Workgroup:	Network Working Group
Internet-Draft:	draft-nennemann-act-01
Published:	12 April 2026
Intended Status:	Standards Track
Expires:	14 October 2026
Author:	C. Nennemann *Independent Researcher*

Agent Context Token (ACT)

Abstract

This document defines the Agent Context Token (ACT), a self-contained JWT-based format that captures the full invocation context of an autonomous AI agent — its capabilities, constraints, delegation provenance, oversight requirements, task metadata, and DAG position — and unifies authorization and execution accountability in a single token lifecycle. An ACT begins as a signed authorization mandate and transitions into a tamper-evident execution record once the agent completes its task, appending cryptographic hashes of inputs and outputs and linking to predecessor tasks via a directed acyclic graph (DAG). ACT requires no Authorization Server, no workload identity infrastructure, and no transparency service for basic operation. Trust is bootstrapped via pre-shared keys and is upgradeable to PKI or Decentralized Identifiers (DIDs). ACT is designed for cross-organizational agent federation in regulated and unregulated environments alike. ACT is the general-purpose agent context primitive; the WIMSE Execution Context Token (ECT) [I-D.nennemann-wimse-ect] is a sibling profile specialized for workload-identity-bound execution recording in WIMSE deployments.

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 14 October 2026.

Copyright Notice

Table of Contents

1. Introduction

Autonomous AI agents increasingly operate across organizational boundaries, executing multi-step workflows where individual tasks are delegated from one agent to another. These workflows create two distinct, inseparable compliance requirements:

1. **Authorization**: was the agent permitted to perform the action, under what constraints, and by whose authority?
2. **Accountability**: what did the agent actually do, with what inputs, producing what outputs, in what causal relationship to prior tasks?

Existing specifications address these requirements in isolation. The Agent Authorization Profile (AAP) [I-D.aap-oauth-profile] provides structured authorization via OAuth 2.0 but requires a central Authorization Server. The WIMSE Execution Context Token [I-D.nennemann-wimse-ect] provides execution accountability but requires WIMSE workload identity infrastructure (SPIFFE/SPIRE).

This document defines the Agent Context Token (ACT), which addresses both requirements in a single, self-contained token that requires no shared infrastructure beyond the ability to verify asymmetric signatures. The word "Context" in the name reflects what the token carries: the complete invocation context of an agent — DAG references, task metadata, capabilities, delegation chain, and oversight claims — bound together in one cryptographically verifiable envelope. ACT is positioned as the general agent context primitive, with the WIMSE Execution Context Token (ECT) [I-D.nennemann-wimse-ect] as a sibling profile specialized for workload-identity-bound execution contexts in WIMSE deployments.

1.1. Problem Statement

Cross-organizational agent federation today faces a bootstrapping problem: deploying shared OAuth infrastructure or a common SPIFFE trust domain requires organizational agreement before the first message is exchanged. In practice this means either:

(a) agents operate without cryptographic authorization or audit trails, relying on application-layer access control only; or

(b) organizations adopt one party's identity infrastructure, creating a hub-and-spoke dependency that contradicts the decentralized nature of agent networks.

ACT solves this by making pre-shared keys the mandatory-to-implement trust baseline — two agents can begin a secure, auditable interaction with nothing more than an out-of-band key exchange — while providing a clean upgrade path to PKI or DID-based trust without changing the token format.

1.2. Design Goals

- **G1 — Zero infrastructure baseline**: ACT **MUST** be deployable with no shared servers, no common identity provider, and no transparency service.
- **G2 — Single token lifecycle**: Authorization and accountability **MUST** be expressed in the same token format to prevent authorization-accountability gaps.
- **G3 — Peer-to-peer delegation**: Delegation chains **MUST** be verifiable without contacting an Authorization Server, using cryptographic chaining of agent signatures.
- **G4 — DAG-native causal ordering**: Workflows with parallel branches and fan-in dependencies **MUST** be expressible natively, without flattening to a linear chain.
- **G5 — Cross-organizational interoperability**: ACTs issued by agents in different trust domains **MUST** be verifiable by any participant holding the issuing agent's public key.
- **G6 — Regulatory applicability**: ACT **MUST** provide sufficient evidence for audit requirements in DORA [DORA], EU AI Act Article 12 [EUAIA], and IEC 62304 [IEC62304] without requiring additional log formats.
- **G7 — Upgrade path**: The trust model **MUST** support migration from pre-shared keys to PKI or DID without breaking existing ACT chains.

1.3. Non-Goals

The following are explicitly out of scope:

- Defining internal AI model behavior or decision logic.
- Replacing organizational security policies or procedures.
- Defining storage formats for audit ledgers.
- Specifying token revocation infrastructure (deployments **MAY** use existing mechanisms such as [RFC7009] for this purpose).
- Providing non-equivocation guarantees in standalone mode (see Section 11.5 for the equivocation discussion and optional transparency anchoring).

1.4. Relationship to Related Work

AAP [I-D.aap-oauth-profile]: ACT addresses the same authorization problem as AAP but does not require an Authorization Server. ACT delegation is peer-to-peer via cryptographic signature chaining; AAP delegation requires OAuth Token Exchange [RFC8693] against a central AS. ACT is not a profile of AAP; it is an infrastructure-independent alternative for the same problem class.

WIMSE ECT [I-D.nennemann-wimse-ect]: ACT addresses the same execution accountability problem as the WIMSE Execution Context Token but does not require WIMSE workload identity infrastructure. ACT is not a profile of WIMSE; it is deployable in environments without SPIFFE/SPIRE. In environments where WIMSE is deployed, ACT **MAY** be carried alongside WIMSE tokens to augment accountability with authorization provenance.

SCITT [I-D.ietf-scitt-architecture]: For deployments requiring non-equivocation guarantees (see Section 11.5), ACT execution records **MAY** be anchored to a SCITT Transparency Service as a Layer 2 mechanism. This is **OPTIONAL** and not required for basic ACT operation. Note: The SCITT architecture draft is currently in AUTH48 (RFC Editor queue) at version -22 and is about to become an RFC; readers should use the RFC number once assigned.

1.4.1. Concurrent Agent Authorization Proposals

Several concurrent proposals in the IETF and academic communities address overlapping portions of the agent authorization problem space. This subsection situates ACT relative to those proposals. Protocol-layer comparison of linear versus DAG delegation structure is deferred to Section 7.3; the summaries below focus on scope and deployability.

AIP / IBCTs [AIP-IBCT]: The Agent Interaction Protocol proposes Interaction-Bound Capability Tokens in two modes: compact signed JWTs for single-hop invocation and Biscuit/Datalog tokens for multi-hop delegation, motivated by a survey of approximately 2,000 Model Context Protocol servers that found no authorization enforcement. ACT addresses the same problem class but relies exclusively on JWT/JOSE throughout (no Biscuit or Datalog dependency), defines an explicit two-phase lifecycle separating authorization (Mandate) from proof-of-execution (Record), and supports DAG delegation structure. IBCTs are modeled as append-only chains at the protocol layer; ACT operates at the authorization graph layer with revocable lifecycle states.

SentinelAgent [SentinelAgent]: SentinelAgent defines a formal Delegation Chain Calculus with seven verifiable properties, a TLA+ mechanization, and reports 100% true-positive and 0% false-positive rates against the DelegationBench v4 benchmark. It addresses the same accountability question as ACT — namely, which principal authorized a given chain of actions. The differentiator is deployment substrate: SentinelAgent expresses its guarantees in a domain-specific formal calculus, whereas ACT encodes the same invariants in IETF-standard JWT infrastructure (RFC 7519, RFC 7515, RFC 8032) already deployable in existing OAuth- and JOSE-aware stacks.

Agentic JWT [AgenticJWT]: Agentic JWT derives a per-agent identity as a one-way hash of the agent's prompt, registered tools, and configuration, and chains delegation assertions across invocations. It is the closest prior-art JWT-based construction for agentic delegation. ACT differs in that it adds an explicit two-phase lifecycle — separating the authorization mandate from the proof-of-execution record — and expresses delegation as a DAG via the array-valued `pred` claim rather than a strictly linear chain.

OAuth Transaction Tokens for Agents [I-D.oauth-transaction-tokens-for-agents]: This draft extends OAuth Transaction Tokens with an `actchain` claim (an ordered delegation array), an `agentic_ctx` claim conveying intent and constraints, and flow-type markers distinguishing interactive from autonomous invocations. It is complementary to ACT at the OAuth layer. The primary differentiators are topology and infrastructure dependency: Transaction Tokens for Agents presume an OAuth Authorization Server and use a linear `actchain`, whereas ACT operates peer-to-peer without any AS and uses a DAG-valued `pred`. A detailed differencing document is referenced in Section 11.

