TBD
Internet-Draft
Intended status: Standards Track
Expires: 31 October 2026

D. Hardt
Hell

29 April 2026

AAuth Protocol
draft-hardt-oauth-aauth-protocol-00

Abstract

This document defines the AAuth authorization protocol for agent-to-resource authorization and identity claim retrieval. The protocol supports four resource access modes — identity-based, resource-managed (two-party), PS-managed (three-party), and federated (four-party) — with agent governance as an orthogonal layer. It builds on the HTTP Signature Keys specification ([I-D.hardt-httpbis-signature-key]) for HTTP Message Signatures and key discovery.

Discussion Venues

Note: This section is to be removed before publishing as an RFC.

This document is part of the AAuth specification family. Source for this draft and an issue tracker can be found at https://github.com/dickhardt/AAuth (https://github.com/dickhardt/AAuth).

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 31 October 2026.

Table of Contents

1. Introduction

1.1. HTTP Clients Need Their Own Identity

In OAuth 2.0 [RFC6749] and OpenID Connect [OpenID.Core], the client has no independent identity. Client identifiers are issued by each authorization server or OpenID provider — a client_id at Google is meaningless at GitHub. The client’s identity exists only in the context of each server it has pre-registered with. This made sense when the web had a manageable number of integrations and a human developer could visit each portal to register.

API keys are the same model pushed further: a shared secret issued by a service, copied to the client, and used as a bearer credential. The problem is that any secret that must be copied to where the workload runs will eventually be copied somewhere it shouldn’t be.

SPIFFE and WIMSE brought workload identity to enterprise infrastructure — a workload can prove who it is without shared secrets. But these operate within a single enterprise's trust domain. They don't help an agent that needs to access resources across organizational boundaries, or a developer's tool that runs outside any enterprise platform.

AAuth starts from this premise: every agent has its own cryptographic identity. An agent identifier (aauth:local@domain) is bound to a signing key, published at a well-known URL, and verifiable by any party — no pre-registration, no shared secrets, no dependency on a particular server. At its simplest, an agent signs a request and a resource decides what to do based on who the agent is. This identity-based access replaces API keys and is the foundation that authorization, governance, and federation build on incrementally.

1.2. Agents Are Different

Traditional software knows at build time what services it will call and what permissions it needs. Registration, key provisioning, and scope configuration happen before the first request. This works when the set of integrations is fixed and known in advance.

Agents don't work this way. They discover resources at runtime. They execute long-running tasks that span multiple services across trust domains. They need to explain what they're doing and why. They need authorization decisions mid-task, long after the user set them in motion. A protocol designed for pre-registered clients with fixed integrations cannot serve agents that discover their needs as they go.

1.3. What AAuth Provides

* *Agent identity without pre-registration*: A domain, static metadata, and a JWKS establish identity with no portal, no bilateral agreement, no shared secret.
* *Per-instance identity*: Each agent instance gets its own identifier (aauth:local@domain) and signing key.
* *Proof-of-possession on every request*: HTTP Message Signatures ([RFC9421]) bind every request to the agent's key — a stolen token is useless without the private key.
* *Two-party mode with first-call registration*: An agent calls a resource it has never contacted before; the resource returns AAuth-Requirement; a browser interaction handles account creation, payment, and consent. The first API call is the registration.
* *Tool-call governance*: A person server (PS) represents the user and manages what tools the agent can call, providing permission and audit for tool use — no resource involved.

* *Missions*: Optional scoped authorization contexts that span multiple resources. The agent proposes what it intends to do in natural language; the person server provides full context — mission, history, justification — to the appropriate decision-maker (human or AI); every resource access is evaluated in context. Missions enable governance over decisions that cannot be reduced to predefined machine-evaluable rules.
* *Cross-domain federation*: The PS federates with access servers (AS) — the policy engines that guard resources — to enable access across trust domains without the agent needing to know about each one.
* *Clarification chat*: Users can ask questions during consent; agents can explain or adjust their requests.
* *Progressive adoption*: Each party can adopt independently; modes build on each other.

