Web Authorization Protocol (OAuth)
Internet-Draft
Intended status: Standards Track
Expires: 17 September 2026

N. A. Niyikiza
Tenuo

16 March 2026

Attenuating Authorization Tokens for Agentic Delegation Chains
draft-niyikiza-oauth-attenuating-agent-tokens-00

Abstract

This document defines Attenuating Authorization Tokens (AATs), a JWT-based credential format for secure delegation in AI agent systems. An AAT encodes which tools an agent may invoke and with what argument constraints. Any holder can derive a more restrictive token offline that narrows or maintains scope but cannot expand it. This invariant is cryptographically enforced and verifiable offline by any party holding the root token’s trust anchor key.

This specification extends the Rich Authorization Requests format (RFC 9396) with delegation-chain semantics and defines a typed constraint vocabulary for tool-level argument restrictions. The accompanying chain verification algorithm enforces the monotonic attenuation invariant at each delegation step and requires no network contact with the root issuer.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 17 September 2026.

Copyright Notice

Table of Contents

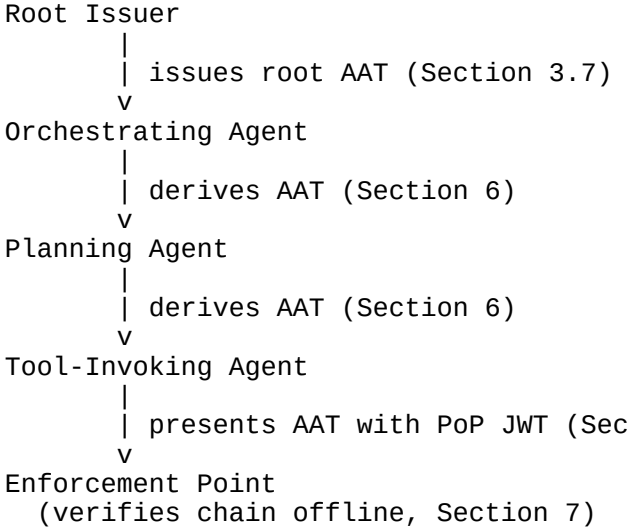
1. Introduction

AI agent systems increasingly delegate tasks to chains of autonomous agents, each invoking tools on behalf of a user or service. Today, the tokens that authorize these invocations are typically scoped to the principal — the user or service account — not to the task the agent is performing. Even when an OAuth scope narrows the token to a subset of APIs, it does not express which tools, with which argument values, a particular agent should use for a particular task. The token that checks flight availability also authorizes completing a purchase and charging a corporate card. A prompt injection attack, a model hallucination, or a compromised sub-agent can exploit this gap, exercising authority the agent should never have needed. This is the confused deputy problem [HARDY88] applied to agentic delegation.

This problem is compounded by a gap in existing infrastructure. The WIMSE architecture [WIMSE-ARCH] provides mechanisms for establishing workload identity and propagating it across service boundaries. OAuth 2.0 [RFC6749] provides token issuance and scoping. Neither provides a mechanism for a token holder to derive a narrower token and pass it downstream. Without delegation-aware semantics, the only options are to trust every agent in the chain with the full token or to require each delegation step to contact the authorization server. The latter is impractical for agentic workflows that execute tool invocations in rapid succession, operate across trust boundaries, or run in environments with intermittent connectivity.

Capability-based systems [DENNIS66] solve both problems. Authority is carried by unforgeable tokens scoped to specific operations; a holder can attenuate a capability before passing it on, but cannot amplify it [SALTZER75]. This document defines such a mechanism for OAuth-based agent systems, complementing WIMSE's identity layer with a delegation and attenuation layer.

The following diagram shows the delegation flow this specification enables:



At each derivation step, the derived token's scope is a subset of the parent's: scope can only narrow or stay the same, never widen. The enforcement point verifies the complete chain using only the root token's trust anchor key; no network calls are required. How token chains are carried to enforcement points is deployment-specific; this document does not define a transport binding.

1.1. Limitations of Existing OAuth Mechanisms for Agentic Delegation

OAuth 2.0 Token Exchange [RFC8693] enables a principal to obtain a new token with reduced scope by contacting the authorization server. The server enforces the scope reduction. This requires a synchronous round-trip to the authorization server at each delegation hop. In multi-agent chains, this makes the authorization server a participant in every delegation decision, coupling the delegation topology to authorization server (AS) availability. [RFC8693] supports representing prior delegation actors via nested act claims, but those claims are informational for access control decisions rather than a cryptographically self-verifiable attenuation chain. The AS mediates each grant independently, and no mechanism ensures that downstream delegation intent remains consistent with the original authorization scope.

Rich Authorization Requests (RAR) [RFC9396] extend OAuth tokens with structured authorization detail objects, enabling expressive capability descriptions. RAR addresses the expressiveness problem. It does not define how a token holder can produce a narrower token, or how a chain of such derivations can be verified.

Proposals to extend the authorization code flow with explicit agent consent, such as introducing a requested_actor parameter at the authorization endpoint, address who the agent is and whether the user approved the delegation. They do not constrain which tools the agent may invoke or with what argument values. AATs are complementary: they scope authority to specific tools and arguments after identity and consent have been established.

To the author's knowledge, no existing OAuth standard defines a delegation chain protocol with a cryptographically enforced attenuation invariant and offline chain verification.

1.2. Design Goals

1. *Least privilege at the invocation boundary.* An agent's authorization token encodes which tools it may call and with what argument constraints, scoped to the task, not to the full authority of the calling principal.

2. *Offline derivation.* A token holder can derive a more restrictive token without contacting the root issuer.

3. *Independent chain verification.* Any enforcement point holding the trust anchor can verify the complete delegation chain without network calls.

4. *Cryptographically enforced attenuation.* A derived token cannot grant broader authority than its parent.

5. *JWT interoperability.* Tokens are representable as signed JWTs [RFC7519], allowing deployments to verify chains using existing JSON Object Signing and Encryption (JOSE) infrastructure without new cryptographic dependencies.

1.3. Relationship to Prior Work

Macaroons [MACAROONS] introduced the concept of attenuating tokens with contextual caveats. Macaroons use HMAC chaining, which provides attenuation but not proof of possession, and express caveats as free-form predicates evaluated at the target service at runtime. This specification adds asymmetric proof of possession, structured tool-level capability claims, and a typed constraint vocabulary. It defines a normative subsumption relation, enabling any party holding the chain to verify monotonicity structurally, without predicate evaluation at a central service.

Biscuit [BISCUIT] extends the Macaroons model with asymmetric public key signatures and offline attenuation, addressing the proof-of-possession gap. Biscuit expresses authorization policies in a Datalog variant, requiring a logic engine at verification time. This specification uses structured constraint types decidable by structural analysis and defines a typed delegation-chain model with explicit token roles and chain-position claims. A detailed comparison appears in Appendix A.

The capability-based security model underlying AATs draws on [DENNIS66], which introduced capabilities as unforgeable tokens of authority, and [MILLER06], which formalized the principle of least authority (POLA) and the attenuation property in object-capability systems. AATs apply these principles at the protocol layer: each token is a capability scoped to specific tools and arguments, and derivation can only attenuate, never amplify, the authority it carries.

[DEEPMIND26] argues that safe multi-agent delegation requires explicit transfer of authority, responsibility, and trust at each delegation step, with bounded operational scope. [CAMEL25] shows that capability-based controls enforced at the tool boundary can provide provable security properties in an agentic framework. These results motivate a protocol-layer mechanism that encodes delegation scope in verifiable credential artifacts enforced independently of model behavior. AATs realize one protocol-layer approach to that goal.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Attenuating Authorization Token (AAT): A signed JWT as defined in this document. An AAT encodes tool-level capability claims and supports offline derivation of derived tokens with authority equal to or narrower than the parent’s.

Root Token: An AAT with no parent token, del_depth: 0, and par_hash absent. A root token may carry either aat_type: "delegation" or aat_type: "execution": a root issuer may mint an execution token directly when an agent requires immediate tool invocation authority without an intermediate delegation step. A root token is signed by the private key corresponding to a trust anchor and establishes the authority ceiling for all derived tokens. A root token is a positional role, not a distinct aat_type value: the two aat_type values are "delegation" and "execution".

Root Issuer: The entity that mints root tokens. The root issuer holds the private key corresponding to a trust anchor and is responsible for verifying agent identity and requested authority before issuance.

Token Holder: The entity that possesses an AAT and the private key corresponding to its cnf.jwk claim. The token holder is the party authorized to present the token, derive further tokens from it, or both, depending on the token’s aat_type.

Derived Token: An AAT produced by a token holder from a parent AAT, also referred to as a child token. A derived token’s authority is a subset of its parent’s authority (equal or narrower). Derivation does not require a round-trip to the root issuer.

Delegation Token: An AAT that authorizes its holder to derive further tokens for a specified set of tools. The name reflects what the holder is authorized to do, not who issued the token. A delegation token does not authorize tool invocation. A delegation token may itself be derived from a parent token; it is not necessarily produced by the root issuer.

Execution Token: An AAT that authorizes its holder to invoke a specific set of tools subject to argument constraints. An execution token holder may derive tokens with authority equal to or narrower than their own, subject to the depth limit of the chain.

Tool: An addressable function or API operation that an agent may invoke. A tool is identified by a string identifier. Tool identifiers SHOULD be URIs ([RFC3986]); see Section 3.3.1 for requirements. Tool identifiers MUST NOT contain characters that are subject to Unicode normalization (such as combining characters or characters with multiple canonical forms), as normalization-sensitive identifiers can produce inconsistent matching behavior across implementations.

Argument Constraint: A predicate over a tool argument value that the argument MUST satisfy for the invocation to be authorized. Constraints are evaluated at the enforcement point before invocation.

Capability Claim: The set of (tool, argument constraints) pairs encoded in an AAT's authorization_details claim.

Attenuation: The process of deriving a token with a capability claim that is a subset of the parent token's capability claim. Attenuation is the only permitted direction of derivation.

Chain: An ordered sequence of AATs from root to leaf, where each token was derived from its predecessor.

Leaf Token: The last token in a chain. The leaf token is the one presented to the enforcement point for authorization. A leaf token MUST have aat_type: "execution". The PoP JWT is signed by the private key corresponding to the leaf token's cnf.jwk.

Enforcement Point: The component that receives a tool invocation request, verifies the presented token chain, evaluates argument constraints, and permits or denies execution.

Trust Anchor: A public key that enforcement points are configured to trust as the root of a delegation chain. Root tokens are signed by the private key corresponding to a trust anchor.

Proof of Possession (PoP): A cryptographic demonstration that the presenter of a token controls the private key corresponding to the public key bound in the token's cnf claim. In this specification, the token holder presents the chain and signs the PoP JWT using the same private key.

3. Token Types and Structure

3.1. Token Types

This specification defines two token types, distinguished by the aat_type claim.

Delegation Tokens (aat_type: "delegation") enumerate the tools for which the holder may derive further tokens; they do not authorize tool invocation. A delegation token holder MUST NOT present a delegation token to an enforcement point to invoke a tool. Root delegation tokens are typically issued by a root issuer to orchestrating agents, but delegation tokens may also appear at intermediate hops in a chain, derived from a parent delegation token with narrower scope. A derived token from a delegation token MUST

NOT authorize tools absent from the delegation token's authorization_details. The capability monotonicity invariant (I4, Section 4.5) applies at every delegation step: derived tokens can only narrow the authorized tool set and argument constraints relative to the delegation token.

Execution Tokens (aat_type: "execution") authorize tool invocation. They enumerate specific tools and the argument constraints that apply to each. Execution tokens are presented to enforcement points at tool invocation boundaries.

The separation between delegation and execution authority is a structural property of the token, not a policy decision left to enforcement points. This moves the authorization boundary from something an enforcement point must judge to something any enforcement point can read directly from the chain, and enables type-transition key separation (Section 8.12) to be structurally enforced rather than left to convention.

Any type transition is permitted, provided del_depth < del_max_depth in the parent token. A holder of either token type may derive a token of either type, narrowing or maintaining the authorized tool set and argument constraints. The capability monotonicity invariant (I4, Section 4.5) applies at every derivation step regardless of the type transition.

A derived token whose aat_type differs from its parent's aat_type MUST have a cnf.jwk whose JWK Thumbprint ([RFC7638]) differs from the parent's cnf.jwk thumbprint. This ensures structural separation between planning authority and invocation authority: an entity that delegates cannot use the same key to invoke, and an entity that invokes cannot use the same key to delegate. When aat_type is preserved across a derivation step, same-key derivation is permitted. The security rationale for type-transition key separation is discussed in Section 8.12.

A root issuer MAY issue an execution token directly, bypassing the delegation step, when an agent requires immediate tool invocation authority without intermediate planning authority.

3.2. Common Claims

The following claims appear in both token types. All claims listed as REQUIRED MUST be present. Claims listed as OPTIONAL MAY be omitted; their absence carries the semantics described in the table.