Helixar Delegation Protocol (HDP) [I-D.helixar-hdp-agentic-delegation]: HDP specifies Ed25519 signatures over RFC 8785-canonicalized JSON, an append-only linear delegation chain with session binding, and offline verification. ACT addresses the same problem but is encoded in JWT/JOSE (aligning with the broader IETF token ecosystem) rather than raw canonical JSON, and its `pred` claim admits DAG topologies rather than strictly linear chains.

SCITT Profile for AI Agent Execution Records [I-D.emirdag-scitt-ai-agent-execution]: This draft defines a SCITT profile in which AgentInteractionRecord (AIR) payloads are carried as COSE_Sign1 statements anchored to a SCITT Transparency Service. It is highly complementary to ACT: where ACT defines the two-phase lifecycle token issued and consumed by agents at runtime, the SCITT AI Agent Execution draft defines the payload format suitable for long-term anchoring. Implementations that anchor Phase 2 ACTs to SCITT (Section 11) **SHOULD** consider the AIR payload structure defined in that draft as the canonical encoding for anchored records.

1.5. Applicability

ACT is designed as a general-purpose primitive for AI agent authorization and execution accountability. While a sibling specification [I-D.nennemann-wimse-ect] profiles execution context tokens specifically for the WIMSE working group's workload identity infrastructure, ACT operates without any shared identity plane. This section identifies deployment contexts where ACT applies independently of WIMSE, and clarifies how ACT complements — rather than competes with — ecosystem-specific agent protocols.

1.5.1. Model Context Protocol (MCP) Tool-Use Flows

The Model Context Protocol [MCP-SPEC] defines a client-server interface by which LLM hosts invoke external tools via structured JSON-RPC calls. MCP 2025-11-25 mandates OAuth 2.1 for transport-layer authentication, but provides no mechanism for carrying per-invocation authorization constraints or for producing a tamper-evident record of what arguments were passed and what result was returned.

ACT addresses this gap as follows: when an MCP host is about to dispatch a tool call on behalf of an agent, it **SHOULD** issue a Phase 1 ACT Mandate encoding the permitted tool name (e.g., as a capability constraint), the declaring scope, and any parameter-level constraints applicable to that invocation. The MCP server, upon receiving the request, **MAY** validate the ACT Mandate and, upon completing the tool execution, **SHOULD** transition the token to Phase 2 by appending SHA-256 hashes of the serialized input arguments and the JSON response, then re-sign. The resulting Phase 2 ACT constitutes an unforgeable record that a specific tool was called with specific arguments and returned a specific result, independently of MCP's OAuth layer.

This integration requires no modification to MCP transport; the ACT **SHOULD** be carried in the `ACT-Mandate` and `ACT-Record` HTTP headers defined in Section 9.1 of this document.

1.5.2. OpenAI Agents SDK and Function Calling

The OpenAI Agents SDK [OPENAI-AGENTS-SDK] enables composition of agents via handoffs — structured transfers of control from one agent to another, each potentially invoking registered function tools. The SDK provides no built-in mechanism for a receiving agent to verify that the handoff was authorized by a named principal, nor for the invoking agent to produce a verifiable record of what functions it called.

ACT is applicable at the handoff boundary: the orchestrating agent **SHOULD** issue a Phase 1 ACT Mandate to the receiving agent at the moment of handoff, encoding the permitted function set as capability constraints and the maximum privilege the receiving agent **MAY** exercise. The receiving agent **SHOULD** attach its Phase 2 ACT Record to any callback or downstream response, providing the orchestrator with cryptographic evidence of the actions taken. In multi-turn chains involving multiple handoffs, the DAG linkage (Section 7) allows each handoff to be expressed as a parent-child edge, preserving the full causal ordering of the agent invocation sequence.

Implementations that use the OpenAI function calling API directly, without the Agents SDK, **MAY** apply ACT at the application layer: the calling process issues a Phase 1 ACT before the function call parameter block is finalized, and the receiving function handler returns a Phase 2 ACT alongside its JSON result.

1.5.3. LangGraph and LangChain Agent Graphs

LangGraph [LANGGRAPH] models agent workflows as typed StateGraphs in which nodes represent agent invocations or tool calls and edges represent conditional transitions. The DAG structure of ACT (Section 7) is a natural fit for this model: each LangGraph node that performs an observable action corresponds to exactly one ACT task identifier (`tid`), and directed edges in the LangGraph correspond to `pred` (predecessor) references in successor ACTs.

ACT is applicable at the node boundary: when a LangGraph node dispatches a sub-agent or invokes a tool with side effects, it **SHOULD** issue a Phase 1 ACT Mandate encoding the node's permitted actions before any external call is made. Upon transition out of the node, a Phase 2 ACT Record **SHOULD** be produced and attached to the LangGraph state object alongside the node's output. Downstream nodes that fan-in from multiple predecessors **MAY** retrieve the set of parent ACT identifiers from the shared state to populate their `pred` array, thereby expressing LangGraph's fan-in semantics within the ACT DAG without any additional infrastructure.

In contrast to LangGraph's built-in state audit trail, which is mutable in-process memory, Phase 2 ACTs are cryptographically signed and portable: they can be exported from a LangGraph run and submitted to an external audit ledger, satisfying compliance requirements that cannot be met by in-process logging alone.

1.5.4. Google Agent2Agent (A2A) Protocol

The Agent2Agent protocol [A2A-SPEC] defines a task-oriented JSON-RPC interface for inter-agent communication, with authentication delegated to OAuth 2.0 or API key schemes declared in each agent's Agent Card. A2A provides no mechanism for a receiving agent to verify the authorization provenance of a task request beyond the transport-layer credential, and produces no token that represents the execution of the task in a verifiable, portable form.

ACT is applicable as a session-layer accountability complement to A2A: a client agent **SHOULD** include a Phase 1 ACT Mandate in the `metadata` field of the A2A Task object, encoding the task type as a capability constraint and the delegating agent's identity as the ACT issuer. The receiving agent **SHOULD** validate the Mandate before beginning task execution and **SHOULD** return a Phase 2 ACT Record as an artifact in the A2A TaskResult, enabling the client agent to retain cryptographic proof of what was executed on its behalf.

This integration does not require modification to A2A's transport or authentication scheme; ACT and A2A's OAuth credentials operate at independent layers and are not redundant. A2A's credential answers "is this client permitted to contact this server?"; the ACT Mandate answers "is this agent permitted to request this specific task under these constraints?".

1.5.5. Enterprise Orchestration Without WIMSE (CrewAI, AutoGen)

Enterprise orchestration frameworks such as CrewAI [CREWAI] and AutoGen [AUTOGEN] deploy multi-agent systems within a single organizational boundary, typically without SPIFFE/SPIRE workload identity infrastructure. In these environments, OAuth Authorization Servers are often unavailable or impractical to deploy for intra-process agent communication.

ACT is applicable in this context via its Tier 1 (pre-shared key) trust model (Section 5.2): each agent role in a CrewAI Crew or AutoGen ConversableAgent graph is assigned an Ed25519 keypair at instantiation time. The orchestrating agent issues Phase 1 Mandates to worker agents before delegating tasks, constraining each worker to only the tools and actions relevant to its role. Worker agents produce Phase 2 Records on task completion. The resulting ACT chain is exportable as a structured audit trail that satisfies the per-action logging requirements of DORA [DORA] and EU AI Act Article 12 [EUAIA] without requiring shared infrastructure beyond the ability to exchange public keys at deployment time.

Implementations **SHOULD NOT** use ACT's self-assertion mode (where an agent issues and records its own mandate without external sign-off) in regulated workflows; at minimum, the orchestrating agent **MUST** sign the initial Mandate so that accountability is anchored to a principal outside the executing agent.

1.5.6. Relationship to WIMSE ECT

Where WIMSE infrastructure is deployed, ACT and the WIMSE Execution Context Token [I-D.nennemann-wimse-ect] serve complementary and non-overlapping functions. The ECT records workload-level execution in WIMSE terms — which SPIFFE workload executed, in which trust

domain, against which service. ACT records the authorization provenance — which agent was permitted to request which action, under what capability constraints, by whose authority — and transitions that authorization record into an execution record upon task completion.