1.4. What AAuth Does Not Do

* Does not require centralized identity providers — agents publish their own identity
* Does not use shared secrets or bearer tokens — every credential is bound to a signing key and useless without it
* Does not require coordination to adopt — each party adds support independently

1.5. Relationship to Existing Standards

AAuth builds on existing standards and design patterns:

* *OpenID Connect vocabulary*: AAuth reuses OpenID Connect scope values, identity claims, and enterprise extensions ([OpenID.Enterprise]), lowering the adoption barrier for identity-aware resources.
* *Well-known metadata and key discovery*: Servers publish metadata at well-known URLs ([RFC8615]) and signing keys via JWKS endpoints, following the pattern established by OAuth Authorization Server Metadata ([RFC8414]) and OpenID Connect Discovery ([OpenID.Core]).
* *HTTP Message Signatures*: All requests are signed with HTTP Message Signatures ([RFC9421]) using keys bound to tokens conveyed via the Signature-Key header ([I-D.hardt-httpbis-signature-key]), providing proof-of-possession, identity, and message integrity on every call.

The HTTP Signature Keys specification ([I-D.hardt-httpbis-signature-key]) defines how signing keys are bound to JWTs and discovered via well-known metadata, and how agents present cryptographic identity using HTTP Message Signatures

([RFC9421]). This specification defines the AAuth-Requirement, AAuth-Access, and AAuth-Capabilities headers, and the authorization protocol across four resource access modes.

Because agent identity is independent and self-contained, AAuth is designed for incremental adoption — each party can add support independently, and rollout does not need to be coordinated. A resource that verifies an agent's signature can manage access by identity alone, with no other infrastructure. When a resource manages its own authorization — via interaction, consent, or existing infrastructure — it operates in resource-managed access (two-party). Issuing resource tokens to the agent's person server enables PS-managed access (three-party), where auth tokens carry user identity, organization membership, and group information. Deploying an access server enables federated access (four-party) with cross-domain policy enforcement. Agent governance — missions, permissions, audit — is an orthogonal layer that any agent with a PS can add, from a simple prompt to full autonomous agent oversight. See Section 13 for details.

2. Conventions and Definitions

{::boilerplate bcp14-tagged}

In HTTP examples throughout this document, line breaks and indentation are added for readability. Actual HTTP messages do not contain these extra line breaks.

3. Terminology

Parties:

* *Person*: A user or organization — the legal person — on whose behalf an agent acts and who is accountable for the agent's actions.
* *Agent*: An HTTP client ([RFC9110], Section 3.5) acting on behalf of a person. Identified by an agent identifier URIs using the aauth scheme, of the form aauth:local@domain Section 5.1. An agent MAY have a person server, declared via the ps claim in the agent token.
* *Agent Server*: A server that manages agent identity and issues agent tokens to agents. Trusted by the person to issue agent tokens only to authorized agents. Identified by an HTTPS URL Section 12.9.1 and publishes metadata at /.well-known/aauth-agent.json.
* *Resource*: A server that requires authentication and/or authorization to protect access to its APIs and data. A resource MAY enforce access policy itself or delegate policy evaluation to

an access server. Identified by an HTTPS URL Section 12.9.1 and publishes metadata at /.well-known/aauth-resource.json. A mission-aware resource includes the mission object from the AAuth-Mission header in the resource tokens it issues.
* *Person Server (PS)*: A server that represents the person to the rest of the protocol. The person chooses their PS; it is not imposed by any other party. The PS manages missions, handles consent, asserts user identity, and brokers authorization on behalf of agents. Identified by an HTTPS URL Section 12.9.1 and publishes metadata at /.well-known/aauth-person.json.
* *Access Server (AS)*: A policy engine that evaluates token requests, applies resource policy, and issues auth tokens on behalf of a resource. Identified by an HTTPS URL Section 12.9.1 and publishes metadata at /.well-known/aauth-access.json.