Claim	Type	Required	Description
jti	string	REQUIRED	Unique token identifier. SHOULD be a UUIDv7 value. When a UUID is used, it MUST be encoded as a lowercase hyphenated string in the form xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx per [RFC9562].
iss	string	REQUIRED	Identifier of the entity that signed this token. For root tokens, MUST be a URI identifying the root issuer. For derived tokens, MUST be a JWK Thumbprint URI as defined in [RFC9278], using the SHA-256 hash algorithm: urn:ietf:params:oauth:jwk-thumbprint:sha-256:<thumbprint>, where <thumbprint> is computed per [RFC7638].
iat	NumericDate	REQUIRED	Time at which the token was issued. MUST NOT be more than MAX_IAT_SKEW in the future relative to the enforcement point's clock (see Section 4.4). In a chain, a derived token's iat MUST NOT be earlier than its parent's iat.
exp	NumericDate	REQUIRED	Time at which the token expires. MUST be greater than iat. MUST NOT exceed iat plus MAX_TOKEN_LIFETIME (see Section 4.4).
cnf	object	REQUIRED	Confirmation claim [RFC7800]. MUST contain jwk with the holder's

			public key. The jwk value MUST be a public key; private key material MUST NOT appear in this field.
aat_type	string	REQUIRED	Token type. MUST be "delegation" or "execution".
del_depth	integer	REQUIRED	Delegation depth. 0 for root tokens. Incremented by exactly 1 at each derivation step (see Section 4.3).
del_max_depth	integer	REQUIRED	Maximum delegation depth permitted in this chain. MUST be a non-negative integer not exceeding the implementation's MAX_DELEGATION_DEPTH (Section 4.3).
par_hash	string	MUST (derived) / MUST NOT (root)	Base64url-encoded SHA-256 digest of the parent token's JWS Signing Input, using base64url encoding without padding as defined in [RFC7515] Appendix C. MUST be absent in root tokens. MUST be present in all derived tokens.
authorization_details	array	REQUIRED	Tool capability claims. Format defined in Section 3.3.

Table 1

Implementations MUST support Ed25519 [RFC8032] for token signing and verification. Implementations MAY support additional algorithms.

In both root and derived tokens, iss is a URI. For root tokens, iss is a URI identifying the root issuer, consistent with conventional OAuth usage. For derived tokens, iss is a JWK Thumbprint URI [RFC9278] that encodes the SHA-256 thumbprint of the signing key. This makes I1 verifiable offline: the enforcement point can confirm that the thumbprint embedded in derived.iss matches parent.cnf.jwk without any external lookup.

This specification intentionally omits the sub claim. In conventional OAuth tokens, sub identifies the resource owner or principal on whose behalf the token is issued. In an AAT chain, the holder's identity is fully determined by cnf.jwk: the entity presenting the token proves possession of the private key corresponding to cnf.jwk. Including a sub claim would introduce an additional identity binding that is not cryptographically enforced by this specification and could be set arbitrarily by any delegating party. Implementations that require a human-readable subject identifier MAY convey one in additional JWT claims outside this specification (see Appendix B.7).

3.3. Capability Claims via authorization_details

This specification profiles [RFC9396] for tool-level capability claims. Each entry in the authorization_details array MUST have type set to "attenuating_agent_token". The entry MUST include a tools member that maps tool names to argument constraint sets.

```
{
  "authorization_details": [
    {
      "type": "attenuating_agent_token",
      "tools": {
        "read_file": {
          "path": {
            "constraint_type": "pattern", "value": "/data/*"
          }
        },
        "search_index": {
          "query": {
            "constraint_type": "pattern", "value": "*public*"
          },
          "limit": { "constraint_type": "range", "max": 100 }
        }
      }
    }
  ]
}
```

A tool entry with an empty constraint map {} is valid and indicates that the tool is authorized without argument restrictions.

When a tool entry contains one or more argument constraints, the enforcement point operates in closed-world mode for that tool invocation: any argument not named in the constraint map MUST be rejected. A constrained argument that is absent from the invocation MUST also be rejected. The presence of a constraint asserts that the issuer has reasoned about that argument. An invocation that omits it has not been validated against that reasoning. This is a security property, not a configuration option.

Issuers who wish to permit an argument to be omitted MUST NOT include a constraint for it in the constraint map. There is no "optional constraint" mechanism; the constraint map is a closed specification of the required invocation shape. To authorize an argument without restricting its value, use a wildcard constraint (see below).

A token issuer that wishes to allow unconstrained arguments alongside constrained ones MUST explicitly include a wildcard constraint for each argument that should be unrestricted. A wildcard constraint satisfies closed-world mode while permitting any value for that argument (see Section 3.4). Enforcement points MUST enforce closed-world mode and MUST NOT permit unconstrained arguments when any constraint is present for the tool (see Section 7, step 6b).

The authorization_details array MAY contain entries of other types alongside attenuating_agent_token entries, consistent with the extensibility model of [RFC9396]. Enforcement points implementing this specification process only entries with type set to attenuating_agent_token and MUST ignore entries of other types. An authorization_details array containing multiple entries with type: "attenuating_agent_token" is invalid; the tools map in a single entry provides sufficient structure for all tool-level capability claims.

3.3.1. Tool Identifier Requirements

Tool identifiers are the keys of the tools map in an authorization_details entry. The following requirements apply.

Tool identifiers MUST be unique within the tools map of a single token. An authorization_details entry containing duplicate tool identifier keys is malformed and MUST be rejected.

Tool identifiers SHOULD be URIs ([RFC3986]). URI-format identifiers provide namespace isolation across agents and prevent semantic collision when multiple agents in a deployment expose tools with identical local names.

When URI-format identifiers are used, the URI SHOULD be scoped to the authority of the agent that exposes the tool. The authority component SHOULD correspond to the agent's workload identity or a domain controlled by the agent's operator. The URI SHOULD include a version component or content hash to ensure that all parties in the chain reason about the same tool schema. For example:

https://billing-agent.example.com/tools/charge/v2

Tool identifiers that are not URIs MAY be used in single-agent deployments where namespace collision is not a concern. Deployments spanning multiple agents or trust domains SHOULD use URI-format identifiers.

A tool identifier carries no inherent authorization semantics beyond naming a capability. The root issuer is responsible for verifying that the tool identifier is meaningful and that the requesting agent is authorized to claim identifiers under the tool URI's authority component before minting a root token (Section 3.7.3).

3.4. Argument Constraints

Each argument constraint is an object with a constraint_type member and type-specific members. The following constraint types are defined normatively. The check predicate and subsumes relation for each type are normative: two independent implementations MUST produce identical results when evaluating either predicate against the same inputs.

For constraint types where this specification does not define a general containment algorithm (specifically pattern, regex, cel, not, and any), this specification prescribes a conservative syntactic strategy. The strategy is sound: it never incorrectly accepts a non-subsuming constraint. It is conservative: it may reject constraints that are semantically subsuming but cannot be proven so syntactically. Deployments requiring richer policy expressiveness with full subsumption support SHOULD use a registered extension constraint type built on a language with a decidable containment algorithm (see Section 3.5 and Appendix C).

constraint_type	Additional Members	Semantics
exact	value (any scalar)	Argument MUST equal value exactly.
pattern	value (string)	Argument MUST match the glob pattern. See

		below for syntax.
range	min (number, optional), max (number, optional), min_inclusive (boolean, optional, default true), max_inclusive (boolean, optional, default true)	Argument MUST be a number satisfying the specified bounds. Both bounds are optional. min_inclusive and max_inclusive control whether the respective bound is included in the valid range; both default to true (closed interval).
one_of	values (array)	Argument MUST be a member of values.
not_one_of	excluded (array)	Argument MUST NOT be a member of excluded.
contains	required (array)	Argument, which MUST be an array, MUST contain every element listed in required.
subset	allowed (array)	Argument, which MUST be an array, MUST be a subset of allowed.
regex	pattern (string)	Argument MUST match the regular expression. See below for dialect and subsumption notes.
cel	expression (string)	Argument MUST satisfy the Common Expression Language (CEL) expression, which MUST return a boolean. See below for subsumption rules.
wildcard	(none)	Any value is accepted.
all	constraints (array)	Logical AND of nested constraints. See Section 4.5 for subsumption rules.

any	constraints (array)	Logical OR of nested constraints. See Section 4.5 for subsumption rules.
not	constraint (object)	Logical negation of a nested constraint. See Section 4.5 for subsumption rules.

Table 2

Pattern glob syntax. The pattern constraint uses the following glob syntax. * matches any sequence of characters not containing a path separator. ? matches any single character. [abc] matches any single character in the set. [!abc] matches any single character not in the set. The token ** (double-star) has no defined semantics in this specification and MUST NOT appear in pattern values. Brace alternation (e.g., {a,b}) has no defined semantics in this specification and MUST NOT appear in pattern values; deployments requiring alternation SHOULD use any with multiple pattern constraints. See Section 4.5 for subsumption rules.

Regex dialect and subsumption. The regex constraint does not standardize a dialect; implementations MUST document the dialect they support. This specification does not define a portable containment algorithm for regular expressions; equality of pattern strings is the normative conservative subsumption strategy. See Section 4.5.

CEL check predicate. The cel expression MUST evaluate to a boolean. The subsumption strategy for cel uses balanced-parentheses conjunction; see Section 4.5 for the normative rules and Section 8.13 for the security rationale.

Enforcement points MUST reject invocations where any argument violates its associated constraint. Enforcement points MUST deny authorization if they encounter a constraint_type they do not recognize (fail-closed behavior). This fail-closed rule applies only to constraint types within authorization_details. Enforcement points MUST ignore unrecognized top-level JWT claims; a token MUST NOT be rejected solely because it contains claims outside those defined in this specification.

Composite constraint types (all, any, not) are recursive. MAX_CONSTRAINT_DEPTH is an implementation-defined finite integer specifying the maximum nesting depth of a constraint tree.

Implementations MUST enforce a finite MAX_CONSTRAINT_DEPTH to prevent resource exhaustion from pathologically deep constraint trees. A value of 32 is RECOMMENDED. Enforcement points MUST reject any constraint tree whose nesting depth exceeds MAX_CONSTRAINT_DEPTH.

Domain-specific constraint types such as path containment, URL safety, or IP range matching may be registered in the extension registry (Section 3.5).

3.5. Extension Constraint Registry

Implementations MAY define extension constraint types beyond those listed in Section 3.4. Extension constraint types MUST be registered in the IANA AAT Constraint Type Registry defined in Section 9.3. The registry exists to preserve security and interoperability in the presence of domain-specific constraints; it is not a requirement that all implementations support arbitrary extensions. An enforcement point that does not recognize a registered extension type MUST deny authorization (Section 3.5.2), but it is not required to implement that type.

3.5.1. Attenuation Compliance Requirement

The capability monotonicity invariant (I4, Section 4.5) applies to extension constraint types without exception. An extension constraint type MUST NOT be registered unless its registration defines all of the following.

A subsumption verification procedure. The registration MUST provide a complete, formal definition of what it means for one instance of the constraint to be at least as restrictive as another instance of the same constraint type. This procedure MUST satisfy three properties:

1. *Decidable.* The procedure MUST terminate in finite time for all inputs. It MUST NOT require solving problems that are undecidable or computationally intractable in the general case. If the constraint language used by the type is not closed under decidable containment analysis, the registration MUST prescribe a conservative syntactic strategy and MUST formally justify that the strategy is sound (never accepts a non-subsuming pair).

2. *Sound.* The procedure MUST NOT return true unless the semantic subsumption relation holds. That is, if the procedure returns true for (C_parent, C_child), then for all argument values v: C_child.check(v) implies C_parent.check(v). The procedure MAY be conservative: it MAY return false for semantically subsuming pairs that it cannot verify, but it MUST NOT return true for non-subsuming pairs.

3. *Deterministic.* Two independent implementations of the procedure MUST produce identical results for the same inputs. The procedure MUST be specified precisely enough to ensure this. Ambiguity in the specification of the procedure is grounds for rejection of the registration.

This specification does not prescribe the internal mechanism of the subsumption verification procedure. Registrations MAY use structural comparison of token claims, formal type-checking, proof-carrying tokens, or any other mechanism that satisfies the three properties above. See Appendix C for non-normative guidance on policy languages with decidable containment algorithms.

Cross-type subsumption rules. For each core constraint type defined in Section 3.4, the registration MUST specify whether a derived token may substitute an extension type instance for a parent constraint of that core type (or vice versa). If substitution is permitted, the registration MUST state the conditions. Any (parent type, child type) pair not explicitly declared valid MUST be treated as invalid by enforcement points.

3.5.2. Enforcement Point Obligations

When an enforcement point encounters an extension constraint type during chain verification, it MUST:

1. Locate the registered subsumption verification procedure for that type. If no registration exists, the enforcement point MUST reject the chain (fail-closed).

2. Evaluate the subsumption relation at every chain link where the constraint appears, as part of the I4 check. A chain link where the derived constraint does not subsume the parent constraint MUST be rejected.

3. Evaluate the constraint’s check predicate against the presented argument value during authorization. If the predicate returns false, the invocation MUST be denied.

An enforcement point that does not implement a registered extension constraint type MUST deny authorization rather than skip the constraint. The presence of an unrecognized constraint type in a token represents a restriction the issuer intended to enforce. Silently omitting that check would violate the attenuation guarantee.

3.5.3. Example Registration: Path Containment

The following is an illustrative example of a conforming extension constraint registration. It is not defined normatively in this document.

Type name: path_containment

Additional members: root (string, required). An absolute path prefix.

check predicate: The argument, after resolving all . and .. components and removing redundant separators, must begin with root. The normalization step is part of the predicate; implementations that compare raw argument strings without normalization do not conform to this registration.

subsumes relation: subsumes(C_parent, C_child) is true if and only if C_child.root is C_parent.root or a descendant of C_parent.root under the normalized path ordering.

Cross-type subsumption: A derived exact constraint subsumes a parent path_containment constraint if and only if the exact value, after normalization, begins with the parent's root. All other cross-type pairs involving path_containment are invalid.

The following additional examples illustrate conforming extension registrations for network-oriented constraint types. Neither is defined normatively in this document.

Type name: cidr

Additional members: network (string, required). An IPv4 or IPv6 network in CIDR notation.

check predicate: The argument must be a valid IPv4 or IPv6 address string that falls within network after normalization of IPv6-mapped IPv4 addresses, octal notation, and URL-encoded hostnames. Implementations must normalize address representations before comparison to prevent encoding bypasses.

subsumes relation: subsumes(C_parent, C_child) is true if and only if C_child.network is a subnet of C_parent.network.

Cross-type subsumption: A derived exact constraint subsumes a parent cidr constraint if and only if the exact value, after normalization, is an address within the parent's network. All other cross-type pairs involving cidr are invalid.

