cleanup does not establish deletion of usage rollups, session histories, provider-held records, exports, or backups. Record effective settings and ownership for each store.

08

Data Room Security

ContextOne Data Rooms provide durable storage for enterprise documents, shared working files, and agent-generated artifacts. Users and agents access the same governed service through browsing, file operations, search, and authorized downloads.

Authorization combines two independent requirements: the caller's token must permit the operation, and the applicable room or file policy must grant access. A token permitting writes does not make its holder a contributor to every room. Equally, room ownership does not allow a caller with a read-only token to modify content.

Two access-control modes

Data Rooms support membership-based rooms and rooms governed by per-file access control entries, or ACEs. The mode belongs to the room. A caller cannot select a different mode to obtain access.

Membership-based rooms support collaboration on content managed within ContextOne. ACE rooms support connector-managed content whose permissions originate in an external system such as SharePoint. Both modes enforce authorization within the Data Room service, independently of instructions given to the agent.

Fine-grained access through ACEs

In an ACE room, access is evaluated for each file against the verified caller's identity and group memberships. Room membership does not grant access to the files it contains.

ACEs associate users or groups with allow or deny decisions. Evaluation denies access by default, and a matching denial takes precedence over an allowance. Files without an applicable allowance remain inaccessible. Where a policy contains multiple required clauses, the caller must satisfy each clause.

These restrictions apply to discovery as well as retrieval. Read, history, and search queries filter against file permissions, preventing an unauthorized file from being exposed through its path, preview, or historical content. Knowing a filename or retaining an earlier reference does not confer continuing access.

SharePoint as a source-managed example

The ContextOne SharePoint connector mirrors documents from SharePoint and OneDrive, together with their source permissions. It also renders supported modern Site Pages into Markdown. The connector reads Microsoft Graph using configured application credentials; agents read the resulting mirror through Data Rooms using their authenticated identity.

This separates the connector's ability to ingest content from the user's authority to consume it. A connector may need access to a document library to maintain the mirror, but that application access is not transferred to agents. Data Rooms evaluates the mirrored permissions for each requesting user.

Mirror rooms are bound to the authenticated connector workload and connector identity. An existing room is reused only after that binding is verified. Matching a room name is insufficient for another connector or tenant to take control of it.

The connector translates supported source permissions into file-level ACEs. Permission updates carry a content fingerprint and an increasing source generation. Data Rooms rejects stale generations, preventing an older update from overwriting newer permission state. An unchanged permission fingerprint avoids an unnecessary replacement.

Webhooks, synchronization, and rescans

Microsoft Graph change notifications trigger synchronization. The connector validates the notification's shared `clientState` secret before accepting it for processing. Missing or mismatched values are dropped. A notification signals that the connector should examine the source; it does not itself supply trusted file contents or permissions.

The connector maintains and renews Graph subscriptions and uses drive change-tracking cursors to retrieve updates. Periodic synchronization and metadata sweeps provide a backstop when notifications are missed or processing is interrupted. The component-chart defaults schedule a safety-net resynchronization every six hours and a metadata sweep every twelve hours; deployments can configure these intervals.

Data Rooms serves stored content and mirrored permissions. Reading or searching a file does not trigger a fresh SharePoint permission check. Updates are applied file by file, and extraction and embedding can complete later than content ingestion.

This creates an explicit freshness boundary. Source edits and permission revocations take effect in the mirror after successful synchronization. Notification delivery, source availability, and processing failures can extend that delay; the configured rescan interval is not a guaranteed maximum.

If permission retrieval fails, the connector retains the previously mirrored ACL. A new file without an allowance remains inaccessible, but an existing allowance can remain effective until the refresh succeeds. Operators must therefore monitor permission synchronization separately from content ingestion. A completed content pass does not prove that every source revocation has reached the mirror.

PERMISSION FRESHNESS AND REVOCATION HANDLING

Data Rooms authorizes reads against the latest successfully mirrored permissions. If a permission refresh fails, the previous ACL remains effective. Access revoked in SharePoint may remain available in the mirror until synchronization succeeds. Webhooks and periodic rescans reduce delay but do not establish a guaranteed revocation deadline.