In mixed environments, both tokens **SHOULD** be carried simultaneously: the `Workload-Identity` header carries the WIMSE ECT; the `ACT-Record` header carries the ACT. Verifiers **MAY** correlate the two by matching the ACT `tid` claim against application-layer identifiers present in the ECT's task context. Neither token is a profile or extension of the other; they operate at different abstraction layers and their co-presence is additive.

2. Conventions and Definitions

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.

Agent: An autonomous software entity that executes tasks, issues ACTs as mandates for sub-agents, and produces ACTs as execution records of its own actions.

Authorization Mandate: An ACT in Phase 1, encoding what an agent is permitted to do, under what constraints, and by whose authority.

Execution Record: An ACT in Phase 2, encoding what an agent actually did, including cryptographic hashes of inputs and outputs and causal links to predecessor tasks.

Directed Acyclic Graph (DAG): A graph structure representing task dependency ordering where edges are directed and no cycles exist. Used by ACT to model causal relationships between tasks in a workflow.

Delegation Chain: A cryptographically verifiable sequence of ACT issuances from a root authority through one or more agents, each signing a new ACT that reduces privileges relative to the one it received.

Trust Tier: A level of key management infrastructure used to establish the public key of an ACT issuer. Tiers range from pre-shared keys (Tier 1, mandatory) to PKI (Tier 2) and DIDs (Tier 3).

Workflow: A set of related tasks, identified by a shared `wid` claim, forming a single logical unit of work.

3. ACT Lifecycle

An ACT has a two-phase lifecycle. The same token format is used in both phases; the presence or absence of execution claims determines which phase a token represents.

A token is a **Phase 2 Execution Record** if and only if the claim `exec_act` is present. A token that does not contain `exec_act` is a **Phase 1 Authorization Mandate**. Verifiers **MUST** determine the phase before applying verification rules, and **MUST** reject a token that is presented in the wrong phase for the operation being performed.

3.1. Phase 1: Authorization Mandate

In Phase 1, an ACT is created by a delegating agent (or a human operator) to authorize a target agent to perform a specific task. The token carries:

- The identity of the issuing agent and the target agent.
- The capabilities granted, with associated constraints.
- Human oversight requirements for high-impact actions.
- The delegation provenance (who authorized the issuer to delegate).
- A task identifier and declared purpose.

The Phase 1 ACT is signed by the issuing agent using its private key. The target agent receives the ACT and uses it as a bearer mandate — evidence that it is authorized to proceed.

Phase 1 ACTs are short-lived. Implementations **SHOULD** set expiration (`exp`) to no more than 15 minutes after issuance (`iat`) for automated agent-to-agent workflows. Longer lifetimes **MAY** be used for human-initiated mandates where the agent may not act immediately.

3.2. Phase 2: Execution Record

Upon completing the authorized task, the executing agent **MUST** transition the ACT to Phase 2 by:

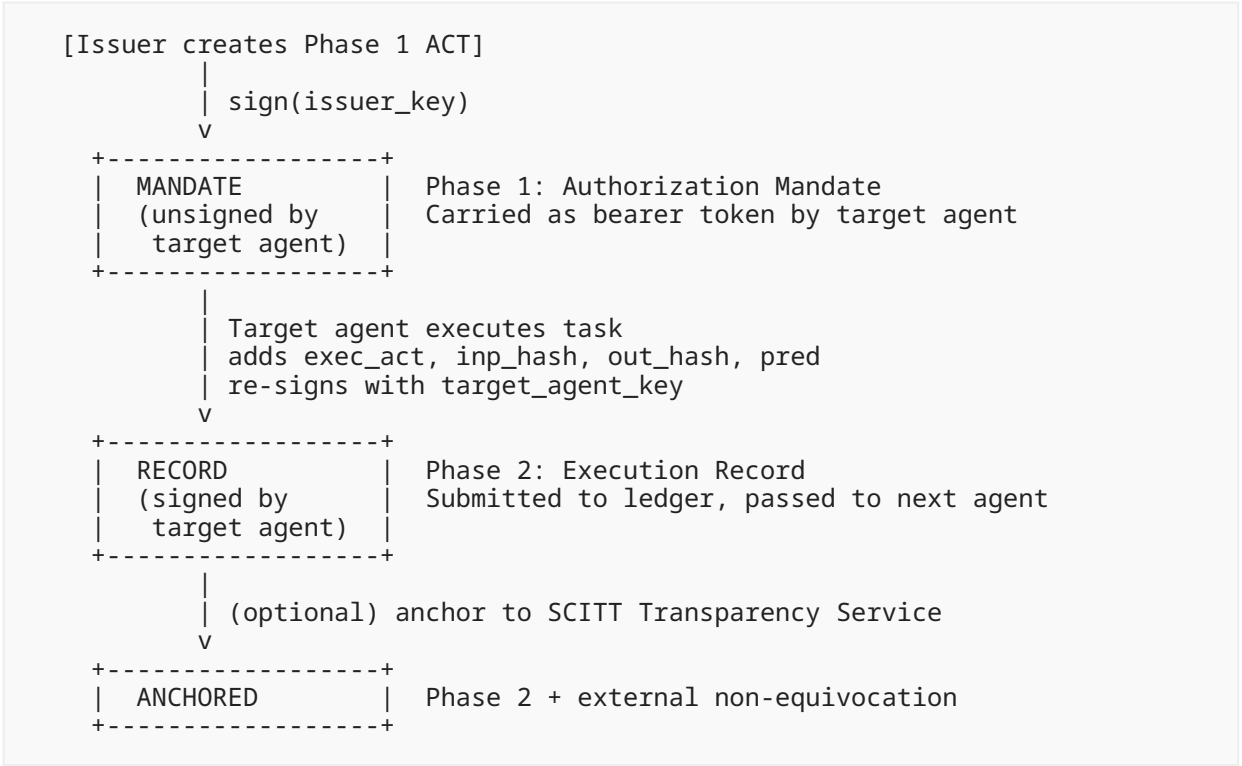
1. Adding the `exec_act` claim describing the action performed.
2. Optionally adding `inp_hash` and/or `out_hash` SHA-256 hashes of task inputs and outputs (**RECOMMENDED** for regulated environments).
3. Adding the `pred` array referencing predecessor task identifiers (DAG dependencies).
4. Adding `exec_ts` and `status` claims.
5. Re-signing the complete token with its own private key.

The re-signing is critical: it produces a new signature over the combined authorization + execution claims, binding the executing agent's cryptographic identity to both the mandate it received and the execution it performed. This creates a single, non-repudiable record that answers both "was this agent authorized?" and "what did it do?"

Note on issuer signature preservation: re-signing replaces the Phase 1 signature produced by the issuing agent (`iss`). The integrity of the original mandate is preserved through the `del.chain` mechanism: the chain entry's `sig` field is the `iss` agent's signature over the Phase 1 ACT, and this signature remains intact and verifiable in the Phase 2 token. For root mandates where `del.chain` is empty, the issuer's signature is not independently preserved in Phase 2. Deployments requiring independent verifiability of the original mandate **SHOULD** retain the Phase 1 ACT separately alongside the Phase 2 record.

The resulting Phase 2 ACT **SHOULD** be submitted to an audit ledger (Section 10) and **MAY** be sent to the next agent in the workflow as evidence of completed prerequisites.

3.3. Lifecycle State Machine



4. ACT Token Format

An ACT is a JSON Web Token [RFC7519] signed as a JSON Web Signature [RFC7515] using JWS Compact Serialization. All ACTs **MUST** use JWS Compact Serialization to ensure they can be carried in a single HTTP header value.

4.1. JOSE Header

The ACT JOSE header **MUST** contain:

```
{
  "alg": "ES256",
  "typ": "act+jwt",
  "kid": "agent-a-key-2026-03"
}
```

alg (**REQUIRED**): The digital signature algorithm. Implementations **MUST** support ES256 [RFC7518]. EdDSA (Ed25519) [RFC8037] is **RECOMMENDED** for new deployments due to smaller signatures and resistance to side-channel attacks. Symmetric algorithms (HS256, HS384, HS512) **MUST NOT** be used. The "alg" value **MUST NOT** be "none".

typ (**REQUIRED**): **MUST** be "act+jwt" to distinguish ACTs from other JWT types.

kid (**REQUIRED**): An identifier for the signing key. In Tier 1 deployments (pre-shared keys), this is an opaque string agreed out-of-band. In Tier 2 deployments (PKI), this is the X.509 certificate thumbprint. In Tier 3 deployments (DID), this is the DID key fragment (e.g., `did:key:z6Mk...#key-1`).

x5c (**OPTIONAL**): In Tier 2 deployments, the X.509 certificate chain **MAY** be included to enable verification without out-of-band key distribution.

did (**OPTIONAL**): In Tier 3 deployments, the full DID of the issuing agent **MAY** be included for resolution.