Tokens:

* *Agent Token*: Issued by an agent server to establish the agent's identity. MAY declare the agent's person server Section 5.2.
* *Resource Token*: Issued by a resource to describe the access the agent needs Section 6.
* *Auth Token*: Issued by a PS or AS to grant an agent access to a resource, containing identity claims and/or authorized scopes Section 9.4.

Protocol concepts:

* *Mission*: A scoped authorization context for agent governance Section 8. Required when the person's PS requires governance over the agent's actions. A mission is a JSON object containing structured fields (approver, agent, approved_at, approved tools) and a Markdown description. Identified by the PS and SHA-256 hash of the mission JSON (s256). Missions are proposed by agents and approved by the PS and person.
* *Mission Log*: The ordered record of all agentPS interactions within a mission — token requests, permission requests, audit records, interaction requests, and clarification chats. The PS maintains the log and uses it to evaluate whether each new request is consistent with the mission's intent Section 8.3.
* *HTTP Sig*: An HTTP Message Signature ([RFC9421]) created per the AAuth HTTP Message Signatures profile defined in this specification Section 12.7, using a key conveyed via the Signature-Key header ([I-D.hardt-httpbis-signature-key]).
* *Markdown*: AAuth uses Markdown ([CommonMark]) as the human-readable content format for mission descriptions, justifications, clarifications, and scope descriptions. Implementations MUST sanitize Markdown before rendering to users.

* *Interaction*: User authentication, consent, or other action at an interaction endpoint Section 7.2. Triggered when a server returns 202 Accepted with requirement=interaction.
* *Justification*: A Markdown string provided by the agent declaring why access is needed, presented to the user by the PS during consent Section 7.1.
* *Clarification*: A Markdown string containing a question posed to the agent by the user during consent via the PS Section 7.3. The agent may respond with an explanation or an updated request.

4. Protocol Overview

All AAuth tokens are JWTs verified using a JWK retrieved from the jwks_uri in the issuer's well-known metadata, binding each token to the server that issued it.

AAuth has two dimensions: *resource access modes* and *agent governance*. Resource access modes define how an agent gets authorized at a resource. Agent governance — missions, permissions, audit — is an orthogonal layer that any agent with a person server can add, independent of which access mode the resource supports.

4.1. Resource Access Modes

AAuth supports four resource access modes, each adding parties and capabilities. The protocol works in every mode — adoption does not require coordination between parties.

Mode	Parties	Description
Identity-based access	Agent Resource	Resource verifies agent's signed identity and applies its own access control
Resource-managed access (two-party)	Agent Resource	Resource manages authorization with interaction, consent, or existing auth infrastructure
PS-managed access (three-party)	Agent Resource PS	Resource issues resource token to PS; PS issues auth token
Federated access (four-party)	Agent Resource PS AS	Resource has its own access server; PS federates with AS

Table 1

The following diagram shows all parties and their relationships. Not all parties or relationships are present in every mode.