Permission-fetch failures are logged, and generation checks prevent older updates from replacing newer state. A generation check protects update ordering; it does not prove that stored permissions are recent.

Production operations must monitor permission-refresh failures independently of content ingestion and establish the age of the last successful permission refresh where that evidence is available. If freshness cannot be established, treat it as unknown. Define a threshold for containment, a tested procedure to make affected content unavailable, reconciliation steps, and evidence required before restoring access.

The implementation review did not establish an automatic maximum-ACL-age cutoff, complete per-file freshness monitoring, or automatic quarantine. These are controls to verify or implement if required by the deployment, rather than existing guarantees.

Membership-based Data Rooms

Ordinary, non-ACE rooms use explicit membership roles:

ROLE	AUTHORITY
Reader	Read authorized files
Contributor	Read, create, modify, move, and delete authorized files
Owner	Contributor capabilities and membership administration

These roles are enforced alongside token scope. A reader cannot write merely because their token includes `data_room:write`.

Ordinary rooms default to members-only access. Owners and tenant administrators can optionally enable read access for authenticated users in the same tenant, including eligible user-owned agents acting through their signed owner identity. This setting grants no write or administrative authority and does not apply to connector rooms or agent-home rooms.

Agents cannot administer room membership. Persistent agent-home rooms have additional bindings to the stable agent and, for owner-scoped homes, its owner. A stable agent identity provides continuity and attribution without granting general access to other rooms.

Access tags and derived information

Files can carry required access tags in addition to their room's authorization policy. An otherwise authorized caller must hold every required tag. The restrictions apply to content, history, lineage, original-byte downloads, and mutations. Listing, grep, and semantic search omit files whose requirements are not satisfied.

Harness writes attach the session's active labels to generated files. Overwriting a labelled file preserves its existing restrictions and adds any new ones; it cannot remove restrictions simply by omitting them from the request. This helps carry the classification of consumed information into artifacts produced by the agent.

Search, evidence, and historical access

Search is an authorized view of stored content. Semantic results may include a version, passage preview, and evidence reference, but the reference is not a transferable permission grant. An evidence read rechecks access, source liveness, version and content identity, and the relevant indexing state before returning the passage.

File history is also governed. Updates create immutable versions, and mutations require the caller's observed version to match the current head. Conflicting edits are rejected so the caller can reread before proceeding. Restoring an earlier version creates a new current version and requires write authority.

Immutable content-addressed storage allows identical bytes to be reused across files and rooms, while authorization remains attached to the requested room and file. Sharing an underlying blob does not merge those access policies.

Audit and operational visibility

Data Rooms retains file versions, events, and lineage linking artifacts to the actors and sessions that produced them. Authorized users can inspect file history and provenance, while room-administration events record access-policy changes.

Operational views respect the same visibility boundaries. SharePoint mirror dashboards expose a permission-filtered inventory rather than unrestricted room-wide activity or membership information. Content availability, successful extraction, embedding readiness, and permission freshness are distinct states: none should be inferred solely from a file count or a successful search.

REQUIRED CONFIGURATION

Configure room mode, membership, tenant-wide read settings, file access tags, and connector ownership explicitly. Restrict SharePoint application permissions and protect webhook secrets. Verify subscription renewal, resynchronization, and permission-refresh monitoring under the deployed intervals.

RETENTION AND DELETION

Live documents, immutable versions, original-byte blobs, extracted text, embeddings, derived artifacts, events, lineage, exports, and backups have distinct lifecycles. File and history access remains governed by the applicable room and file controls; direct storage administration and export permissions require separate review.

Set retention and deletion procedures for each class, including shared blobs and derived search data. Logical deletion or loss of search visibility does not by itself establish physical erasure. Revoking future access cannot retract copies already delivered to a user or agent.

Talk to us about ContextOne security.

Our team can walk through the architecture, the controls described in this paper, and the configuration your deployment requires.

EMAIL

info@ai.one →

WEBSITE

ai.one →