4.2. JWT Claims: Authorization Phase

4.2.1. Standard JWT Claims

iss (**REQUIRED**): The identifier of the agent issuing the mandate. Format depends on trust tier: an opaque string (Tier 1), an X.509 Subject DN (Tier 2), or a DID (Tier 3).

sub (**REQUIRED**): The identifier of the agent authorized to act. **MUST** use the same format convention as `iss`.

aud (**REQUIRED**): The intended recipient(s). **MUST** include the identifier of the target agent (`sub`). When an audit ledger is deployed, **MUST** also include the ledger's identifier. When multiple recipients are present, **MUST** be an array. Verifiers that are audit ledgers **MUST** verify that their own identifier appears in `aud`.

iat (**REQUIRED**): Issuance time as a NumericDate [RFC7519].

exp (**REQUIRED**): Expiration time. Implementations **SHOULD** set to no more than 15 minutes after `iat` for automated workflows.

jti (**REQUIRED**): A UUID [RFC9562] uniquely identifying this ACT and, in Phase 2, the task it records. Used as the task identifier for DAG predecessor references in `pred`.

4.2.2. ACT Authorization Claims

wid (**OPTIONAL**): A UUID identifying the workflow to which this task belongs. When present, groups related ACTs and scopes `jti` uniqueness to the workflow.

task (**REQUIRED**): An object describing the authorized task:

```
{
  "task": {
    "purpose": "validate_patient_dosage",
    "data_sensitivity": "restricted",
    "created_by": "operator:clinical-admin-01",
    "expires_at": 1772064750
  }
}
```

- `purpose` (**REQUIRED**): A string describing the intended task. Implementations **SHOULD** use a controlled vocabulary or reverse- domain notation (e.g., "com.example.validate_dosage") to enable semantic consistency checking by the receiving agent.
- `data_sensitivity` (**OPTIONAL**): One of "public", "internal", "confidential", "restricted". Receiving agents **MUST NOT** perform actions that would expose data above this classification.
- `created_by` (**OPTIONAL**): An identifier for the human or system that initiated the workflow. **SHOULD** be pseudonymous (see Section 12).
- `expires_at` (**OPTIONAL**): A NumericDate after which the task mandate is no longer valid, independent of exp.

cap (**REQUIRED**): An array of capability objects, each specifying an action the agent is authorized to perform and the constraints under which it may do so:

```
{
  "cap": [
    {
      "action": "read.patient_record",
      "constraints": {
        "patient_id_scope": "current_task_only",
        "max_records": 1,
        "data_classification_max": "restricted"
      }
    },
    {
      "action": "write.dosage_recommendation",
      "constraints": {
        "status": "draft_only"
      }
    }
  ]
}
```

Action names **MUST** conform to the ABNF grammar:

```
action-name = component *( "." component )
component   = ALPHA *( ALPHA / DIGIT / "-" / "_" )
```

Receiving agents **MUST** perform exact string matching on action names. Wildcard matching is NOT part of this specification.

When multiple capabilities match the same action, OR semantics apply: if ANY capability grants the action, the request is authorized subject to that capability's constraints. When multiple constraints exist within a single capability, AND semantics apply: ALL constraints **MUST** be satisfied. When the same constraint key appears in both a capability-level and a policy-level context, the more restrictive value applies: lower numeric limits, narrower allow-lists (intersection), broader block-lists (union), and narrower time windows.

oversight (**OPTIONAL**): Human oversight requirements:

```
{
  "oversight": {
    "requires_approval_for": ["write.publish", "execute.payment"],
    "approval_ref": "https://approval.example.com/workflow/w-123"
  }
}
```

When `requires_approval_for` lists an action, the receiving agent **MUST NOT** execute that action autonomously. The approval mechanism is out of scope for this specification.

del (**OPTIONAL**): Delegation provenance, establishing the chain of authority from the root mandate to this ACT. If `del` is absent, the ACT **MUST** be treated as a root mandate with `depth` = 0 and further delegation is not permitted (i.e., the receiving agent **MUST NOT** issue sub-mandates based on this ACT).

```
{
  "del": {
    "depth": 1,
    "max_depth": 3,
    "chain": [
      {
        "delegator":
"did:key:z6MkhaXgBZDvotDkL5257faiztiGiC2QtKLGpbnnEGta2doK",
        "jti": "550e8400-e29b-41d4-a716-446655440000",
        "sig": "base64url-encoded-signature-of-parent-act-hash"
      }
    ]
  }
}
```

- `depth`: The current delegation depth. 0 means this is a root mandate issued by a human or root authority.
- `max_depth`: The maximum permitted delegation depth. Receiving agents **MUST NOT** issue sub-mandates that would exceed this depth.
- `chain`: An array of delegation provenance records ordered from root to immediate parent (`chain[0]` is the root authority, `chain[depth-1]` is the direct parent of this ACT). Each entry contains:
  - `delegator`: The identifier of the agent that authorized this delegation step (i.e., the `iss` of the parent ACT at that depth).

- `jti`: The `jti` of the parent ACT that authorized this delegation step.
- `sig`: The delegating agent's signature over the SHA-256 hash of that parent ACT, providing cryptographic linkage without requiring the full parent ACT to be transmitted.

The `sig` field in each chain entry is the critical departure from AAP's delegation model: rather than requiring a central AS to validate the chain, any verifier holding the delegating agent's public key can independently verify each step by recomputing the hash and checking the signature.

4.3. JWT Claims: Execution Phase

The following claims are added by the executing agent when transitioning to Phase 2. Their presence distinguishes an Execution Record from an Authorization Mandate.

exec_act (**REQUIRED** in Phase 2): A string identifying the action actually performed. **MUST** conform to the same ABNF grammar as capability action names. **MUST** match one of the `action` values in the `cap` array of the Phase 1 claims.

pred (**REQUIRED** in Phase 2): An array of `jti` values of predecessor tasks in the DAG. An empty array indicates a root task. Each value **MUST** be the `jti` of a previously verified ACT (Phase 2) within the same workflow (same `wid`) or the global ACT store if `wid` is absent.

inp_hash (**OPTIONAL**): The base64url encoding (without padding) of the SHA-256 hash of the task's input data, computed over the raw octets of the serialized input. Provides cryptographic evidence of what data the agent processed.

out_hash (**OPTIONAL**): The base64url encoding (without padding) of the SHA-256 hash of the task's output data, using the same format as `inp_hash`. Provides cryptographic evidence of what data the agent produced.

exec_ts (**REQUIRED** in Phase 2): A NumericDate recording the actual time of task execution. **MAY** differ from `iat` when the agent queued the mandate before execution. **MUST** be greater than or equal to `iat`. **SHOULD** be less than or equal to `exp`; execution after mandate expiry is possible when tasks are long-running and **MUST NOT** cause automatic rejection, but implementors **SHOULD** log a warning.

status (**REQUIRED** in Phase 2): One of "completed", "failed", "partial". Allows audit systems to distinguish successful execution from partial or failed attempts, which is essential for regulated environments where failed attempts must be recorded.

err (**OPTIONAL**, present when `status` is "failed" or "partial"): An object providing error context:

```
{
  "err": {
    "code": "constraint_violation",
    "detail": "data_classification_max exceeded"
  }
}
```

Error detail **SHOULD NOT** reveal internal system state beyond what is necessary for audit purposes.