Type name: url_safe

Additional members: allow_schemes (array of strings, optional, default ["http", "https"]); allow_domains (array of strings, optional); deny_domains (array of strings, optional); block_private (boolean, optional, default true); block_loopback (boolean, optional, default true); block_metadata (boolean, optional, default true).

check predicate: The argument must be a URL whose scheme appears in allow_schemes. If block_private, block_loopback, or block_metadata are true, the resolved host must not be a private, loopback, or cloud metadata address respectively, after normalization of all known IP encoding forms (decimal, hex, octal, IPv6-mapped, URL-encoded). If allow_domains is non-empty, the host must match at least one entry. If deny_domains is non-empty, the host must not match any entry. deny_domains takes precedence over allow_domains on overlap.

subsumes relation: subsumes(C_parent, C_child) is true if and only if C_child is at least as restrictive as C_parent on every field: allow_schemes is a subset of parent's; block_private, block_loopback, and block_metadata are each equal to or more restrictive than the parent's corresponding flag; allow_domains is a subset of parent's (or parent has none); deny_domains is a superset of parent's.

Cross-type subsumption: A derived exact constraint subsumes a parent url_safe constraint if and only if the exact value passes the parent's check predicate. All other cross-type pairs involving url_safe are invalid.

3.6. Examples

3.6.1. Root Delegation Token

```
{
  "jti": "01957a3f-4e23-7b01-a9d1-0050569c2e4f",
  "iss": "https://auth.example.com",
  "iat": 1741600000,
  "exp": 1741603600,
  "aat_type": "delegation",
  "del_depth": 0,
  "del_max_depth": 3,
  "cnf": {
    "jwk": {
      "kty": "OKP",
      "crv": "Ed25519",
      "x": "11qYAYKxCrfVS_7TyWQHOg7hcvPapiMlrwIaaPcHURo"
    }
  },
  "authorization_details": [
    {
      "type": "attenuating_agent_token",
      "tools": {
        "read_file": {
          "path": {
            "constraint_type": "pattern", "value": "/data/*"
          }
        },
        "search_index": {}
      }
    }
  ]
}
```

3.6.2. Derived Execution Token

```
{
  "jti": "01957a41-0081-7c20-bf3a-00a0c91e1234",
  "iss": "urn:ietf:params:oauth:jwk-thumbprint:sha-256:KAKnRDlMQVIKCfS5JhHlABCHjAFLdyE
VVHdpnGnLLg8",
  "iat": 1741600120,
  "exp": 1741601920,
  "aat_type": "execution",
  "del_depth": 1,
  "del_max_depth": 3,
  "par_hash": "sha256_base64url_of_parent_jws_signing_input",
  "cnf": {
    "jwk": {
      "kty": "OKP",
      "crv": "Ed25519",
      "x": "rAl9xvTDAeUADPnIWlGpFHtGg4Y8OqcQE5N4XYNdLPs"
    }
  },
  "authorization_details": [
    {
      "type": "attenuating_agent_token",
      "tools": {
        "read_file": {
          "path": {
            "constraint_type": "exact",
            "value": "/data/q3-report.pdf"
          }
        }
      }
    }
  ]
}
```


Note that the derived token:

* Carries a par_hash linking it to its parent.

* Has del_depth incremented to 1.

* Restricts read_file to a single file rather than the full /data subtree.

* Omits search_index, which the parent permitted. Tool omission is valid attenuation.

* Expires 1800s after its own issuance, versus the parent’s 3600s window.

3.7. Root Issuer Support and Root Token Issuance

3.7.1. Root Issuer Discovery

A root issuer that supports AAT issuance SHOULD advertise this capability using the following metadata parameter in its authorization server metadata document [RFC8414], if supported.

Metadata Parameter	Value
aat_issuer	Boolean. true if the AS can issue AAT root tokens.

Table 3

This document requests registration of aat_issuer in the IANA OAuth Authorization Server Metadata registry (Section 9.4).

3.7.2. Agent Token Request

An agent requesting a root AAT MUST include a cnf parameter in its token endpoint request (in the OAuth 2.0 sense, the agent acts as the client for this request). This specification treats cnf as a profiled use of the key confirmation object structure defined in [RFC7800] Section 3.1; its semantics are fully specified by that document. This document requests registration of cnf as an OAuth token endpoint request parameter (Section 9.6). The value MUST be a JSON object containing a jwk member with the agent's public key in JWK format [RFC7517]. This is the key that the root issuer will bind into the root token's cnf.jwk claim.

```
POST /token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials
&authorization_details=%5B%7B%22type%22%3A%22attenuating_agent_
  token%22%2C...%7D%5D
&cnf=%7B%22jwk%22%3A%7B%22kty%22%3A%22OKP%22%2C...%7D%7D
```

The request MUST also include authorization_details in RAR format [RFC9396] with type set to attenuating_agent_token, enumerating the tools and argument constraints the agent is requesting authority to invoke or delegate.

3.7.3. Root Token Issuance

Upon a valid request, the AS constructs and returns a root AAT. The AS:

1. Sets iss to the AS's own URI.

2. Sets jti to a unique token identifier, RECOMMENDED to be a UUIDv7 value per [RFC9562].

3. Sets iat to the current time and exp to the token's expiry time, subject to the constraints in Section 4.4.

4. Sets aat_type to "delegation" or "execution" as appropriate for the grant.

5. Sets del_depth to 0, del_max_depth to the maximum delegation depth permitted for this grant, and par_hash to absent.

6. Sets cnf.jwk to the public key submitted in the agent's cnf request parameter. The root issuer MUST validate that the submitted key is well-formed and is a public key. The root issuer SHOULD require the agent to demonstrate possession of the corresponding private key, for example via a signed proof-of-possession assertion in the token request.

7. Sets authorization_details to the capability claims granted, which MAY be a subset of what the agent requested.

8. For each URI-format tool identifier in the requested authorization_details, the root issuer SHOULD verify that the requesting agent's identity, as established by the agent's client authentication credentials, corresponds to the authority component of each claimed tool URI. If this verification fails, the root issuer MUST reject the request. The mechanism by which agent identity is mapped to tool URI authority is deployment-specific and outside the scope of this specification.

9. Signs the token with the AS's own private key.

The AS returns the token in a standard OAuth 2.0 token endpoint response ([RFC6749] Section 5.1) with the following field values:

```
{
  "access_token": "<compact-serialized AAT JWT>",
  "token_type": "aat",
  "expires_in": <seconds until exp>
}
```

The token_type value "aat" is registered in Section 9.5. Clients MUST NOT treat the returned token as a bearer token for use with arbitrary resource servers. Its only valid use is as the root of an AAT delegation chain presented to an enforcement point per Section 7.

Note: this specification defines token endpoint issuance for interoperability with existing OAuth 2.0 deployments. Unlike bearer tokens, an AAT carries its own holder key binding and is not usable as a credential for HTTP resource access. Alternative issuance profiles are outside the scope of this document.

The AS does not need to store or track derived tokens issued downstream by the initial token holder. Chain verification is performed by enforcement points using only the root token’s public key as a trust anchor.

4. Attenuation Invariants

Every derived token in a chain MUST satisfy all of the following invariants. The verification algorithm in Section 7 enforces these invariants; enforcement points MUST reject any chain that violates any invariant.

4.1. Capability Lattice Model (Non-Normative)

The attenuation invariants in this section are instances of a single abstract structure: a capability lattice. This subsection states that structure informally to give readers a mental model for interpreting the normative rules that follow.

For a token T, define its capability set C(T) as the set of (tool, args) pairs that T authorizes (that is, the pairs for which T would permit invocation). The core security property of this protocol is:

C(child) ⊆ C(parent)

Every delegation step moves down or stays at the same position in this partial order. A derived token can only authorize a subset of what its parent authorized. It cannot add tools, loosen argument constraints, or extend the chain’s authority in any dimension.

The ⊆ relation is not defined by enumerating (tool, args) pairs (argument spaces are typically infinite) but by the structural subsumption rules in Section 4.5. At the tool level, the derived token’s tool set must be a subset of the parent’s. At the argument level, when the parent’s constraint map is non-empty, the derived token must preserve the parent’s key set exactly (Section 4.5 explains why closed-world semantics require this).

When the parent's map is empty, the derived token may introduce keys, transitioning from open-world to closed-world. Each per-key constraint must be at least as restrictive. A derived constraint c_child subsumes a parent constraint c_parent (written c_child c_parent) if every argument value that satisfies c_child also satisfies c_parent.

Two boundary cases complete the structure. The empty capability set is the bottom element: a token with no tools authorized is a valid but useless terminal token. The root token's capability set is the ceiling for the entire chain: no derived token at any depth can exceed what the root authorized.

Token lifetime (I3) is a mandatory attenuation dimension orthogonal to the capability lattice. A derived token with C(child) == C(parent) is still strictly more constrained if its exp is earlier than its parent's. Time-to-live (TTL) bounds are enforced independently of capability monotonicity. Both must hold for a chain to be valid.

Invariants I1 through I6 are the normative enforcement mechanism for this property. I4 (Section 4.5) directly enforces C(child) ⊆ C(parent). The remaining invariants enforce the conditions under which that comparison is meaningful: that the chain is cryptographically linked (I1, I5), that depth and time bounds are respected (I2, I3), and that the presenter holds the key (I6).

4.2. I1: Delegation Authority

```
derived.iss == jwk_thumbprint_uri(parent.cnf.jwk)
```

where jwk_thumbprint_uri constructs the [RFC9278] URI from the key's SHA-256 thumbprint. The entity that signed the derived token MUST be the holder of the parent token. Authority flows from parent holder to derived token issuer. This invariant establishes an unambiguous audit trail: each link in the chain was signed by the party that held the preceding token.

4.3. I2: Depth Monotonicity

```
derived.del_depth == parent.del_depth + 1
derived.del_depth <= parent.del_max_depth
derived.del_depth <= derived.del_max_depth
derived.del_depth <= MAX_DELEGATION_DEPTH
derived.del_max_depth <= parent.del_max_depth
```

Delegation depth increments exactly by one at each link. The chain cannot skip depths, branch, or exceed the maximum depth established in the root token. del_max_depth is an absolute ceiling, not a remaining count. A token is terminal (its holder cannot derive further tokens) when del_depth == del_max_depth. A root token with del_max_depth: 0 is therefore immediately terminal and cannot produce any derived tokens.

The del_max_depth claim serves two related purposes. First, it is a security boundary: each delegation hop is a trust extension, delegating authority through another agent whose key, runtime, and behavior must be trusted to maintain the attenuation invariant. Unbounded chains mean unbounded trust extensions; the depth limit constrains how far authority can propagate before it must be reissued from the root. Second, it is a policy expression by the root issuer: a root delegation token with del_max_depth: 2 asserts that this grant of authority should pass through no more than two intermediate agents, regardless of what those agents might prefer. Intermediate token holders can only lower del_max_depth, never raise it (I2), so the root issuer's topology constraint is cryptographically enforced across the entire chain.

MAX_DELEGATION_DEPTH is an implementation-defined finite integer specifying the maximum permitted delegation chain depth. Implementations MUST enforce a finite maximum delegation depth to prevent resource exhaustion from pathologically deep chains. The appropriate value depends on the deployment topology; swarm architectures with deep fan-out may require significantly larger values than linear delegation chains. See Appendix B.5 for guidance.

The del_max_depth claim in any token in the chain MUST NOT exceed the implementation's MAX_DELEGATION_DEPTH.

4.3.1. Implementation Resource Limits

MAX_TOKEN_SIZE is an implementation-defined finite integer specifying the maximum encoded size of a single token in bytes. Implementations MUST enforce this limit to prevent memory exhaustion from pathologically large tokens. A value of 65536 bytes (64 KiB) is RECOMMENDED.

MAX_STACK_SIZE is an implementation-defined finite integer specifying the maximum total encoded size of a chain in bytes. Implementations MUST enforce this limit. A value of 262144 bytes (256 KiB) is RECOMMENDED.

4.4. I3: TTL Monotonicity

```
derived.exp  <= parent.exp
derived.exp  >  now
derived.exp  >  derived.iat
derived.iat  >= parent.iat
derived.iat  <= now + MAX_IAT_SKEW
derived.exp  <= derived.iat + MAX_TOKEN_LIFETIME
```

MAX_IAT_SKEW is an implementation-defined finite integer specifying the maximum number of seconds a token's iat may be in the future relative to the enforcement point's clock. Implementations MUST enforce a finite MAX_IAT_SKEW. A value of 30 seconds is RECOMMENDED.

MAX_TOKEN_LIFETIME is an implementation-defined finite integer specifying the maximum permitted duration in seconds between a token's iat and exp. Implementations MUST enforce a finite MAX_TOKEN_LIFETIME. A value of 90 days is RECOMMENDED as an upper bound; deployments SHOULD use significantly shorter lifetimes in practice (see Appendix B.8).

A derived token cannot outlive its parent. Authority cannot extend beyond the lifetime of the token that granted it. A derived token's issuance time MUST NOT precede its parent's issuance time. A token with an earlier iat indicates clock manipulation or chain forgery. Tokens with iat more than MAX_IAT_SKEW in the future relative to the enforcement point's clock MUST be rejected. A token's lifetime MUST NOT exceed MAX_TOKEN_LIFETIME.

4.5. I4: Capability Monotonicity

```
tools(derived) ⊆ tools(parent)
∀ tool ∈ tools(derived):
  constraints(derived, tool)  constraints(parent, tool)
```

A derived token MUST NOT authorize tools that the parent did not authorize. For each tool that appears in both parent and derived token:

* If the parent's constraint map for that tool is non-empty, the derived token's constraint map MUST contain exactly the same set of argument keys. Under closed-world semantics (Section 3.3), the constraint map keys define the required invocation shape: any argument not named is forbidden, and any named argument must be present. Adding a key would produce invocations that the parent's closed-world check rejects (the extra argument is unknown). Dropping a key would produce invocations that omit a parent-required argument. In both cases the derived invocation set is disjoint from the parent's, not a subset.