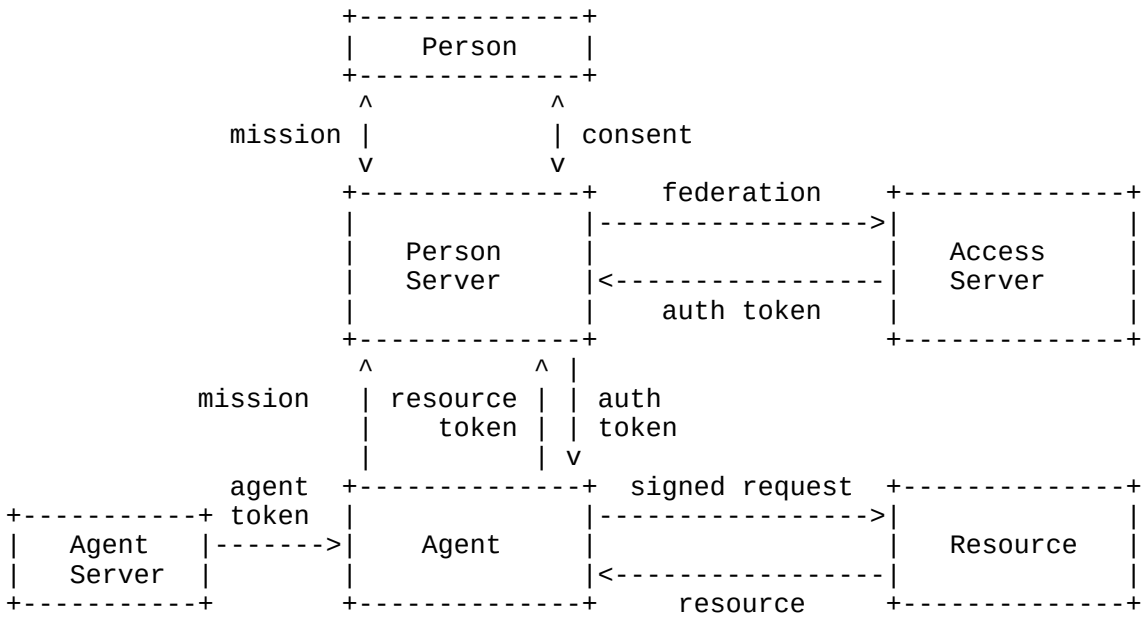


Figure 1: Protocol Parties and Relationships

* *Agent Server → Agent*: Issues an agent token binding the agent's signing key to its identity.
* *Agent Resource*: Agent sends signed requests; resource returns responses. In PS-managed and federated modes, the resource also returns resource tokens at its authorization endpoint.
* *Agent PS*: Agent sends resource tokens to obtain auth tokens. With governance, agent also creates missions and requests permissions.
* *PS AS*: Federation (four-party only). The PS sends the resource token to the AS; the AS returns an auth token.
* *Person PS*: Mission approval and consent for resource access.

Detailed end-to-end flows are in Appendix A. The following subsections describe each mode.

4.1.1. Identity-Based Access

The agent signs requests with its agent token Section 5.2. The resource verifies the agent's identity via HTTP signatures and applies its own access control policy — granting or denying based on who the agent is. This replaces API keys with cryptographic identity. No authorization flow, no tokens beyond the agent token.

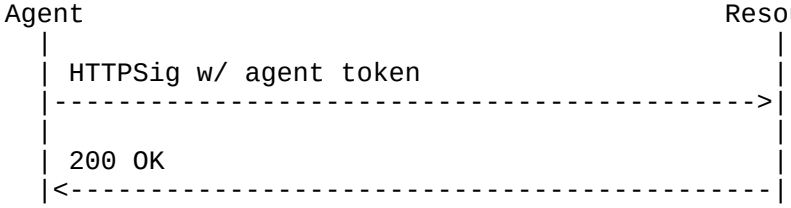


Figure 2: Identity-Based Access

4.1.2. Resource-Managed Access (Two-Party)

The resource handles authorization itself — via interaction Section 7.2, existing OAuth/OIDC infrastructure, or internal policy. After authorization, the resource MAY return an AAuth-Access header Section 6.3 with an opaque access token for subsequent calls.