4.4. Complete Examples

4.4.1. Example: Phase 1 — Authorization Mandate

```
{
  "alg": "ES256",
  "typ": "act+jwt",
  "kid": "agent-clinical-key-2026-03"
}
.
{
  "iss": "did:key:z6MkhaXgBZDvotDkL5257faiztiGiC2QtKLGpbnnEGta2doK",
  "sub": "did:key:z6MknGc3omCyas4b1GmEn4xySHgLuSHxrKrUBnrhJekxZHFz",
  "aud": [
    "did:key:z6MknGc3omCyas4b1GmEn4xySHgLuSHxrKrUBnrhJekxZHFz",
    "https://ledger.hospital.example.com"
  ],
  "iat": 1772064000,
  "exp": 1772064900,
  "jti": "550e8400-e29b-41d4-a716-446655440001",

  "wid": "a0b1c2d3-e4f5-6789-abcd-ef0123456789",

  "task": {
    "purpose": "validate_treatment_recommendation",
    "data_sensitivity": "restricted",
    "created_by": "operator:clinical-admin-01"
  },

  "cap": [
    {
      "action": "read.patient_record",
      "constraints": {
        "patient_id_scope": "current_task_only",
        "max_records": 1
      }
    },
    {
      "action": "write.safety_assessment",
      "constraints": {
        "status": "draft_only"
      }
    }
  ],

  "oversight": {
    "requires_approval_for": ["write.publish_assessment"]
  },

  "del": {
    "depth": 0,
    "max_depth": 2,
    "chain": []
  }
}
```

4.4.2. Example: Phase 2 — Execution Record (same token, re-signed by target agent)

```
{
  "alg": "EdDSA",
  "typ": "act+jwt",
  "kid": "agent-safety-key-2026-03"
}
.
{
  "iss": "did:key:z6MkhaXgBZDvotDkL5257faiztiGiC2QtKLGpbnnEGta2doK",
  "sub": "did:key:z6MknGc3omCyas4b1GmEn4xySHgLuSHxrKrUBnrhJekxZHFz",
  "aud": [
    "did:key:z6MknGc3omCyas4b1GmEn4xySHgLuSHxrKrUBnrhJekxZHFz",
    "https://ledger.hospital.example.com"
  ],
  "iat": 1772064000,
  "exp": 1772064900,
  "jti": "550e8400-e29b-41d4-a716-446655440001",

  "wid": "a0b1c2d3-e4f5-6789-abcd-ef0123456789",

  "task": {
    "purpose": "validate_treatment_recommendation",
    "data_sensitivity": "restricted",
    "created_by": "operator:clinical-admin-01"
  },

  "cap": [
    {
      "action": "read.patient_record",
      "constraints": {
        "patient_id_scope": "current_task_only",
        "max_records": 1
      }
    },
    {
      "action": "write.safety_assessment",
      "constraints": {
        "status": "draft_only"
      }
    }
  ],

  "oversight": {
    "requires_approval_for": ["write.publish_assessment"]
  },

  "del": {
    "depth": 0,
    "max_depth": 2,
    "chain": []
  },

  "exec_act": "write.safety_assessment",
  "pred": ["550e8400-e29b-41d4-a716-446655440000"],
  "inp_hash": "n4bQgYhMfWWaL-qgxVrQFaO_TxsrC4Is0V1sFbDwCgg",
  "out_hash": "LCa0a2j_xo_5m0U8HTBBNBNCLXBkg7-g-YpeiGJm564",
  "exec_ts": 1772064300,
```

```
  "status": "completed"
}
```

5. Trust Model

ACT defines four trust tiers. Tier 1 is mandatory-to-implement; all others are optional upgrades. An ACT verifier **MUST** be able to process ACTs from any tier it has configured. The trust tier in use is determined by the `kid` format and the presence of `x5c` or `did` header parameters.

5.1. Tier 0: Bootstrap (TOFU — Trust On First Use)

Tier 0 is NOT part of the normative trust model and **MUST NOT** be used in regulated environments. It is defined here for documentation purposes only, to describe the common bootstrapping scenario.

In Tier 0, the first ACT received from an agent establishes its public key. This is equivalent to SSH TOFU behavior: an attacker who intercepts the first message can substitute their own key. Tier 0 deployments **MUST** transition to Tier 1 or higher before exchanging ACTs that carry sensitive capabilities.

5.2. Tier 1: Pre-Shared Keys (Mandatory-to-Implement)

In Tier 1, both parties exchange public keys out-of-band prior to the first ACT exchange. The `kid` is an opaque string agreed during the key exchange. Implementations **MUST** support Tier 1.

Key exchange **MAY** occur via any out-of-band mechanism: manual configuration, a configuration management system, or a prior authenticated channel. This specification does not mandate a specific key exchange protocol.

Tier 1 public keys **MUST** be Ed25519 [RFC8037] or P-256 (ES256) [RFC7518] keys. RSA keys **SHOULD NOT** be used in Tier 1 deployments due to key size. Key rotation **MUST** be performed out-of-band using the same mechanism as the initial exchange.

5.3. Tier 2: PKI / X.509

In Tier 2, agent identity is bound to an X.509 certificate issued by a mutually trusted Certificate Authority (CA). The `kid` is the certificate thumbprint (SHA-256 of the DER-encoded certificate).

Cross-organizational ACT exchange in Tier 2 requires either:

(a) a mutually trusted root CA, or (b) cross-certification between the organizations' CAs, or (c) explicit trust anchoring (one organization's CA is added to the other's trust store).

The `x5c` JOSE header parameter [RFC7515] **MAY** carry the full certificate chain to enable verification without out-of-band trust store configuration.

5.4. Tier 3: Decentralized Identifiers (DID)

In Tier 3, agent identity is expressed as a DID [W3C-DID]. The `kid` is a DID key fragment. The `did` JOSE header parameter carries the full DID for resolution.

Implementations **SHOULD** support at minimum `did:key` [DID-KEY] for self-contained key distribution without external resolution, and `did:web` [DID-WEB] for organizations that prefer DNS-anchored identity.

DID resolution latency introduces a dependency on external infrastructure. To preserve the zero-infrastructure baseline, implementations using Tier 3 **MAY** cache DID Documents and **MUST** specify a maximum cache TTL in their configuration.

5.5. Cross-Tier Interoperability

A delegation chain **MAY** include agents operating at different trust tiers. Each step in the chain is verified using the trust tier of the signing agent at that step. Verifiers **MUST NOT** reject a chain solely because it mixes trust tiers, but **MAY** apply stricter policy for chains that include Tier 0 or Tier 1 steps when exchanging sensitive capabilities.

6. Delegation Chain

ACT delegation is peer-to-peer: no Authorization Server is involved. Delegation is expressed as a cryptographically verifiable chain of ACT issuances, where each step reduces privileges relative to the previous step.

6.1. Peer-to-Peer Delegation

When Agent A authorizes Agent B to perform a sub-task, Agent A:

1. Creates a new ACT with `sub` set to Agent B's identifier.
2. Sets `cap` to a subset of A's own authorized capabilities, with constraints at least as restrictive as those in A's mandate.
3. Sets `del.depth` to A's own `del.depth + 1`.
4. Sets `del.max_depth` to no more than the `del.max_depth` value in A's own mandate.
5. Adds a chain entry containing A's identifier as `delegator`, the `jti` of A's own mandate, and a `sig` value computed as:

   ```
   sig = Sign(A.private_key, SHA-256(canonical_ACT_phase1_bytes))
   ```

   where `canonical_ACT_phase1_bytes` is the UTF-8 encoded bytes of the JWS Compact Serialization of A's Phase 1 ACT.
6. Signs the new ACT with A's private key.

6.2. Privilege Reduction Requirements

When issuing a delegated ACT, the issuing agent **MUST** reduce privileges by one or more of:

- Removing capabilities (sub-set of parent capabilities only).
- Adding stricter constraints (lower rate limits, narrower domains, shorter time windows, lower data classification ceiling).
- Reducing token lifetime (`exp` closer to `iat`).
- Reducing `del.max_depth`.

The issuing agent **MUST NOT** grant capabilities not present in its own mandate. Capability escalation via delegation is prohibited and **MUST** be detected and rejected by verifiers.

For well-known numeric constraints (e.g., `max_records`, `max_requests_per_hour`), "more restrictive" means a numerically lower or equal value. For well-known enumerated constraints (e.g., `data_sensitivity`), "more restrictive" means a value that is equal or higher in the defined ordering ("public" < "internal" < "confidential" < "restricted"). For unknown or domain-specific constraint keys, verifiers **MUST** treat the constraint as non-comparable and **MUST** reject the delegation unless the delegated constraint value is byte-for-byte identical to the parent constraint value.

6.3. Delegation Verification

A verifier receiving a delegated ACT **MUST**:

1. Verify the ACT's own signature (Section 8.1).
2. For each entry in `del.chain`, in order from index 0 to `del.depth - 1`: a. Retrieve the public key for `entry.delegator`. b. Verify that `entry.sig` is a valid signature over the SHA-256 hash of the referenced parent ACT (identified by `entry.jti`). c. Verify that the capabilities in the current ACT are a subset of the capabilities in the parent ACT, per the constraint comparison rules in Section 6.2.
3. Verify that `del.depth` does not exceed `del.max_depth`.
4. Verify that `del.chain` length equals `del.depth`.

If any step fails, the ACT **MUST** be rejected.