* If the parent's constraint map is empty (open-world), the derived token MAY introduce constraint keys, transitioning to closed-world. Any closed-world constraint set is a subset of the unrestricted open-world set.

For each argument constraint key present in both parent and derived token, the derived constraint MUST be at least as restrictive as the parent's constraint.

Constraint subsumption is defined per constraint type. The normative rules are:

* *exact:* A derived exact constraint subsumes a parent constraint of the same or different type as follows: it subsumes a parent exact if the values are identical; it subsumes a parent pattern if the exact value matches the parent pattern; it subsumes a parent range if the exact value falls within the parent range; it subsumes a parent one_of if the exact value is a member of the parent set; it subsumes a parent regex if the exact value matches the parent regex pattern; it subsumes a parent wildcard unconditionally. All other parent types are invalid cross-type targets for a derived exact constraint.

* *pattern:* This specification does not define a general containment algorithm for glob patterns; therefore it uses a conservative syntactic strategy. Enforcement points MUST apply the following rules, in order:

  1. A derived exact constraint subsumes a parent pattern if the exact value matches the parent pattern.

  2. If the parent pattern ends with a single * wildcard (a terminal wildcard), a derived pattern subsumes it if the derived pattern also ends with a single * wildcard and the derived pattern's fixed prefix (the portion before the terminal *) is equal to or longer than the parent's fixed prefix, and the derived prefix begins with the parent prefix. For example, a derived pattern /data/reports/* subsumes a parent pattern /data/* because /data/reports/ begins with /data/ and is longer.

  3. In all other cases, a derived pattern subsumes a parent pattern if and only if the pattern strings are identical.

  This strategy applies to patterns containing character classes ([abc], [!abc]) without exception: a derived pattern with a different character class than its parent MUST be treated as non-subsuming even if semantically narrower. Enforcement points MUST

NOT attempt semantic evaluation to determine pattern subsumption. Deployments requiring richer containment analysis SHOULD use a registered extension constraint type (see Appendix C).

* *range:* A derived range constraint is valid only if its bounds are at least as restrictive as the parent's (derived min >= parent min, derived max <= parent max). A missing bound on the parent is treated as unbounded; a missing bound on the derived constraint is only valid if the parent bound is also missing. A derived bound's inclusivity may only become more restrictive: a derived min_inclusive: false is valid when the parent has min_inclusive: true at the same min value (exclusive is strictly tighter), but the reverse is not. The same applies to max_inclusive.

* *one_of:* A derived one_of constraint is valid only if its value set is a subset of the parent's value set. Cross-type pairs involving a derived not_one_of against a parent one_of are invalid: a not_one_of constraint accepts values outside the parent's permitted set and cannot be verified as subsuming a one_of without domain knowledge. Enforcement points MUST reject this cross-type pair.

* *not_one_of:* A derived not_one_of constraint is valid only if its excluded set is a superset of the parent's excluded set (can only add exclusions, never remove them).

* *regex:* This specification does not standardize a regex dialect or portable containment algorithm; equality of pattern strings is the normative conservative subsumption strategy. A derived regex constraint is valid attenuation of a parent regex constraint if and only if the pattern strings are identical. A derived exact constraint subsumes a parent regex constraint if and only if the exact value matches the parent regex pattern. All other cross-type pairs involving regex are invalid. Deployments requiring richer containment analysis SHOULD use a registered extension constraint type.

* *cel:* This specification does not define a general semantic containment algorithm for CEL expressions. The normative subsumption strategy requires balanced-parentheses conjunction: a derived cel constraint subsumes a parent cel constraint if and only if the derived expression string is exactly (parent_expression) && (additional_clause). The parent expression MUST appear verbatim inside the leading parentheses. Each additional clause MUST be individually wrapped in balanced parentheses. Multiple additional clauses are permitted: (parent) && (clause1) && (clause2). No other form is considered subsuming.

This requirement is a security invariant, not a style guideline. CEL's && operator binds tighter than ||. Without balanced parentheses around the additional clause, an attacker can append || evil_predicate after the conjunction, which CEL parses as (parent && clause) || evil_predicate, a top-level disjunction that widens rather than narrows authority. Wrapping each additional clause in balanced parentheses ensures that any || inside the clause is contained within the conjunction and cannot widen authority.

Enforcement points MUST verify balanced parentheses by bracket counting, not by semantic evaluation. Enforcement points MUST NOT attempt semantic evaluation to determine subsumption.

Issuers constructing a derived cel constraint MUST copy the parent expression string verbatim inside the leading parentheses and MUST wrap each additional clause in balanced parentheses. If the issuer requires a structurally different expression that is semantically narrower, it MUST request a new root token from the root issuer rather than attempting to rewrite the parent clause.

Deployments requiring richer policy expressiveness with full subsumption support SHOULD use a registered extension type built on a language with a decidable containment algorithm (see Appendix C).

* *wildcard:* A derived wildcard is valid only if the parent is also wildcard. Any other constraint type subsumes a parent wildcard.

* *all:* A derived all constraint is valid attenuation of a parent all if the derived constraint contains all clauses present in the parent (none may be dropped) and each corresponding clause satisfies the subsumption relation. The derived constraint MAY add additional clauses at any position, which only further restrict the accepted value set. Dropping any parent clause from the derived all would expand authority and MUST be rejected.

  Clause matching for all is subsumption-keyed: for each clause C_p in the parent array, the enforcement point MUST find at least one clause C_d in the derived array with the same constraint_type such that C_d subsumes C_p per this section. Each parent clause MUST be matched to a distinct derived clause (one-to-one assignment); a single derived clause MUST NOT be used to satisfy more than one parent clause. If any parent clause cannot be matched, the check MUST fail. Unmatched additional clauses in the derived array are permitted.

The following pseudocode describes the matching algorithm. Because the one-to-one assignment requirement is order- sensitive when multiple clauses share the same constraint_type, the algorithm backtracks when a greedy match leads to a dead end. (When each constraint_type appears at most once in the parent, no backtracking occurs and the algorithm reduces to a linear scan.)

```
function check_all_subsumption(parent_clauses, derived_clauses):
  used = set()
  return match(parent_clauses, 0, derived_clauses, used)

function match(parents, idx, derived, used):
  if idx == len(parents):
    return PASS
  C_p = parents[idx]
  for i, C_d in enumerate(derived):
    if i not in used and
       C_d.constraint_type == C_p.constraint_type and
       subsumes(C_d, C_p):
      used.add(i)
      if match(parents, idx + 1, derived, used) == PASS:
        return PASS
      used.remove(i)    // backtrack
  return FAIL
```

The search space is bounded by the number of clauses sharing a constraint_type. In typical usage each type appears at most once, giving O(n) behavior. Implementations MAY employ Hopcroft-Karp or similar maximum matching algorithms for the general case.

* *any:* A derived any constraint subsumes a parent any constraint if every clause in the derived constraint is subsumed by at least one clause in the parent constraint, using the per-type subsumption rules defined in this section. Formally: for each clause_d in derived.any.constraints, there MUST exist a clause_p in parent.any.constraints such that clause_d clause_p. Removing clauses is valid (it narrows the accepted set). Adding clauses is invalid (it widens it). The derived any MUST contain at least one clause. Cross-type subsumption between clauses is permitted: for example, a derived clause of exact("pdf") is subsumed by a parent clause of pattern("*.pdf") under the cross-type rules in this section.

Example: a parent token carries any([exact("pdf"), exact("csv"), exact("xlsx")]). A derived token MAY carry any([exact("pdf"), exact("csv")]) because each derived clause is subsumed by a parent clause. A derived token MUST NOT carry any([exact("pdf"), exact("docx")]) because exact("docx") is not subsumed by any parent clause.

* *not:* This specification prescribes a conservative strategy for not → not attenuation. A derived not constraint is valid attenuation of a parent not constraint if and only if the two constraints are structurally identical, compared as JCS-canonical JSON ([RFC8785]). Although widening the inner constraint semantically narrows the negation and restricts authority, verifying this direction requires recursive subsumption in reverse, which introduces significant implementation complexity and risk of divergence. Implementations MUST NOT attempt semantic evaluation to determine not → not subsumption. Issuers that need to tighten a not constraint SHOULD re-issue from the root issuer or use a registered extension type with defined negation semantics. Cross-type pairs involving not (for example, exact → not(X) or not(X) → exact) are not valid attenuations and MUST be rejected.

  Example: a parent token carries not(one_of(["a", "b"])), meaning the argument must not be "a" or "b". A derived token MAY carry an identical not(one_of(["a", "b"])) constraint (identity is valid).

  A derived token MUST NOT carry not(one_of(["a"])). That inner set is narrower, so the negation is wider: the derived token accepts "b", which the parent denies. This is a privilege escalation and the JCS-identity check rejects it because the two constraints are not structurally identical.

  Conversely, a derived token MUST NOT carry not(one_of(["a", "b", "c"])). That inner set is wider, so the negation is narrower: the derived token is genuinely more restrictive (it additionally denies "c"). However, verifying this direction requires reversing the subsumption check on the inner constraint, which introduces implementation complexity and risk of divergence. Implementations MUST NOT attempt semantic evaluation to determine not → not subsumption. The issuer MUST re-issue from root.

* *contains:* A derived contains constraint is valid attenuation of a parent contains if the derived required set is a superset of the parent's required set. Requiring additional elements is a restriction; removing required elements would expand the set of accepted argument arrays and MUST be rejected.

* *subset:* A derived subset constraint is valid attenuation of a parent subset if the derived allowed set is a subset of the parent's allowed set. Shrinking the allowed set is a restriction; adding allowed elements would expand the set of accepted argument arrays and MUST be rejected.

Any (parent constraint type, derived constraint type) pair not explicitly permitted by the above rules, or by a registered extension constraint's cross-type subsumption declaration (Section 3.5.1), MUST be rejected.

4.6. I5: Cryptographic Linkage

```
derived.par_hash ==
  base64url-nopad(SHA-256(parent JWS Signing Input))
```

Each derived token is cryptographically bound to its parent by including the SHA-256 digest of the parent token's JWS Signing Input in the par_hash claim. The JWS Signing Input is the ASCII string BASE64URL(JWS Protected Header) || '.' || BASE64URL(JWS Payload) as defined in [RFC7515] Section 5.1.

This binding prevents chain splicing: an attacker cannot substitute a different parent token to weaken the attenuation constraints visible at a given depth.

4.7. I6: Proof of Possession

```
pop_signature verifies under derived.cnf.jwk
```

The presenter of a token chain MUST demonstrate control of the private key corresponding to the leaf token's cnf.jwk. Proof of Possession is defined in Section 5.

5. Proof of Possession

5.1. Rationale

A token without proof of possession can be replayed by any party that obtains a copy of the token. In agent systems, tokens flow through model context, tool invocation results, and inter-agent message channels, all of which are observable by other components. PoP binds a specific invocation to the private key of the leaf token's holder.

5.2. PoP Token Structure

The holder of the leaf execution token produces a PoP JWT for each tool invocation. The PoP JWT is a compact serialization signed with the holder's private key. It MUST contain the following claims.

Claim	Type	Description
jti	string	Fresh random identifier. Issuers MUST NOT generate the same jti value for two different PoP JWTs. When a UUID is used, it MUST be encoded as a lowercase hyphenated string per [RFC9562]. Whether an enforcement point can detect reuse depends on whether stateful jti tracking is deployed (see Section 8.6).
iat	NumericDate	Time of PoP creation. MUST reflect the actual time of creation. Enforcement points validate this against a clock tolerance window (see Section 5.3).
aat_id	string	The jti of the execution token being presented.
aat_tool	string	The tool identifier being invoked. MUST exactly match a key in the tools map of the leaf token's authorization_details. Tool identifiers are compared as byte strings; no Unicode normalization is applied.
hta	object	The tool arguments for this invocation. Keys are argument names; values are argument values.

Table 4

The PoP JWT payload MUST be serialized as JCS-canonical JSON ([RFC8785]) before JWS signing. This is a whole-payload requirement, not specific to the hta member. The JWS signing input is therefore BASE64URL(JWS Protected Header) || '.' || BASE64URL(JCS(PoP claims)). Whole-payload JCS canonicalization ensures a deterministic byte representation; in particular, it gives hta stable equality semantics so that argument map comparison is unambiguous across implementations and languages regardless of JSON serialization choices.

The PoP JWT MUST be signed using the private key corresponding to the leaf execution token's cnf.jwk. The enforcement point verifies the PoP JWT signature against the leaf token's cnf.jwk.

```
{
  "jti": "c980f2a1-4a37-4e88-bb3c-9defd37c1a45",
  "iat": 1741600300,
  "aat_id": "01957a41-0081-7c20-bf3a-00a0c91e1234",
  "aat_tool": "read_file",
  "hta": { "path": "/data/q3-report.pdf" }
}
```

5.3. Verification

PoP verification is only meaningful against a leaf token whose chain has been fully verified per Section 7. An enforcement point MUST complete chain verification (Section 7, steps 1-6) before evaluating the PoP JWT. A valid PoP JWT against an unverified or invalid chain MUST NOT result in authorization.

The enforcement point MUST reject a PoP JWT that:

1. Has a signature that does not verify under the leaf token's cnf.jwk.

2. References an aat_id that does not match the jti of the presented leaf token.

3. Names a tool in aat_tool that is not authorized by the leaf token.

4. Presents arguments in hta that violate constraints in the leaf token, per the verification algorithm in Section 7 (step 6b).

5. Has iat that is outside the enforcement point's accepted clock tolerance window (RECOMMENDED: ±30 seconds).

The PoP JWT iat timestamp and clock tolerance window bound the replay surface to a short interval. Implementations that wish to avoid shared state MAY use fixed-width time buckets (for example, accepting PoP JWTs whose iat falls within the current or immediately preceding 30-second bucket) to simplify enforcement point implementation.

Note: The time bucket approach is stateless but probabilistic: a PoP JWT captured early in a bucket remains usable until the end of the following bucket. This approach MUST NOT be used for tool invocations that have side effects or are not idempotent. For any tool invocation where duplicate execution causes unintended side effects, stateful jti tracking MUST be used.

Full replay prevention — guaranteeing that a given PoP JWT is accepted at most once — requires stateful tracking of presented jti values across all enforcement points in a deployment. The mechanism for that state (shared cache, database, token-binding infrastructure) is deployment-specific and outside the scope of this specification. Deployments with strong replay prevention requirements SHOULD consult the security considerations in Section 8.6.

6. Token Derivation

A holder of any AAT whose del_depth is strictly less than del_max_depth MAY derive a child token as follows.

1. Set jti to a fresh unique token identifier, RECOMMENDED to be a UUIDv7 value per [RFC9562].

2. Set iat to the current time. Set exp to any value <= parent.exp, subject to the constraints in Section 4.4. Token lifetime is a mandatory attenuation dimension. Every derived token is temporally bounded by its parent regardless of capability scope. TTL is the primary revocation mechanism in this specification; see Appendix B.8 for deployment guidance.

3. Select the aat_type of the child token. The permitted transitions are defined in Section 3.1. If the child’s aat_type differs from the parent’s, the child MUST use a different cnf.jwk than the parent (type-transition key separation).

4. Select the set of tools to authorize. This set MUST be a subset of the tools authorized by the parent token.

5. For each tool, construct a constraint map with the same argument keys as the parent's constraint map for that tool (Section 4.5). For each key, select a constraint that is at least as restrictive as the parent's, per the subsumption rules in Section 4.5. If the parent's constraint map is empty, the derived token MAY introduce constraint keys.

6. Set del_depth to parent.del_depth + 1.

7. Set del_max_depth to any integer value greater than or equal to child.del_depth and less than or equal to parent.del_max_depth. Setting del_max_depth equal to child.del_depth produces a terminal token that cannot be further delegated; higher values permit further delegation up to the parent's ceiling. Both bounds are inclusive; the upper bound enforces I2.

8. Set par_hash to base64url(SHA-256(parent JWS Signing Input)), using base64url encoding without padding ([RFC7515] Appendix C).

9. Set cnf.jwk to the intended holder's public key. The value MUST be a public key; private key material MUST NOT appear in this field.

10. Sign the token with the private key corresponding to the parent token's cnf.jwk. The iss claim MUST be set to the JWK Thumbprint URI [RFC9278] of that signing key, using the SHA-256 hash algorithm.

Derivation is performed locally by the token holder. No authorization server communication is required.

A derivation in which none of the above dimensions is strictly narrowed (the tool set is identical, all constraints are unchanged, del_max_depth is unchanged, and exp is unchanged) is technically valid by the invariants. However, it produces a child token with authority identical to its parent, while consuming one delegation depth. Such derivations serve no purpose from a least-privilege standpoint and SHOULD NOT be issued. Enforcement points MAY log same-scope derivations as anomalous.

7. Chain Verification Algorithm

The enforcement point receives a chain of tokens ordered from root to leaf and MUST execute the following algorithm. Any failure MUST result in denial.

Verification requires only the token chain and the trust anchor public key. No network calls or authorization server availability are required. Chain verification itself is fully offline. Strong replay protection for side-effecting tool invocations may additionally require stateful jti tracking as described in Section 8.6; that state is outside the inputs of this algorithm.

Inputs:

```
chain:         ordered array of signed JWTs, [root, ..., leaf]
trust_anchors: set of public keys trusted as root issuers
tool:          the tool being invoked
args:          the arguments being passed to the tool
pop_jwt:       the PoP JWT presented by the agent
```

Algorithm:

1. If chain is empty, DENY.

2. Verify chain size limits:
   a. Verify the encoded size of each token does not exceed MAX_TOKEN_SIZE. If any token exceeds this limit, DENY.
   b. Verify the total encoded size of the chain does not exceed MAX_STACK_SIZE. If the chain exceeds this limit, DENY.
   c. For each token, decode the base64url payload segment and extract only the ‘jti‘ field using minimal JSON parsing. If the payload is not valid JSON or does not contain a string-valued ‘jti‘ field, DENY. Collect all extracted ‘jti‘ values; if any value appears more than once in the chain, DENY (cycle detection). This limited extraction prior to signature verification is permitted and required for this structural check; it does not constitute the application-layer claim deserialization prohibited by the post-algorithm note. The extracted ‘jti‘ values MUST be treated as untrusted until each token’s signature is verified. Full claim parsing MUST still be deferred until after signature verification succeeds for each token.

3. Verify root token:
   a. Verify the root token’s JWS alg header is on the implementation’s permitted algorithm allowlist and is consistent with the verifying trust anchor key’s kty and crv parameters. If alg is "none", not on the allowlist, or inconsistent with the key type, DENY. [Sec 8.14]
   b. Verify the root token signature against a key in trust_anchors. After signature verification succeeds, parse the root token’s claims. All subsequent root checks (3c through 3n) operate on parsed claims.
   c. Verify root.aat_type is "delegation" or "execution".

If any other value, DENY.
d. Verify root.del_depth == 0.
e. Verify root.par_hash is absent.
f. Verify root.exp > now.
g. Verify root.iat <= now + MAX_IAT_SKEW.
h. Verify root.exp > root.iat.
i. Verify root.exp <= root.iat + MAX_TOKEN_LIFETIME.
j. Verify root.del_max_depth is a non-negative integer not exceeding MAX_DELEGATION_DEPTH. If absent or invalid, DENY.
k. Verify root.jti is present and is a non-empty string. If absent or not a string, DENY.
l. Verify root.iss is present and is a URI. If absent or not a URI-formatted string, DENY.
m. Verify root.cnf is present, contains a ‘jwk‘ member, and that the ‘jwk‘ encodes a public key (MUST NOT contain a private key parameter such as ‘d‘ for EC/OKP keys or ‘p‘, ‘q‘ for RSA keys). If absent or invalid, DENY.
n. Verify root.authorization_details is present and is a non-empty array containing at most one entry with type "attenuating_agent_token". If absent, empty, or more than one such entry is present, DENY.
Note: for single-token chains where root is also the leaf, steps 3 and 6 are the only validation applied. Steps 3k through 3n ensure that required claims are present before step 6 depends on them, closing the bypass window that exists when step 4 does not run.

4. For each adjacent pair (parent, child) in chain:
a. Verify child token’s JWS alg header is on the implementation’s permitted algorithm allowlist and is consistent with parent.cnf.jwk’s kty and crv parameters. If alg is "none", not on the allowlist, or inconsistent with the key type, DENY. [Sec 8.14]
b. Verify child signature under the key in parent.cnf.jwk. [I1]
After signature verification, verify required claims are present:
b1. Verify child.jti is present and is a non-empty string. If absent or not a string, DENY.
b2. Verify child.cnf is present, contains a ‘jwk‘ member, and that the ‘jwk‘ encodes a public key (MUST NOT contain a private key parameter such as ‘d‘ for EC/OKP keys or ‘p‘, ‘q‘ for RSA keys). If absent or invalid, DENY.
b3. Verify child.authorization_details is present and is a non-empty array. If absent or empty, DENY.
b4. Verify child.del_depth and child.del_max_depth are both present and are non-negative integers. If absent or not integers, DENY.

b5. Verify child.iss, child.iat, child.exp, child.aat_type, and child.par_hash are all present. If any is absent, DENY.

c. Verify child.iss equals jwk_thumbprint_uri(parent.cnf.jwk). [I1]

d. Verify child.aat_type is "delegation" or "execution". If any other value, DENY.

e. Verify child.del_depth == parent.del_depth + 1. [I2]

f. Verify child.del_depth <= parent.del_max_depth. [I2]

g. Verify child.del_depth <= MAX_DELEGATION_DEPTH. [I2]

h. Verify child.del_max_depth <= parent.del_max_depth.[I2]
Note: the requirement that every token's del_max_depth <= MAX_DELEGATION_DEPTH is transitively satisfied: step 3j verifies this for the root, and step 4h at each link ensures the value can only decrease. Implementations MAY add this check explicitly as defense in depth.

i. Verify child.exp <= parent.exp. [I3]

j. Verify child.exp > now. [I3]

k. Verify child.iat >= parent.iat. [I3]

l. Verify child.iat <= now + MAX_IAT_SKEW. [I3]

m. Verify child.exp > child.iat. [I3]
Note: the requirement child.exp <= child.iat + MAX_TOKEN_LIFETIME is transitively satisfied: by induction, child.exp <= root.exp (step 4i at each link), root.exp <= root.iat + MAX_TOKEN_LIFETIME (step 3i), and child.iat >= root.iat (step 4k at each link), therefore child.exp <= root.iat + MAX_TOKEN_LIFETIME <= child.iat + MAX_TOKEN_LIFETIME. Implementations MAY add this check explicitly as defense in depth.

n. Verify child.del_depth <= child.del_max_depth. [I2]

o. Verify child.authorization_details contains at most one entry with type "attenuating_agent_token". If more than one such entry is present, DENY. Note: zero entries of this type is permitted for non-leaf tokens and represents an empty capability set. Step 4q will verify this is a valid attenuation of the parent (an empty tool set is always a subset).

p. For each constraint in child.authorization_details, verify the constraint tree depth does not exceed MAX_CONSTRAINT_DEPTH. If any constraint tree exceeds this limit, DENY.

q. Verify capability monotonicity (Section 4.5): [I4]

q1. Verify every tool in child's authorization_details is also present in parent's authorization_details. If any child tool is absent from the parent, DENY.

q2. For each tool present in both parent and child: if the parent's constraint map is non-empty, verify the child's constraint map contains exactly the same set

of argument keys. If any key is added or removed, DENY.
q3. For each tool present in both parent and child: if the parent's constraint map is empty, the child's constraint map MAY contain any set of keys.
q4. For each argument key present in both parent and child constraint maps, verify the child's constraint subsumes the parent's per the per-type rules in Section 4.5. If any constraint fails subsumption, DENY.

r. Verify child.par_hash equals base64url-nopad([I5] SHA-256(parent JWS Signing Input)), where base64url-nopad denotes base64url encoding without padding per {{RFC7515}} Appendix C.
s. If child.aat_type differs from parent.aat_type, verify that the JWK Thumbprint ({{RFC7638}}) of child.cnf.jwk differs from the JWK Thumbprint of parent.cnf.jwk (type-transition key separation). Implementations MUST compare thumbprints, not raw JWK objects, to prevent bypasses through key serialization variance.

5. (Defense in depth) Verify len(chain) equals leaf.del_depth + 1. A mismatch indicates a malformed or spliced chain assembly.

6. Verify leaf token:
   a. Verify leaf.authorization_details contains exactly one entry with type "attenuating_agent_token". If zero or more than one such entry is present, DENY.
   b. If leaf.aat_type is "execution", verify tool is present in leaf.authorization_details. Then, for each argument in args: if the tool's constraint map is non-empty and the argument name is not present in the constraint map, DENY (closed-world mode). For each argument name present in the constraint map, if that argument is absent from args, DENY (constrained argument MUST be present). For each argument name present in both the constraint map and args, verify the argument value satisfies the constraint.
   c. If leaf.aat_type is "delegation", DENY (delegation tokens do not authorize direct tool invocation).

7. Verify PoP JWT:
   a. Verify pop_jwt signature under leaf.cnf.jwk. [I6]
   b. Verify pop_jwt.aat_id == leaf.jti.
   c. Verify pop_jwt.aat_tool == tool.
   d. Verify pop_jwt.hta, when JCS-canonicalized ({{RFC8785}}), equals the JCS-canonical form of the args map for this invocation. If the canonical byte

sequences differ, DENY.

e. Verify pop_jwt.iat is within the clock tolerance window. If outside the window, DENY.

8. PERMIT.

Enforcement points MUST verify the JWS signature of each token before deserializing its payload claims into application-layer data structures. Signature verification operates on the raw encoded header and payload bytes (the JWS Signing Input) and does not require claim parsing. Full claim parsing MUST NOT occur until after signature verification succeeds for that token. This ordering prevents parser-based denial-of-service attacks on maliciously crafted payloads. The sole exception is step 2c: extracting only the jti string field for cycle detection prior to signature verification is permitted, provided the implementation treats the extracted value as untrusted until the corresponding signature is verified. Enforcement points MUST reject any token whose JWS alg header is "none". The "none" algorithm provides no cryptographic protection and MUST NOT be used in any AAT or PoP JWT.

The hta comparison in step 7d requires both the enforcement point and the PoP JWT issuer to use JCS canonicalization ([RFC8785]). The enforcement point MUST canonicalize the args map independently and compare the resulting byte sequence against the canonical form committed to by the PoP JWT signature. Implementations MUST NOT compare raw JSON strings; surface differences such as key ordering or numeric representation (e.g., 1.0 vs 1) are resolved by canonicalization before comparison.

The JWS alg header value MUST be consistent with the key type of the key used to verify the signature: the trust anchor public key for root tokens, and the cnf.jwk of the parent token for derived tokens. Enforcement points MUST reject any token where the declared alg is not compatible with the verifying key’s kty and crv parameters. For example, a token whose alg is "EdDSA" MUST be verified against an OKP key with "crv": "Ed25519" or "crv": "Ed448". A mismatch between the declared algorithm and the verifying key type MUST result in denial, regardless of whether the signature bytes would verify under an alternate interpretation.

8. Security Considerations

8.1. Threat Model

This section characterizes the threats that AATs mitigate and the threats that are outside the scope of this mechanism. Implementations SHOULD use this characterization to evaluate whether AATs are sufficient for their threat environment and to identify what complementary controls are required.

8.1.1. Threats Mitigated

Prompt injection leading to unauthorized tool invocation. An attacker who injects instructions into an agent's input cannot cause the agent to invoke tools outside the scope encoded in its token. The enforcement point rejects any invocation of an unauthorized tool regardless of the agent's stated rationale.