7. DAG Structure and Causal Ordering

ACTs in Phase 2 form a DAG over the `pred` (predecessor) claim. The DAG encodes causal dependencies: a task **MAY** NOT begin before all its parent tasks are completed.

7.1. DAG Validation

When processing a Phase 2 ACT, implementations **MUST**:

1. **Uniqueness**: Verify the `jti` is unique within the workflow (`wid`) or globally if `wid` is absent.
2. **Predecessor Existence**: Verify every `jti` in `pred` corresponds to a Phase 2 ACT available in the ACT store or audit ledger.
3. **Temporal Ordering**: Verify that for each parent: `parent.exec_ts < child.exec_ts + clock_skew_tolerance` (**RECOMMENDED** tolerance: 30 seconds). Causal ordering is primarily enforced by DAG structure, not timestamps.
4. **Acyclicity**: Following parent references **MUST NOT** lead back to the current ACT's `jti`. Implementations **MUST** enforce a maximum ancestor traversal limit (**RECOMMENDED**: 10,000 nodes).
5. **Capability Consistency**: Verify that `exec_act` matches one of the `action` values in the `cap` array from Phase 1.

7.2. Root Tasks and Fan-in

A root task has `pred = []`. A workflow **MAY** have multiple root tasks representing parallel branches with no shared predecessor.

Fan-in — a task with multiple parents — is expressed naturally:

```
{
  "pred": [
    "550e8400-e29b-41d4-a716-446655440001",
    "550e8400-e29b-41d4-a716-446655440002"
  ]
}
```

This indicates the current task depends on the completion of both referenced parent tasks, which **MAY** have been executed in parallel by different agents.

7.3. DAG vs Linear Delegation Chains

Several concurrent proposals for agent authorization model delegation as an ordered, linear chain of tokens or principals. Examples include the `actchain` claim of [I-D.oauth-transaction-tokens-for-agents], the Agentic JWT construction of [AgenticJWT], the AIP / Interaction-Bound Context Token (IBCT) model of [AIP-IBCT], and the delegation record defined in [I-D.helixar-hdp-agentic-delegation]. In each of these designs, the trail from the originator to the final executor is represented as an ordered array recording one predecessor per hop.

7.3.1. What Linear Chains Express Well

Linear chains are a natural fit for simple sequential delegation: agent A delegates to agent B, which delegates to agent C. The chain records the history of that single hand-off in order, and verifiers can walk from the current holder back to the originator without branching. For interactive user-to-agent-to-service flows, where each step has exactly one predecessor, a linear chain is both sufficient and compact.

7.3.2. Limitations of Linear Chains

Agentic workflows in practice are rarely purely linear. Planner agents dispatch parallel sub-tasks; synthesizer agents consume results from multiple independent branches; tool calls execute concurrently and their outputs are merged. A linear chain cannot faithfully represent the following common topologies:

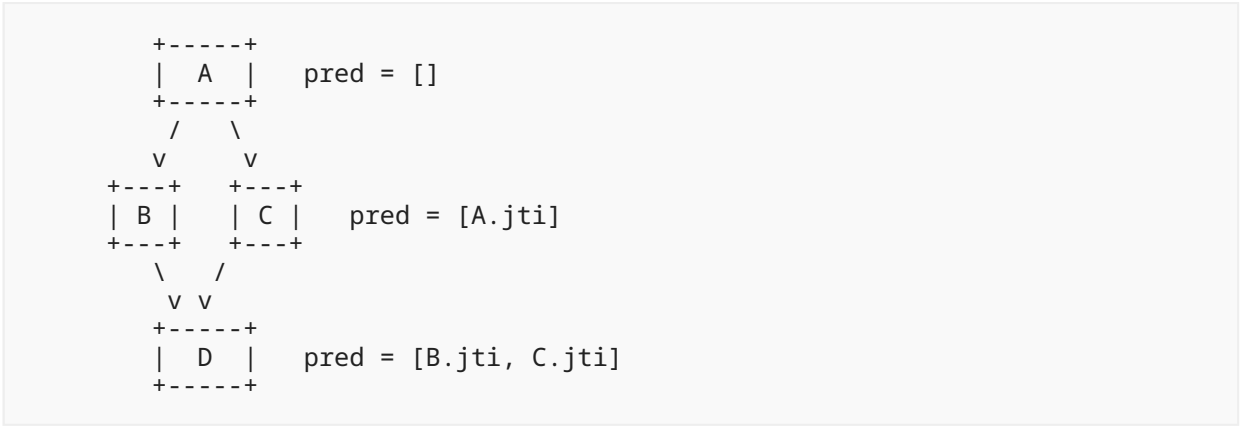
- **Fork**: A single task spawns multiple independent sub-tasks. A linear chain cannot express that two concurrent sub-executions share a common parent authorization but are otherwise independent; each sub-task would either omit its siblings or fabricate a false ordering between them.
- **Join (fan-in)**: A task whose output depends on results from several predecessors has no single prior hop. Linear chains cannot express multiple-parent relationships without either collapsing parallel branches into an arbitrary order or duplicating records.
- **Diamond dependencies**: A planner dispatches parallel work and later synthesizes the results. The synthesis step depends on every branch, and all branches depend on the same planner. This diamond shape requires a DAG; a linear chain forces the verifier to pick one branch and discard the others.
- **Cross-chain references**: When two independently authorized chains produce outputs that are later combined (e.g., a shared cache lookup and a fresh retrieval), linear chains force a single history and cannot record that the combined result has two distinct provenances.

7.3.3. ACT's DAG Approach

As specified in Section 4.3, the `pred` claim is an array of parent `jti` values rather than a single scalar. This allows an ACT to record:

- Zero parents (a root task, `pred = []`);
- Exactly one parent (a linear chain, equivalent to the single-predecessor designs referenced above);
- Multiple parents (fan-in from parallel branches); and
- Any acyclic shape that matches the actual execution structure.

The following example illustrates a diamond workflow. A research agent (A) dispatches a web-search agent (B) and a code-analysis agent (C) in parallel; both complete, and their outputs are combined by a writer agent (D):



A linear `actchain` representation cannot express that D depends on both B and C. At best, it can record one of the two parents and lose the other, or serialize B and C into a false sequential order.

7.3.4. Verifiability Implications

With a DAG representation, an auditor holding the set of Phase 2 ACTs for a workflow can reconstruct the full execution graph, not just one chain per final record. This matters for:

- **Debugging**: identifying which branch contributed an erroneous input to a downstream synthesis.
- **Compliance**: demonstrating that every input to a regulated decision was itself authorized, not only the most recent hop.
- **Tamper-evidence**: detecting that a branch has been omitted, since the surviving siblings' `pred` arrays name the missing predecessor by `jti`.

7.3.5. Interoperability with Linear-Chain Designs

ACT's DAG reduces to a linear chain in the degenerate case where every `pred` array has length zero or one. An implementation that requires linear-chain semantics **MAY** treat such ACTs as equivalent to `actchain`-style records and ignore the fork/join capability. The reverse reduction is not available: a linear-chain-only design cannot represent ACT DAG topologies without loss of information.

ACT therefore takes the linear chain as a strict subset of its model rather than as a competing approach. The DAG generalization is deliberate and is motivated by the concurrent, branching nature of real agentic executions rather than by any deficiency in the linear-chain designs for the sequential cases they target.

8. Verification Procedure

8.1. Authorization Phase Verification

A receiving agent **MUST** verify a Phase 1 ACT as follows:

1. Parse JWS Compact Serialization per [RFC7515].

2. Verify `typ` is "act+jwt".
3. Verify `alg` is in the verifier's algorithm allowlist. The allowlist **MUST NOT** include "none" or any symmetric algorithm.
4. Retrieve the public key for `kid` per the applicable trust tier (Section 5).
5. Verify the JWS signature.
6. Verify `exp` has not passed (with clock skew tolerance: **RECOMMENDED** maximum 5 minutes).
7. Verify `iat` is not unreasonably in the future (**RECOMMENDED**: no more than 30 seconds ahead).
8. Verify `aud` contains the verifier's own identifier.
9. Verify `iss` is a trusted agent identity per local policy.
10. Verify `sub` matches the verifier's own identifier (the agent is the intended recipient of this mandate).
11. Verify all required claims are present and well-formed.
12. Verify delegation chain (Section 6.3) if `del.chain` is non-empty.
13. Verify capabilities are within policy limits.