Hallucinated tool invocations with out-of-scope arguments. Even when an agent invokes an authorized tool, argument constraints in the leaf token restrict the argument values the enforcement point will accept. An agent that hallucinates an argument value outside the authorized range is denied at the enforcement point before the tool executes.

Confused deputy attacks. A sub-agent that receives a derived token cannot exercise authority beyond what its token encodes, even if it is invoked by a trusted orchestrator. The token carries its own authorization ceiling. There is no ambient authority to confuse.

Privilege escalation across delegation hops. The capability monotonicity invariant (I4) ensures that authority can only narrow at each delegation step. A derived token cannot authorize tools or argument values absent from its parent token. An agent that attempts to mint a derived token with broader scope will produce a token that fails chain verification at the enforcement point.

Compromised sub-agents. If a sub-agent is compromised, the blast radius is bounded by the scope of the token it holds. The attacker cannot use the compromised agent to escalate to broader authority, invoke tools outside the token's scope, or derive tokens with wider permissions than the compromised token encodes.

Chain splicing. The par_hash claim (I5) binds each derived token to the specific bytes of its parent token. An attacker cannot substitute a different parent token into the chain to change the authority ceiling without invalidating the par_hash verification at the enforcement point.

Token replay for irreversible operations. For irreversible or side-effecting tool invocations, stateful jti tracking at the enforcement point enables prevention of PoP JWT replay. See Section 8.6 for the distinction between stateful and probabilistic replay controls and the deployment requirements for each.

8.1.2. Threats Not Mitigated

Malicious or compromised root issuer. The security of all chains depends on the integrity of the trust anchor key. A root issuer that mints tokens with overly broad scopes, or whose signing key is compromised, undermines the authorization guarantees of every chain it anchors. AATs provide no mechanism to detect or constrain a malicious root issuer. Key management, rotation procedures, and root issuer accountability are deployment concerns outside the scope of this specification.

Compromised enforcement point. An enforcement point that skips chain verification, ignores constraint evaluation, or accepts forged tokens provides no security guarantee regardless of the token format. AATs assume enforcement points are honest and implement the verification algorithm in Section 7 correctly. Enforcement point integrity is a deployment concern.

Actions within authorized argument constraints. AATs restrict which tools an agent may invoke and what argument values are permitted. They do not restrict which authorized invocations an agent chooses to make, in what order, or how many times. An agent that makes excessive or unintended use of its authorized tools within the bounds of its token is not detectable at the enforcement point. Rate limiting, audit logging, and behavioral monitoring are complementary controls for this threat.

Compromised holder key. If an agent’s private key is stolen, the attacker can exercise the full authority encoded in that agent’s token until the token expires. The blast radius is bounded by the token scope, but within that scope the attacker has full authorization. Short token lifetimes (Section 8.4) limit the exposure window.

Model exfiltration and side-channel attacks. An attacker who extracts an agent’s model weights, system prompt, or in-context state may be able to predict or manipulate the agent’s behavior independently of its token constraints. AATs operate at the authorization layer and have no visibility into the model layer.

Social engineering of the root issuer. An attacker who convinces the root issuer to mint a root delegation token with broad scope obtains broad authority through legitimate token issuance. This is not detectable by chain verification.

8.2. Attenuation as the Security Invariant

The security guarantee of this specification rests entirely on the enforcement of the capability monotonicity invariant (I4). An enforcement point that fails to check I4, or that checks it incorrectly, provides no blast radius containment. Implementers MUST test I4 enforcement against the full constraint attenuation matrix in Section 4.5, including all (parent type, child type) pairs, and MUST reject all pairs not explicitly permitted.

The other invariants (I1, I2, I3, I5, I6) rely on well-established cryptographic primitives and validation patterns with substantial prior art in deployed systems. I4 is novel. Formal verification of the I4 subsumption rules is in progress, using bounded model checking ([ALLOY]) for set-theoretic constraint types and SMT solving ([Z3]) for domain-specific types requiring arithmetic or string reasoning. For the remaining constraint types, which use conservative syntactic strategies (Section 4.5), model-level verification is ongoing. Implementers are encouraged to treat the subsumption logic for those types with corresponding caution and to publish independent analyses.

The reference implementation includes a test suite covering monotonicity of the attenuation invariants under arbitrary sequences, normalization idempotence across encode/decode round-trips, and enforcement agreement between in-memory and deserialized constraint evaluation. See Appendix E for implementation status.

8.3. Root Key Compromise

A compromised trust anchor key allows an attacker to issue arbitrary root tokens. This breaks the security guarantees of all chains anchored to that key. Deployments SHOULD implement key rotation procedures and revocation mechanisms appropriate to their risk model. The specific mechanism for root key revocation, including revocation list formats, distribution protocols, and enforcement point update procedures, is outside the scope of this specification.

8.4. Holder Key Compromise

A compromised holder key allows an attacker to present existing tokens issued to that holder. The attacker cannot derive tokens with broader scope than the compromised token grants. Mitigation is revocation of tokens bound to the compromised key, or expiry-based recovery for short-lived tokens.

8.5. Chain Splicing

The par_hash invariant (I5) is the primary defense against chain splicing, as described in Section 8.1.1. Enforcement points MUST verify par_hash at every chain link per step 4r of the verification algorithm (Section 7).

8.6. Replay Attacks

The PoP JWT binds a specific invocation to a nonce, a timestamp, the target tool, and the presented arguments. The timestamp window limits the interval during which a captured PoP JWT remains usable to approximately twice the clock tolerance (RECOMMENDED: ±30 seconds, giving a window of roughly 60 seconds). This provides probabilistic replay resistance and is appropriate only for idempotent, read-only tool invocations where duplicate execution is harmless.

For tool invocations that are irreversible or have significant side effects, including financial transactions, data deletion, writes to external systems, and any operation that cannot be undone: enforcement points MUST implement stateful jti tracking for PoP JWTs and MUST NOT rely solely on the timestamp window for replay protection.

This specification requires stateful jti tracking for irreversible operations but does not define the storage backend, consistency model, or distribution protocol for that state. The required consistency properties depend on the deployment topology and the risk tolerance of the application. Deployments SHOULD treat the time-windowed PoP as a probabilistic control and layer additional idempotency mechanisms at the application level for high-value operations.

8.7. Constraint Evaluation

Several constraint types introduce potential for denial-of-service through expensive evaluation. regex patterns can cause catastrophic backtracking. cel expressions execute arbitrary logic at argument check time. Enforcement points SHOULD impose evaluation timeouts on any constraint type whose check predicate is not O(n) in the length of the argument value.

The subsumption check for cel constraints (the balanced-parentheses bracket counting described in Section 4.5) is O(n) in expression length and does not evaluate the expression. Only the runtime check predicate, which evaluates the CEL expression against actual argument values at invocation time, carries unbounded cost. These are distinct operations; implementations MUST NOT conflate them.

Extension constraint types registered under Section 3.5 MUST document their computational complexity and any resource limits implementations SHOULD enforce.

8.8. Depth Limit

Enforcement points MUST enforce a finite MAX_DELEGATION_DEPTH to prevent resource exhaustion from artificially deep chains. The appropriate value is deployment-specific: linear orchestration chains require far fewer hops than swarm architectures with deep fan-out delegation. Implementations SHOULD choose a value that reflects the maximum chain depth their deployment topology requires, without imposing an artificial ceiling on legitimate use cases. See Appendix B.5 for guidance on selecting an appropriate value.

The security rationale for depth limiting goes beyond resource exhaustion. Each delegation hop introduces an additional agent into the trust chain: the enforcement point necessarily trusts not only that the leaf token holder is honest, but that every intermediate holder made sound attenuation decisions. A compromised or misdirected intermediate agent can narrow constraints in ways that serve an attacker’s goals while remaining within the invariants. The depth limit bounds the number of such trust extensions that a single root grant can produce.

The del_max_depth claim in the root token is the root issuer’s explicit policy on chain topology. An implementation that ignores del_max_depth or enforces only a global implementation limit without checking per-token values violates this policy. Enforcement points MUST check both the per-token del_max_depth value (step 4f of Section 7) and the global MAX_DELEGATION_DEPTH (step 4g of Section 7). Neither check alone is sufficient.

8.9. Unknown Constraint Types

Enforcement points MUST deny authorization when they encounter an unknown constraint type. Permitting invocation in the presence of an unrecognized constraint would silently remove a restriction the issuer intended to enforce.

8.10. Token Revocation

Revocation of individual AATs, including derived tokens, is outside the scope of this specification. The offline delegation model trades per-token revocation granularity for verifiability without authorization server availability. This is a deliberate design choice, not an oversight.

Deployments SHOULD use short token lifetimes as the primary recovery mechanism. A short-lived execution token provides a bounded damage window that is operationally equivalent to revocation for most threat models, without the availability and consistency requirements that a revocation list imposes. Root tokens SHOULD be issued with the shortest lifetime compatible with the intended delegation chain depth.

Root trust anchor rotation (replacing the trust anchor signing key and re-issuing root tokens) is the appropriate response to a root key compromise. Enforcement points SHOULD support configurable trust anchor sets to enable rotation without downtime.

Revocation list distribution, token status list integration, and per-token introspection mechanisms are deferred to a companion document.

8.11. Clock Skew

This specification uses clock-based checks in two distinct contexts with different semantics. MAX_IAT_SKEW (Section 4.4, RECOMMENDED: 30 seconds) is a one-sided future-dating tolerance applied to token iat values: it prevents a token issued slightly in the future from being rejected due to minor clock drift between issuer and enforcement point. The PoP JWT timestamp window (Section 5.3, RECOMMENDED: ±30 seconds) is a bilateral replay window applied to PoP JWT iat values: it bounds how long a captured PoP JWT remains usable. These are independent parameters enforced at different points in the verification algorithm and SHOULD be configured separately.

PoP JWT timestamp verification requires synchronized clocks. The RECOMMENDED tolerance window is ±30 seconds, which accommodates typical Network Time Protocol (NTP) synchronized deployments with generous margin. Deployments running on cloud infrastructure with

guaranteed NTP synchronization SHOULD target ±5 to ±10 seconds. Deployments with stricter security requirements MAY reduce this window further.

Implementations MUST enforce a finite maximum tolerance window. Values beyond ±60 seconds provide negligible additional clock skew tolerance while meaningfully expanding the PoP replay window and are NOT RECOMMENDED. A value of ±30 seconds is the conservative baseline; the ±60 second ceiling is intended only for heterogeneous environments such as embedded systems or degraded connectivity scenarios.

8.12. Type-Transition Key Separation

The requirement that any type transition use a distinct key (Section 3.1) is not an architectural preference; it closes specific escalation paths. In traditional capability systems, same-key self-issuance across authority types is often benign because the principal model is static and the runtime is trusted. Agentic systems have neither property: agents operate autonomously across untrusted environments, may be subject to prompt injection or context manipulation, and act on behalf of humans without per-action confirmation.

If a single key can move between planning authority and tool invocation authority, the delegation/execution boundary collapses under any compromise. The restriction ensures that a compromised orchestrator holding a delegation token cannot directly invoke tools without involving a distinct execution-bound key, and that a compromised executor cannot create new delegation branches under the same identity. This preserves the accountability boundary between authority types and ensures that key separation is enforced structurally rather than by policy.

8.13. CEL Conjunction Privilege Escalation

The cel constraint subsumption strategy (Section 4.5) requires that additional clauses in a derived expression be individually wrapped in balanced parentheses: (parent) && (clause). This requirement is a security invariant.

Without balanced parentheses, an attacker constructing a derived cel constraint can append a top-level disjunction to widen authority. For example, given a parent expression `amount < 10000`, the derived expression `(amount < 10000) && true || amount < 1000000` passes a naive prefix check but CEL parses it as `((amount < 10000) && true) || (amount < 1000000)`, which simplifies to `amount < 1000000`, a 100x privilege escalation.

Enforcement points MUST implement the subsumption check using balanced-parentheses bracket counting as specified in Section 4.5. Implementations MUST reject any derived cel expression where the additional clause is not wrapped in balanced parentheses, regardless of whether the expression appears semantically narrower. Implementations MUST NOT attempt semantic evaluation to determine subsumption.

8.14. Algorithm Confusion

AATs are signed JWTs. Implementations are subject to the full class of JWT algorithm confusion attacks, including alg: "none" acceptance, symmetric/asymmetric key confusion (RS256/HS256 key reuse), and algorithm substitution across tokens in the same chain.

Enforcement points MUST maintain an explicit allowlist of permitted signature algorithms and MUST reject any token whose alg header value is not on that list. Implementations MUST reject tokens with alg: "none" unconditionally and MUST NOT treat the absence of an alg header as equivalent to any permitted algorithm.

In chains where multiple token types are verified, implementations MUST apply the algorithm allowlist independently to each token. Accepting a weaker algorithm on an intermediate token because the leaf token used a strong algorithm is a verification failure.

The RECOMMENDED algorithm set is the same as for DPoP [RFC9449]: ES256, ES384, ES512, RS256, RS384, RS512, PS256, PS384, PS512, EdDSA. Symmetric algorithms (HS256, HS384, HS512) MUST NOT be used for AAT signatures; symmetric keys cannot provide the per-holder key binding that PoP requires.

8.15. Token Content Visibility

AAT payloads are integrity-protected but not encrypted. Token contents, including tool identifiers, argument constraints, and delegation chain structure, are visible to any party that observes the token in transit or at rest. Deployments SHOULD transmit AAT chains over encrypted transport (e.g., TLS) and SHOULD treat stored tokens as sensitive material. Token encryption is outside the scope of this specification.

9. IANA Considerations

9.1. JWT Claims Registry

This document requests registration of the following claims in the IANA JSON Web Token Claims Registry [RFC7519].