8.2. Execution Phase Verification

In addition to all Phase 1 verification steps, a verifier processing a Phase 2 ACT **MUST**:

1. Verify `exec_act` is present and matches an `action` in `cap`.
2. Verify `pred` is present and perform DAG validation (Section 7.1).
3. Verify `exec_ts` is present and is greater than or equal to `iat`. If `exec_ts` is after `exp`, implementations **SHOULD** log a warning but **MUST NOT** reject the record solely on this basis.
4. Verify `status` is present and has a valid value.
5. Verify the re-signature was produced by the `sub` agent (the executing agent), not the `iss` agent (the mandating agent). This is verified by checking that the `kid` in the Phase 2 JOSE header corresponds to the `sub` agent's public key.
6. If `inp_hash` or `out_hash` are present, verify them against locally available input/output data when possible.

9. Transport

9.1. HTTP Header Transport

This specification defines two HTTP header fields for ACT transport:

ACT-Mandate: Carries a Phase 1 ACT issued by an upstream agent or operator. Value is the JWS Compact Serialization of the ACT.

```
GET /api/safety-check HTTP/1.1
Host: safety-agent.example.com
ACT-Mandate: eyJhbGci...Phase1ACT...
```

ACT-Record: Carries a Phase 2 ACT from a predecessor agent, serving as evidence of completed prerequisites.

```
POST /api/downstream HTTP/1.1
Host: downstream-agent.example.com
ACT-Mandate: eyJhbGci...Phase1ACT...
ACT-Record: eyJhbGci...Phase2ACT...
```

Multiple `ACT-Record` header lines **MAY** be included when a task has multiple completed predecessors (DAG fan-in). If any single ACT-Record fails verification, the receiver **MUST** reject the entire request.

9.2. Non-HTTP Transports

For non-HTTP transports (MCP stdio, A2A message queues, AMQP, etc.), ACTs **SHOULD** be carried as a dedicated field in the transport's metadata envelope. The field name **SHOULD** be `act_mandate` for Phase 1 ACTs and `act_record` for Phase 2 ACTs. Implementations **MUST** use the JWS Compact Serialization form in all transports.

10. Audit Ledger Interface

Phase 2 ACTs **SHOULD** be submitted to an immutable audit ledger. A ledger is **RECOMMENDED** for regulated environments but is not required for basic ACT operation. This specification does not mandate a specific storage technology.

When an audit ledger is deployed, the implementation **MUST** provide:

1. **Append-only semantics**: Once an ACT is recorded, it **MUST NOT** be modified or deleted.
2. **Ordering**: A monotonically increasing sequence number per recorded ACT.
3. **Lookup**: Efficient retrieval by `jti` value.
4. **Integrity**: A cryptographic commitment scheme over recorded ACTs (e.g., hash-chaining, Merkle tree anchoring, or SCITT registration per [I-D.ietf-scitt-architecture]).

11. Security Considerations

11.1. Threat Model

ACT assumes an adversarial environment where:

- Individual agents may be compromised.
- Network paths may be intercepted (mitigated by transport security).
- Attackers may attempt to replay valid ACTs from prior interactions.
- Colluding agents may attempt to fabricate execution records.
- Agents may attempt privilege escalation via manipulated delegation chains.

ACT does NOT assume:

- A trusted central authority (by design).
- Synchronized clocks beyond the stated skew tolerance.
- Availability of external network services during verification.

11.2. Self-Assertion Limitation

Phase 2 ACTs are self-asserted: an executing agent signs its own execution record. A compromised agent with an intact private key can produce Phase 2 ACTs claiming arbitrary inputs, outputs, and action types, as long as the claimed `exec_act` matches an authorized capability.

This is a fundamental limitation of self-sovereign attestation. It is the same limitation affecting WIMSE ECT [I-D.nennemann-wimse-ect].

Mitigations:

- **Cross-agent corroboration**: A receiving agent that processes an ACT-Record as a prerequisite independently verifies that the claimed `out_hash` matches the data it actually received.
- **Ledger sequencing**: An append-only ledger with monotonic sequence numbers prevents retroactive insertion of fabricated records.
- **SCITT anchoring**: For high-assurance deployments, Phase 2 ACTs **SHOULD** be anchored to a SCITT Transparency Service, providing external witness that the record was submitted at a claimed time.

11.3. Key Compromise

If an agent's private key is compromised, an attacker can issue arbitrary Phase 1 mandates (impersonating the agent as an issuer) and fabricate Phase 2 records (impersonating the agent as an executor).

Key compromise response:

1. The compromised agent's identifier **MUST** be added to all verifiers' deny lists.
2. In Tier 2 (PKI) deployments, the certificate **MUST** be revoked via CRL or OCSP.
3. In Tier 3 (DID) deployments, the DID Document **MUST** be updated to revoke the compromised key.
4. In Tier 1 (pre-shared key) deployments, both parties **MUST** perform an out-of-band key rotation.

ACT chains that include records signed by a compromised key **MUST** be treated as potentially tainted from the point of compromise. Audit systems **MUST** flag all ACTs signed after the estimated compromise time.

11.4. Replay Attack Prevention

`jti` uniqueness within the applicable scope (workflow or global) provides replay detection. Verifiers **MUST** reject ACTs whose `jti` has already been seen and processed.

`exp` provides a time-bounded replay window. Verifiers **MUST** reject expired ACTs. The combination of `jti` and `exp` means that replay detection state only needs to be maintained for the duration of token lifetimes.

11.5. Equivocation

In standalone deployment (no audit ledger, no SCITT anchoring), ACT does NOT provide non-equivocation guarantees. A compromised agent can maintain two valid ACT chains — presenting Phase 2 records with different `out_hash` values to different verifiers — and both will pass independent verification.

Deployments claiming DORA [DORA] Article 10/11 compliance or EU AI Act [EUAIA] Article 12 compliance MUST use one of:

(a) A shared append-only audit ledger visible to all relevant parties, with cryptographic integrity (hash chaining or Merkle trees).

(b) SCITT anchoring [I-D.ietf-scitt-architecture] providing external Transparency Service receipts.

Standalone ACT provides tamper detection (a verifier can detect modification of a record it has seen) but not split-view prevention (a verifier cannot detect a different record shown to another verifier).

11.6. Privilege Escalation

Verifiers **MUST** check that each step in `del.chain` reduces or maintains (never increases) the capabilities relative to the preceding step. Implementations **MUST** reject ACTs where:

- `del.depth` exceeds `del.max_depth`.
- `cap` contains actions not present in any referenced parent ACT.
- Constraints in `cap` are less restrictive than those in the parent.

11.7. Denial of Service

ACT verification is more computationally expensive than standard JWT validation due to delegation chain verification and DAG traversal.

Mitigations:

- Reject ACTs larger than 64KB before parsing.
- Enforce maximum `del.chain` length (**RECOMMENDED**: 10 entries).
- Enforce maximum DAG ancestor traversal depth (**RECOMMENDED**: 10,000 nodes, Section 7.1).

- Cache verification results for recently seen `jti` values within the token lifetime window.

12. Privacy Considerations

ACT tokens and audit ledger records may contain information that identifies agents, organizations, or individuals. Implementations **SHOULD** apply data minimization principles:

- `task.created_by` **SHOULD** use a pseudonymous identifier rather than a personal email address or real name.
- `task.purpose` **SHOULD** use a controlled vocabulary code rather than free-text descriptions that may contain personal data.
- `del.chain` entries reveal organizational structure. Cross- organizational delegation chains **SHOULD** use Tier 3 (DID) identifiers that do not reveal organizational affiliation.
- `inp_hash` and `out_hash` are hashes of data, not the data itself, and do not constitute personal data under GDPR Article 4(1) provided the underlying data is not trivially reversible (e.g., hashes of very short strings).

For GDPR Article 17 (right to erasure) compliance, audit ledgers **SHOULD** store only ACT tokens (which contain hashes, not raw data) and **SHOULD** implement crypto-shredding for any associated encrypted payloads.

13. IANA Considerations

13.1. Media Type Registration

This document requests registration of the following media type:

- Type name: application
- Subtype name: act+jwt
- Required parameters: none
- Encoding considerations: binary (base64url-encoded JWT)
- Security considerations: See Section 11.
- Interoperability considerations: See Section 8.1.
- Specification: This document.

13.2. HTTP Header Field Registration

This document requests registration of the following HTTP header fields in the "Hypertext Transfer Protocol (HTTP) Field Name Registry":

- Header field name: ACT-Mandate
- Applicable protocol: HTTP
- Status: permanent
- Specification: This document, Section 9.1.

- Header field name: ACT-Record
- Applicable protocol: HTTP
- Status: permanent
- Specification: This document, Section 9.1.

13.3. JWT Claims Registration

This document requests registration of the following claims in the IANA "JSON Web Token Claims" registry:

Claim Name	Description	Reference
wid	Workflow identifier	This document
task	Task authorization context	This document
cap	Capabilities with constraints	This document
oversight	Human oversight requirements	This document
del	Delegation provenance chain	This document
exec_act	Executed action identifier	This document
pred	Predecessor task identifiers (DAG)	This document
inp_hash	SHA-256 hash of task input	This document
out_hash	SHA-256 hash of task output	This document
exec_ts	Actual execution timestamp	This document
status	Execution status	This document
err	Execution error context	This document

Table 1

14. References

14.1. Normative References

[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/rfc/rfc2119>.

[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/rfc/rfc7515>.

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

[RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <https://www.rfc-editor.org/rfc/rfc7518>.

[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/rfc/rfc7519>.

[RFC8037] Liusvaara, I., "CFRG Elliptic Curve Diffie-Hellman (ECDH) and Signatures in JSON Object Signing and Encryption (JOSE)", RFC 8037, DOI 10.17487/RFC8037, January 2017, <https://www.rfc-editor.org/rfc/rfc8037>.

[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/rfc/rfc8174>.

[RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/rfc/rfc9110>.

[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/rfc/rfc9562>.

14.2. Informative References

[A2A-SPEC] Google, "Agent2Agent (A2A) Protocol", <https://github.com/a2aproject/A2A>.

[AgenticJWT] "Agentic JWT: A JSON Web Token Profile for Delegated Agent Authorization", 2025, <https://arxiv.org/abs/2509.13597>.

[AIP-IBCT] S., P., "AIP: Agent Interaction Protocol with Interaction-Bound Context Tokens", March 2026, <https://arxiv.org/abs/2603.24775>.

[AUTOGEN] Microsoft, "AutoGen Documentation", <https://microsoft.github.io/autogen/>.

[CREWAI] CrewAI, "CrewAI Documentation", <https://docs.crewai.com/>.

[DID-KEY] D., L., "The did:key Method v0.7", 2021, <https://w3c-ccg.github.io/did-method-key/>.

[DID-WEB] O., S., "did:web Method Specification", 2022, <https://w3c-ccg.github.io/did-method-web/>.

[DORA] European Parliament, "Digital Operational Resilience Act (DORA), Regulation (EU) 2022/2554", 2022, <https://eur-lex.europa.eu/eli/reg/2022/2554/oj>.

[EUAIA] European Parliament, "EU Artificial Intelligence Act, Regulation (EU) 2024/1689", 2024, <https://eur-lex.europa.eu/eli/reg/2024/1689/oj>.

[I-D.aap-oauth-profile] A., C., "Agent Authorization Profile (AAP) for OAuth 2.0", Work in Progress, Internet-Draft, draft-aap-oauth-profile-01, February 2026, <https://datatracker.ietf.org/doc/draft-aap-oauth-profile/>.

[I-D.emirdag-scitt-ai-agent-execution] Emirdag, "A SCITT Profile for AI Agent Execution Records", Work in Progress, Internet-Draft, draft-emirdag-scitt-ai-agent-execution-00, April 2026, <https://datatracker.ietf.org/doc/draft-emirdag-scitt-ai-agent-execution/>.

[I-D.helixar-hdp-agentic-delegation] Helixar, "Helixar Delegation Protocol (HDP) for Agentic Delegation", Work in Progress, Internet-Draft, draft-helixar-hdp-agentic-delegation-00, 2026, <https://datatracker.ietf.org/doc/draft-helixar-hdp-agentic-delegation/>.

[I-D.ietf-scitt-architecture] Birkholz, H., Delignat-Lavaud, A., Fournet, C., Deshpande, Y., and S. Lasker, "An Architecture for Trustworthy and Transparent Digital Supply Chains", Work in Progress, Internet-Draft, draft-ietf-scitt-architecture-22, 10 October 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-scitt-architecture-22>.

[I-D.nennemann-wimse-ect] Nennemann, C., "Execution Context Tokens for Distributed Agentic Workflows", Work in Progress, Internet-Draft, draft-nennemann-wimse-ect-02, 2026, <https://datatracker.ietf.org/doc/draft-nennemann-wimse-ect/>.

[I-D.oauth-transaction-tokens-for-agents] G., F., "Transaction Tokens for Agentic AI Systems", Work in Progress, Internet-Draft, draft-oauth-transaction-tokens-for-agents-06, 2026, <https://datatracker.ietf.org/doc/draft-oauth-transaction-tokens-for-agents/>.

[IEC62304] IEC, "Medical device software — Software life cycle processes, IEC 62304:2006+AMD1:2015", 2015, <https://www.iec.ch/>.

[LANGGRAPH] LangChain, "LangGraph Documentation", <https://langchain-ai.github.io/langgraph/>.

[MCP-SPEC] "Model Context Protocol Specification", 25 November 2025, <https://modelcontextprotocol.io/specification/2025-11-25>.

[OPENAI-AGENTS-SDK] OpenAI, "OpenAI Agents SDK", <https://openai.github.io/openai-agents-python/>.

[RFC7009] Lodderstedt, T., Ed., Dronia, S., and M. Scurtescu, "OAuth 2.0 Token Revocation", RFC 7009, DOI 10.17487/RFC7009, August 2013, <https://www.rfc-editor.org/rfc/rfc7009>.

[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/rfc/rfc8693>.

[SentinelAgent] Patil, "SentinelAgent: A Formal Delegation Chain Calculus for Verifiable Agent Authorization", April 2026, <https://arxiv.org/abs/2604.02767>.

[W3C-DID] M., S., "Decentralized Identifiers (DIDs) v1.0", July 2022, <https://www.w3.org/TR/did-core/>.

Appendix A: Complete JSON Schema

The normative JSON Schema for ACT Phase 1 and Phase 2 tokens is available at [TODO: reference implementation repository].

Appendix B: Test Vectors

B.1. Valid Phase 1 ACT — Root Mandate (Tier 1, Pre-Shared Key)

```
[TODO: include encoded test vector with signing key, payload,
and expected JWS Compact Serialization]
```

B.2. Valid Phase 2 ACT — Completed Execution

```
[TODO: include encoded test vector demonstrating Phase 1 -> Phase 2
transition with re-signature by target agent]
```

B.3. Valid Phase 2 ACT — Fan-in (Multiple Parents)

```
[TODO: demonstrate pred with two predecessor jti values from parallel
workflow branches]
```

B.4. Invalid ACT — Delegation Depth Exceeded

```
[TODO: demonstrate del.depth > del.max_depth rejection]
```

B.5. Invalid ACT — Capability Escalation

```
[TODO: demonstrate rejection when delegated cap contains action
not present in parent ACT]
```

B.6. Invalid ACT — exec_act Mismatch

```
[TODO: demonstrate rejection when exec_act does not match any
cap.action in the Phase 1 claims]
```

Appendix C: Deployment Scenarios

C.1. Minimal Deployment (Zero Infrastructure)

Two organizations exchange pre-shared public keys via secure email. Each agent signs Phase 1 mandates and Phase 2 records with its Ed25519 key. No ledger, no external services. Suitable for development and low-risk workflows.

Limitation: No non-equivocation (Section 11.5).

C.2. Regulated Deployment with Hash-Chained Ledger

Phase 2 ACTs are submitted to a shared append-only ledger with hash-chaining. Each recorded ACT extends a cryptographic chain, providing tamper evidence for each ACT and the chain as a whole. The ledger is shared between all regulated parties participating in the workflow. Suitable for DORA compliance.

C.3. High-Assurance Cross-Organizational Deployment

Phase 2 ACTs are anchored to a SCITT Transparency Service. SCITT receipts are attached to the audit record as non-equivocation proofs. DID-based agent identities (Tier 3) enable self-sovereign key management without shared CA infrastructure.

C.4. WIMSE Environment Integration

In environments where WIMSE is already deployed, ACT-Mandate and ACT-Record headers are carried alongside the WIMSE Workload-Identity header. The ECT and ACT serve different purposes: the ECT records workload-level execution in WIMSE terms; the ACT records the authorization provenance and capability constraints that governed the action.

Acknowledgments

The author thanks the IETF WIMSE, OAuth, and SCITT working groups for foundational work on workload identity, delegated authorization, and transparent supply chain records that informs this specification.

Author's Address

Christian Nennemann
Independent Researcher
Email: ietf@nennemann.de