AAT claims:

Claim Name	Claim Description	Change Controller	Reference
aat_type	Attenuating Authorization Token type	IETF	This document
del_depth	Delegation chain depth	IETF	This document
del_max_depth	Maximum delegation chain depth	IETF	This document
par_hash	Parent token JWS Signing Input hash	IETF	This document

Table 5

The tools map is not a top-level JWT claim; it is a member nested inside the authorization_details array entry with type: "attenuating_agent_token", as defined in Section 3.3. Its structure and semantics are governed by the AAT Constraint Type Registry (Section 9.3) and the RAR profile defined in this document, not by the JWT Claims Registry.

PoP JWT claims:

Claim Name	Claim Description	Change Controller	Reference
aat_id	AAT jti being presented	IETF	This document
aat_tool	Tool identifier for PoP binding	IETF	This document
hta	Tool arguments for PoP binding	IETF	This document

Table 6

9.2. OAuth Authorization Details Types Registry

This document requests registration of the following type in the IANA OAuth Authorization Details Types Registry established by [RFC9396].

Type Name	Reference
attenuating_agent_token	This document

Table 7

9.3. AAT Constraint Type Registry

This document requests IANA create the "Attenuating Authorization Token Constraint Types" registry. The registration policy for this registry is Specification Required [RFC8126].

9.3.1. Designated Expert Instructions

Designated experts MUST verify that each submitted registration satisfies all of the following criteria before approving it:

1. The type name is a lowercase string containing only letters, digits, and underscores, and does not conflict with an existing registered type name.

2. The check predicate is fully specified: given any argument value, an independent implementer can determine without ambiguity whether the predicate returns true or false.

3. The subsumes verification procedure satisfies the decidable, sound, and deterministic properties defined in Section 3.5.1. If the constraint language does not support a general containment algorithm, the registration prescribes a conservative syntactic strategy and formally justifies its soundness.

4. The cross-type subsumption rules enumerate every (parent type, child type) pair involving both the new type and all existing core types that the registration declares valid, with explicit conditions. Unlisted pairs are implicitly invalid; the registration MUST NOT rely on the catch-all rejection rule to handle pairs that deserve explicit treatment.

5. The reference is a stable, publicly accessible document. Internet-Drafts that have not yet been published as RFCs are not acceptable as stable references.

Designated experts SHOULD request clarification when cross-type rules are incomplete, when the subsumption procedure's soundness is not formally justified, or when the check predicate leaves ambiguous cases unresolved.

9.3.2. Registration Template

Registration requests MUST use the following template:

```
   Type name:
     (A lowercase string. Example: "path_containment")

   Additional members:
     (List each JSON member name, its JSON type, whether it is required
     or optional, its default value if optional, and its semantics.
     Example: "root (string, required): An absolute path prefix.")

   check predicate:
     (A complete, unambiguous specification of the boolean predicate
     evaluated against an argument value at invocation time. Must
     cover all edge cases including null, empty, and out-of-range
     inputs.)

   subsumes verification procedure:
     (A complete formal definition of what it means for one instance
     of this constraint type to be at least as restrictive as another.
     Must state whether the procedure is conservative and, if so, which
     semantically subsuming pairs it rejects. Must include a soundness
     argument: if the procedure returns true for (C_parent, C_child),
     then for all values v: C_child.check(v) implies C_parent.check(v).)

   cross-type subsumption rules:
     (An explicit enumeration of every (parent type, child type) pair
     involving this type that is a valid attenuation, and the conditions
     under which it is valid. List both directions: this type as parent
     and this type as child. All unlisted pairs are implicitly invalid.
     Example:
       - (exact, this_type): valid if the exact value satisfies this
         type's check predicate.
       - (this_type, exact): invalid.
       - (this_type, this_type): valid if [condition].)

   security considerations:
     (Any security properties, limitations, or attack surfaces specific
     to this constraint type, including known cases where the check
     predicate or subsumption procedure can be bypassed or confused.)

   reference:
     (A stable, publicly accessible document defining all of the above.)
```

9.3.3. Initial Registry Entries

The core constraint types defined in Section 3.4 of this document constitute the initial registry entries. For each type, the check predicate and additional members are defined in Section 3.4, and the subsumption rules and cross-type pairs are defined in Section 4.5.

Type Name	Reference
exact	This document (Sections 3.4, 4.5)
pattern	This document (Sections 3.4, 4.5)
range	This document (Sections 3.4, 4.5)
one_of	This document (Sections 3.4, 4.5)
not_one_of	This document (Sections 3.4, 4.5)
contains	This document (Sections 3.4, 4.5)
subset	This document (Sections 3.4, 4.5)
regex	This document (Sections 3.4, 4.5)
cel	This document (Sections 3.4, 4.5)
wildcard	This document (Sections 3.4, 4.5)
all	This document (Sections 3.4, 4.5)
any	This document (Sections 3.4, 4.5)
not	This document (Sections 3.4, 4.5)

Table 8

9.4. OAuth Authorization Server Metadata Registry

This document requests registration of the following parameter in the IANA OAuth Authorization Server Metadata registry established by [RFC8414].

Metadata Parameter	Metadata Description	Change Controller	Reference
aat_issuer	Indicates root AAT issuance support	IETF	This document

Table 9

aat_issuer is a boolean value. When present and true, it indicates that the root issuer supports issuance of AAT root tokens as described in Section 3.7. When absent, the AS is assumed not to support AAT issuance.

9.5. OAuth Token Type Registration

This document requests registration of the following token type in the OAuth Token Type Registry ([RFC6749] Section 11.1):

* Type name: aat
* Additional Token Endpoint Response Parameters: (none)
* HTTP Authentication Scheme(s): (none; not a bearer token)
* Change controller: IETF
* Specification document(s): This document

9.6. OAuth Token Endpoint Parameters Registry

This document requests registration of the following parameter in the OAuth Parameters Registry established by [RFC6749] Section 11.2:

* Parameter name: cnf
* Parameter usage location: token request
* Change controller: IETF
* Specification document(s): This document, [RFC7800] Section 3.1

The cnf parameter carries a key confirmation object as defined in [RFC7800]. When used at the token endpoint, its value MUST be a JSON object containing a jwk member with the requesting agent’s public key. The authorization server binds this key into the issued AAT’s cnf.jwk claim. The semantics of the cnf object are defined by [RFC7800]; this registration documents its use at the OAuth token endpoint for AAT issuance.

10. References

10.1. Normative References

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <https://www.rfc-editor.org/info/rfc7515>.

[RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <https://www.rfc-editor.org/info/rfc7517>.

[RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <https://www.rfc-editor.org/info/rfc7519>.

[RFC7638] Jones, M. and N. Sakimura, "JSON Web Key (JWK) Thumbprint", RFC 7638, DOI 10.17487/RFC7638, September 2015, <https://www.rfc-editor.org/info/rfc7638>.

[RFC7800] Jones, M., Bradley, J., and H. Tschofenig, "Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)", RFC 7800, DOI 10.17487/RFC7800, April 2016, <https://www.rfc-editor.org/info/rfc7800>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

[RFC8785] Rundgren, A., Jordan, B., and S. Erdtman, "JSON Canonicalization Scheme (JCS)", RFC 8785, DOI 10.17487/RFC8785, June 2020, <https://www.rfc-editor.org/info/rfc8785>.

[RFC9278] Jones, M. and K. Yasuda, "JWK Thumbprint URI", RFC 9278, DOI 10.17487/RFC9278, August 2022, <https://www.rfc-editor.org/info/rfc9278>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/info/rfc8949>.

[RFC9396] Lodderstedt, T., Richer, J., and B. Campbell, "OAuth 2.0 Rich Authorization Requests", RFC 9396, DOI 10.17487/RFC9396, May 2023, <https://www.rfc-editor.org/info/rfc9396>.

[RFC9562] Davis, K., Peabody, B., and P. Leach, "Universally Unique IDentifiers (UUIDs)", RFC 9562, DOI 10.17487/RFC9562, May 2024, <https://www.rfc-editor.org/info/rfc9562>.

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <https://www.rfc-editor.org/info/rfc6749>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

10.2. Informative References

[RFC8414] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", RFC 8414, DOI 10.17487/RFC8414, June 2018, <https://www.rfc-editor.org/info/rfc8414>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8392] Jones, M., Wahlstroem, E., Erdtman, S., and H. Tschofenig, "CBOR Web Token (CWT)", RFC 8392, DOI 10.17487/RFC8392, May 2018, <https://www.rfc-editor.org/info/rfc8392>.

[RFC8693] Jones, M., Nadalin, A., Campbell, B., Ed., Bradley, J., and C. Mortimore, "OAuth 2.0 Token Exchange", RFC 8693, DOI 10.17487/RFC8693, January 2020, <https://www.rfc-editor.org/info/rfc8693>.

[RFC9052] Schaad, J., "CBOR Object Signing and Encryption (COSE): Structures and Process", STD 96, RFC 9052, DOI 10.17487/RFC9052, August 2022, <https://www.rfc-editor.org/info/rfc9052>.

[RFC9449] Fett, D., Campbell, B., Bradley, J., Lodderstedt, T., Jones, M., and D. Waite, "OAuth 2.0 Demonstrating Proof of Possession (DPoP)", RFC 9449, DOI 10.17487/RFC9449, September 2023, <https://www.rfc-editor.org/info/rfc9449>.

[BISCUIT] Eclipse Foundation, "Biscuit: Distributed Authorization Tokens", n.d., <https://doc.biscuitsec.org/reference/specifications.html>.

[MACAROONS] Birgisson, A., Politz, J. G., Erlingsson, U., Taly, A., Vrable, M., and M. Lentczner, "Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud", NDSS 2014, 2014, <https://research.google/pubs/pub41892/>.

[SALTZER75] Saltzer, J. H. and M. D. Schroeder, "The Protection of Information in Computer Systems", Proceedings of the IEEE Vol. 63, No. 9, 1975, <https://doi.org/10.1109/PROC.1975.9939>.

[HARDY88] Hardy, N., "The Confused Deputy (or why capabilities might have been invented)", ACM SIGOPS Operating Systems Review Vol. 22, No. 4, 1988, <https://dl.acm.org/doi/10.1145/54289.871709>.

[CAMEL25] Debenedetti, E., Shumailov, I., Fan, T., Hayes, J., Carlini, N., Fabian, D., Kern, C., Shi, C., Terzis, A., and F. Tramr, "Defeating Prompt Injections by Design", 2025, <https://arxiv.org/abs/2503.18813>.

[DEEPMIND26] Tomaev, N., Franklin, M., and S. Osindero, "Intelligent AI Delegation", 2026, <https://arxiv.org/abs/2602.11865>.

[WIMSE-ARCH] Salowey, J., Rosomakho, Y., and H. Tschofenig, "Workload Identity in a Multi System Environment (WIMSE) Architecture", March 2026, <https://datatracker.ietf.org/doc/draft-ietf-wimse-arch/>.

[WIMSE-S2S] Campbell, B., Salowey, J. A., Schwenkschuster, A., and Y. Sheffer, "WIMSE Workload-to-Workload Authentication", October 2025, <https://datatracker.ietf.org/doc/draft-ietf-wimse-s2s-protocol/>.

[DENNIS66] Dennis, J. B. and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations", Communications of the ACM Vol. 9, No. 3, 1966, <https://doi.org/10.1145/365230.365252>.

[MILLER06] Miller, M. S., "Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control", PhD Dissertation Johns Hopkins University, 2006, <http://www.erights.org/talks/thesis/>.

[ALLOY] Jackson, D., "Alloy: A Lightweight Object Modelling Notation", ACM Transactions on Software Engineering and Methodology Vol. 11, No. 2, 2002, <https://doi.org/10.1145/505145.505149>.

[Z3] de Moura, L. and N. Bjrner, "Z3: An Efficient SMT Solver", TACAS 2008, LNCS 4963, 2008, <https://github.com/Z3Prover/z3>.

Appendix A. Comparison with Related OAuth Mechanisms (Non-Normative)

A.1. Token Exchange (RFC 8693)

RFC 8693 allows a client to exchange one token for another, potentially with reduced scope, by contacting the authorization server. The server enforces scope reduction. This requires network connectivity to the authorization server at each delegation hop and cannot be performed offline.

This specification allows a token holder to derive a new token locally, without contacting the authorization server. The attenuation invariant is enforced by the chain verification algorithm, not by a server-side policy check.

A.2. Rich Authorization Requests (RFC 9396)

RFC 9396 defines a structured format for expressing fine-grained authorization details in OAuth tokens. This specification uses the authorization_details claim format from RFC 9396 and extends it with: (1) a delegation chain model that links tokens via cryptographic hashes, (2) monotonic attenuation invariants that constrain what derived tokens may express, and (3) proof-of-possession binding that ties invocations to specific key holders.

A.3. DPoP (RFC 9449)

DPoP ([RFC9449]) is a token theft prevention mechanism that binds an existing OAuth access token to a holder key, ensuring that a stolen token cannot be presented without the corresponding private key. DPoP does not change what the access token authorizes; the token's authorization claims are unchanged. The resource server grants whatever the access token permits; DPoP adds a cryptographic proof that the presenter holds the bound key.

AATs encode the authorization itself. The token specifies which tools may be invoked, with what argument constraints, and by which key holder. Holders can derive tokens with authority equal to or narrower than their own, without contacting the authorization server. The PoP JWT in Section 5 serves a similar cryptographic role to a DPoP proof, binding a specific invocation to the leaf token's holder key, but operates in a different context. Everything else in this specification (the chain model, the attenuation invariants, the constraint type registry, the subsumption matrix) addresses questions outside DPoP's scope.

Structurally, DPoP is a two-party protocol between a client and a resource server. There is no delegation model, no parent-child chain, and no attenuation invariant. The chain model of this specification (del_depth, par_hash, del_max_depth, and the six attenuation invariants) has no DPoP analog.

At the proof level, DPoP binds to an HTTP method (htm) and URI (htu). AAT PoP JWTs bind to a tool name (aat_tool) and a structured argument map (hta). Tool invocations are function calls, not HTTP requests, and a URI alone carries insufficient information for argument-level constraint evaluation. This is why aat_tool and hta differ structurally from htm and htu: (1) hta carries the full argument map required for constraint evaluation at the enforcement point; (2) aat_id binds the proof to a specific leaf token jti and chain position, for which DPoP has no equivalent.

The cryptographic mechanism is the same: an asymmetric key in cnf.jwk, compact JWT serialization, verified against the leaf token's bound key. DPoP could in principle be layered alongside AATs as a transport-level binding for chain delivery, but that combination is outside the scope of this specification.

A.4. Biscuit

Biscuit [BISCUIT] is a capability-based authorization token format that combines asymmetric public key signatures with offline attenuation, building on the Macaroons model. Like AATs, Biscuit tokens support offline derivation and monotonic attenuation: a holder can produce a more restricted token without contacting the original issuer, and the resulting token cannot exceed the authority of its parent.

The primary structural difference is the policy language. Biscuit encodes authorization logic in a Datalog variant that is evaluated at verification time, requiring a logic engine at the enforcement point. This enables expressive relational policies but introduces a runtime dependency and non-trivial computational bounds to manage.

AATs encode authorization as a structured capability map with typed argument constraints. The core constraint types are decidable by structural analysis, requiring no logic engine. For cases where structural constraints are insufficient, CEL expressions provide a bounded escape hatch, and a registered extension type mechanism supports domain-specific constraint types. This tradeoff favors simplicity and predictability at the enforcement point, at the cost of the relational expressiveness Datalog provides.

A second difference is scope. Biscuit is a general-purpose authorization token format. It does not natively encode token roles such as delegation versus execution authority, depth-limit claims, or explicit chain position declarations. This specification defines a typed delegation-chain structure with those properties in the token model itself, making the chain independently verifiable as a delegation protocol rather than as a sequence of policy blocks.

Appendix B. Implementation Notes (Non-Normative)

B.1. Algorithm Recommendations

* *Signing algorithm:* Ed25519 [RFC8032]. The normative requirement is in Section 3.2. EdDSA provides compact 64-byte signatures suitable for constrained agent environments. The JWS alg header value for Ed25519 is "EdDSA".

* *Key representation:* JWK [RFC7517] with "kty": "OKP" and "crv": "Ed25519".

* *Token identifier:* UUIDv7 is recommended for jti values, providing time-ordered identifiers without central coordination.

The algorithm allowlist requirement is normatively defined in Section 7 (steps 3a and 4a) and discussed in Section 8.14.

Post-quantum migration: the cnf.jwk key type is not hardcoded to Ed25519. Implementations should be designed to support key type migration. NIST finalized ML-DSA (FIPS 204, formerly Dilithium) in 2024 as a post-quantum digital signature standard. Deployments with long-term security requirements should design their key management infrastructure to support algorithm migration without requiring changes to token structure.

B.2. Performance

Chain verification requires one signature verification per chain link. Ed25519 verification is computationally lightweight; for typical chain depths of 3 to 5 links, verification overhead is negligible relative to network latency in most deployment contexts. Constraint evaluation for exact, one_of, range, and similar structural types is O(n) in the number of constraints. For regex and cel, evaluation cost depends on expression complexity and input size; see Section 8.7 for resource limit guidance.

B.3. Recognizing Derived Token iss Values in Middleware

In both root and derived AATs, iss is a URI. For root tokens it is a conventional issuer URI. For derived tokens it is a JWK Thumbprint URI ([RFC9278]) with the urn:ietf:params:oauth:jwk-thumbprint:sha-256: prefix. Middleware that routes or policy-evaluates based on iss should recognize the JWK Thumbprint URI scheme and apply chain-aware processing rather than attempting to resolve the URI as an issuer endpoint. The verification key for derived tokens is parent.cnf.jwk, resolved from the preceding chain link.

B.4. Relationship to WIMSE

The WIMSE architecture [WIMSE-ARCH] and service-to-service protocol [WIMSE-S2S] address workload identity and authentication for entities that hold and present AATs. A WIMSE workload credential identifies an agent; the iss claim in a root AAT issued to that agent may reference the agent’s WIMSE workload identifier. The two specifications are complementary: WIMSE establishes workload identity and authentication; this specification defines a holder-derivable, invocation-scoped delegation and attenuation mechanism that WIMSE does not standardize.

B.5. Delegation Depth Guidance

The normative requirement is only that implementations enforce a finite MAX_DELEGATION_DEPTH. This appendix provides non-normative guidance for selecting an appropriate value.

The appropriate MAX_DELEGATION_DEPTH depends on the deployment topology. Linear orchestration chains — root issuer, one or two planning layers, leaf executor — require few hops. Swarm architectures with dynamic fan-out, sub-task delegation, or hierarchical agent groups may require significantly deeper chains. The implementation ceiling should reflect the maximum depth the deployment actually needs, not an arbitrary conservative default.

Regardless of the implementation ceiling, issuers should set del_max_depth in individual tokens to the minimum depth the specific workflow requires. A grant with a lower del_max_depth than the implementation ceiling is always permitted and limits blast radius if a token is misused. The security value comes from tight per-chain policy, not from a low implementation ceiling.

B.6. Implementation Size Limits

The normative requirement is only that implementations enforce finite limits on token size, chain size, constraint nesting depth, and tool count to prevent resource exhaustion. This appendix provides non-normative recommended defaults for implementations with no specific deployment constraints:

Parameter	Recommended Default
Maximum token size	64 KB
Maximum chain size	256 KB
Maximum tools per token	256
Maximum constraints per tool	64
Maximum constraint nesting depth	32
Maximum tool name length	256 bytes
Maximum constraint value length	4 KB

Table 10

Deployments should document their enforced limits. Interoperating parties should verify that their respective limits are compatible before deployment.

Implementations should prefer exact and one_of constraints over pattern, regex, or cel where the policy permits, as these types produce significantly more compact tokens and simpler subsumption checks.

Implementations concerned about parser exposure on unverified payloads in step 2c of the chain verification algorithm (Section 7) may extract jti using a length-limited byte scan rather than a full JSON parser, provided the extraction correctly handles JSON whitespace and string escaping.

A single AAT is typically 1-4 KB when base64url-encoded. Chains of two or more tokens will commonly exceed the 4-8 KB header size limits enforced by common reverse proxies and load balancers, resulting in 431 errors. Deployments should transmit AAT chains in a request body field rather than an HTTP header. For size-constrained environments, the CBOR/CWT profile in Appendix D reduces chain size by 30-50% and is recommended when HTTP header transport is required.

B.7. Signed Passthrough Metadata

Deployments may need to convey additional signed metadata through the delegation chain, such as a request trace identifier, a tenant context used for logging or routing, or a human-readable subject identifier. This specification does not define a mechanism for such metadata, but the JWT format accommodates it naturally.

Implementations may include additional JWT claims in AATs beyond those defined in Section 3. Claims used for passthrough metadata should use collision-resistant names (e.g., reverse domain notation such as com.example.trace_id) and should not encode tool permissions or argument constraints that this specification models in authorization_details.

Because additional claims are included in the token's JWS signature, they are integrity-protected within each individual token. However, this specification's chain verification algorithm (Section 7) does not enforce preservation of unrecognized claims across derivation steps. Per Section 3.4, enforcement points ignore unrecognized top-level JWT claims; the fail-closed rule applies only to constraint types within authorization_details. A token carrying a com.example.trace_id claim will not be rejected solely for containing that claim. Deployments that require chain-wide preservation of passthrough metadata must define and enforce their own derivation and verification rules for those claims, either through deployment-specific policy or in a companion profile.

B.8. TTL Guidance

The normative requirement is only that derived tokens cannot outlive their parents and that token lifetime does not exceed MAX_TOKEN_LIFETIME (Section 4.4). This appendix provides non-normative guidance for selecting appropriate TTL values.

TTL is the primary revocation mechanism in this specification. A token that has expired cannot be used regardless of whether a revocation list exists. Short lifetimes reduce the window of exposure from key compromise, token theft, or scope misconfiguration. The operational cost of short TTLs is re-issuance frequency; this cost is low when the root issuer is available and derivation is offline.

The appropriate TTL for each token role depends on the deployment context. Root delegation tokens should be long enough to cover the full orchestration and execution window for the task, but no longer. Leaf execution tokens should be scoped to the expected duration of a single tool invocation. Deployments with intermittent connectivity (edge, embedded, or air-gapped) may need longer lifetimes, with the awareness that longer lifetimes expand the compromise window.

Deployments should treat TTL as a policy expression rather than a convenience parameter. A root delegation token with a 24-hour TTL effectively grants the holder 24 hours of authority regardless of how narrowly the capability scope is defined.

Appendix C. Policy Languages with Decidable Containment (Non-Normative)

This appendix provides non-normative guidance for implementers considering extension constraint types that use structured policy languages.

The core constraint types regex and cel use conservative syntactic subsumption strategies because this specification does not define general containment algorithms for their respective languages. This conservatism limits expressiveness: two semantically subsuming expressions that do not satisfy the syntactic criteria will be rejected by enforcement points applying the normative strategy, even if a human reviewer could determine that subsumption holds.

Implementers requiring richer policy expressiveness without sacrificing subsumption decidability should consider languages that were designed specifically for authorization use cases and that provide formal containment algorithms as a first-class operation. Such languages are better positioned to provide conforming extension constraint registrations under Section 3.5.1, because their containment algorithms are decidable and formally verified rather than approximated by conservative syntactic rules.

The key property to look for is whether the language's policy containment problem ("does every input permitted by policy A also satisfy policy B?") is decidable and implemented in available tooling. Languages with this property allow a registration to specify a subsumption verification procedure that is both decidable and non-conservative: it returns true for all semantically subsuming pairs, not just syntactically obvious ones.

This document takes no position on which specific policy language implementers should choose. The choice depends on the deployment environment, existing infrastructure, tooling availability, and the specific authorization semantics required. The normative requirement is only that whatever language is used, the resulting extension constraint registration satisfies the three properties defined in Section 3.5.1: decidable, sound, and deterministic.

Appendix D. CBOR/CWT Profile (Normative)

The claim semantics, attenuation invariants, constraint subsumption rules, and chain verification algorithm defined in this document are format-agnostic. They describe a protocol, not an encoding. This appendix notes the relationship to CBOR-based token formats for implementers operating in constrained or throughput-sensitive environments. The normative content of this appendix is limited to encoding constraints (deterministic CBOR per [RFC8949]); integer CWT claim key assignments are deferred to a companion document and are not normative here.

D.1. CWT Representation

CBOR Web Tokens [RFC8392] and COSE message signing [RFC9052] are the IETF-native binary analogs of JWT and JOSE respectively. An AAT can be represented as a CWT with no change to its semantic content. The attenuation invariants in Section 4 apply identically. The chain verification algorithm in Section 7 applies identically, substituting COSE_Sign1 verification for JWS signature verification.

CBOR encoding offers meaningful size advantages over base64url-encoded JSON for token payloads. In typical AAT payloads with several constraint entries, CBOR encoding reduces token size by 30-50% relative to compact JWT serialization. For high-throughput agent pipelines or resource-constrained edge deployments, this difference is operationally significant.

D.2. Deterministic Encoding

CWT implementations of this protocol MUST use deterministic CBOR encoding as defined in [RFC8949] Section 4.2. Deterministic encoding requires that integer keys be used for all map entries where assignments exist, that map keys be sorted in length-first, bytewise lexicographic order, and that the shortest-form encoding be used for all values. This ensures that two compliant implementations produce identical byte sequences for the same AAT payload, which is required for correct par_hash computation and cross-implementation chain verification.

The hta map within a CWT PoP token MUST also use deterministic CBOR encoding. Implementations MUST NOT use indefinite-length encoding for any AAT or PoP token structure.

D.3. Claim Key Assignments

JWT uses string claim names. CWT uses integer claim keys for registered claims to achieve compact encoding. The AAT-specific claims defined in Sections 3 and 5 — namely aat_type, del_depth, del_max_depth, par_hash, aat_tool, hta, and aat_id — require integer key assignments in the IANA CWT Claims Registry [RFC8392] before a normative CWT profile can be published.

Those registrations, along with COSE algorithm guidance and CWT-specific serialization rules, are deferred to a companion Internet-Draft. That companion document will reference [RFC8392] and [RFC9052] normatively; this document references them informatively. This document makes no CWT claim key assignments.

D.4. Companion Document

A normative CWT profile is intended to be published as a companion Internet-Draft. That document will define integer CWT claim key assignments for all AAT-specific claims, COSE algorithm requirements, and any CWT-specific serialization considerations. The protocol defined in this document serves as the common specification that both profiles reference.

Appendix E. Implementation Status (Non-Normative)

This appendix describes the implementation status of this specification at the time of submission, per the practice described in RFC 7942.

E.1. Reference Implementation

A reference implementation of this protocol exists. The chain verification algorithm (Section 7) and token derivation procedure (Section 6) are both implemented. The implementation uses CBOR encoding for the wire format, with Ed25519 signatures carried in COSE_Sign1 structures, and demonstrates that the Section 7 verification algorithm operates correctly over both JWT and CWT representations of the same chain, confirming format independence of the core protocol.

RFC Editor Note: Implementation attribution will be updated or removed prior to WG adoption per IETF norms.

E.2. Formal Verification

Formal verification of the attenuation algebra is in progress, using three complementary techniques: bounded model checking ([ALLOY]) for set-theoretic constraint types, SMT solving ([Z3]) for domain-specific types requiring arithmetic or string reasoning, and property-based testing against the Rust implementation for all constraint types. Bounded model checking has found no counterexamples for scopes up to 8 constraints and 8 values. The combination is intended to provide evidence toward monotonicity of the I4 invariant across the full constraint attenuation matrix.

RFC Editor Note: Implementation details will be updated or removed per IETF norms prior to publication.

Author’s Address

Niki Aimable Niyikiza
Tenuo

Email: niki@tenuo.ai