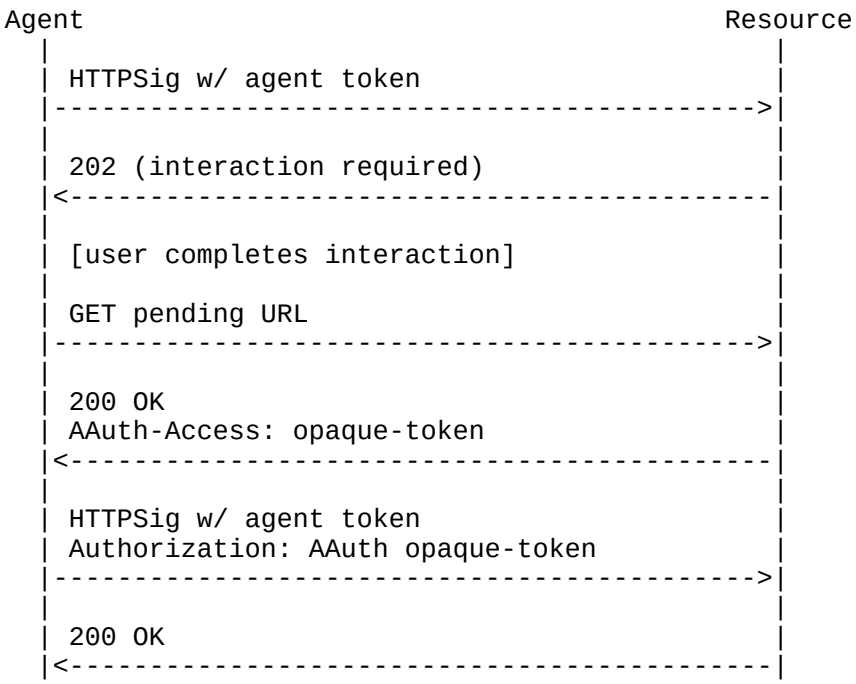


Figure 3: Resource-Managed Access (Two-Party)

4.1.3. PS-Managed Access (Three-Party)

The resource discovers the agent's PS from the ps claim in the agent token and issues a resource token Section 6 with aud = PS URL. The agent obtains the resource token either by calling the resource's authorization_endpoint (if published in resource metadata) or by receiving a 401 challenge with requirement=auth-token when calling the resource directly Section 6.5. The agent sends the resource token to the PS's token endpoint Section 7.1, and the PS issues an auth token Section 9.4 directly. The auth token can carry user identity, organization membership, and group information — enabling the resource to get rich authorization context from the PS without building its own identity infrastructure.

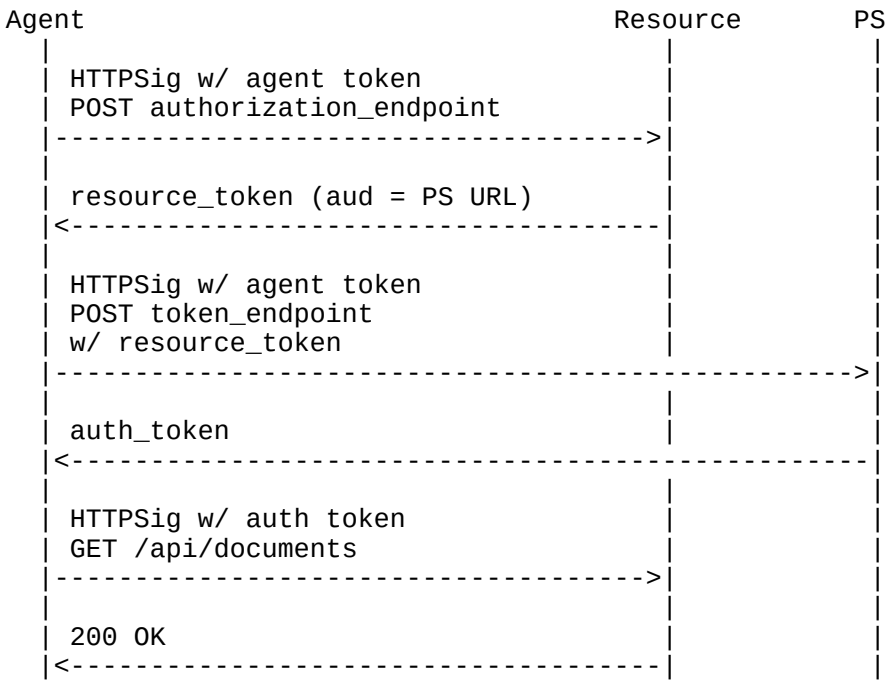


Figure 4: PS-Managed Access (Three-Party)

4.1.4. Federated Access (Four-Party)

The resource has its own access server. The resource issues a resource token Section 6 with aud = AS URL — either via its authorization_endpoint or a 401 challenge Section 6.5. The PS federates with the AS Section 9.3 to obtain the auth token Section 9.4.

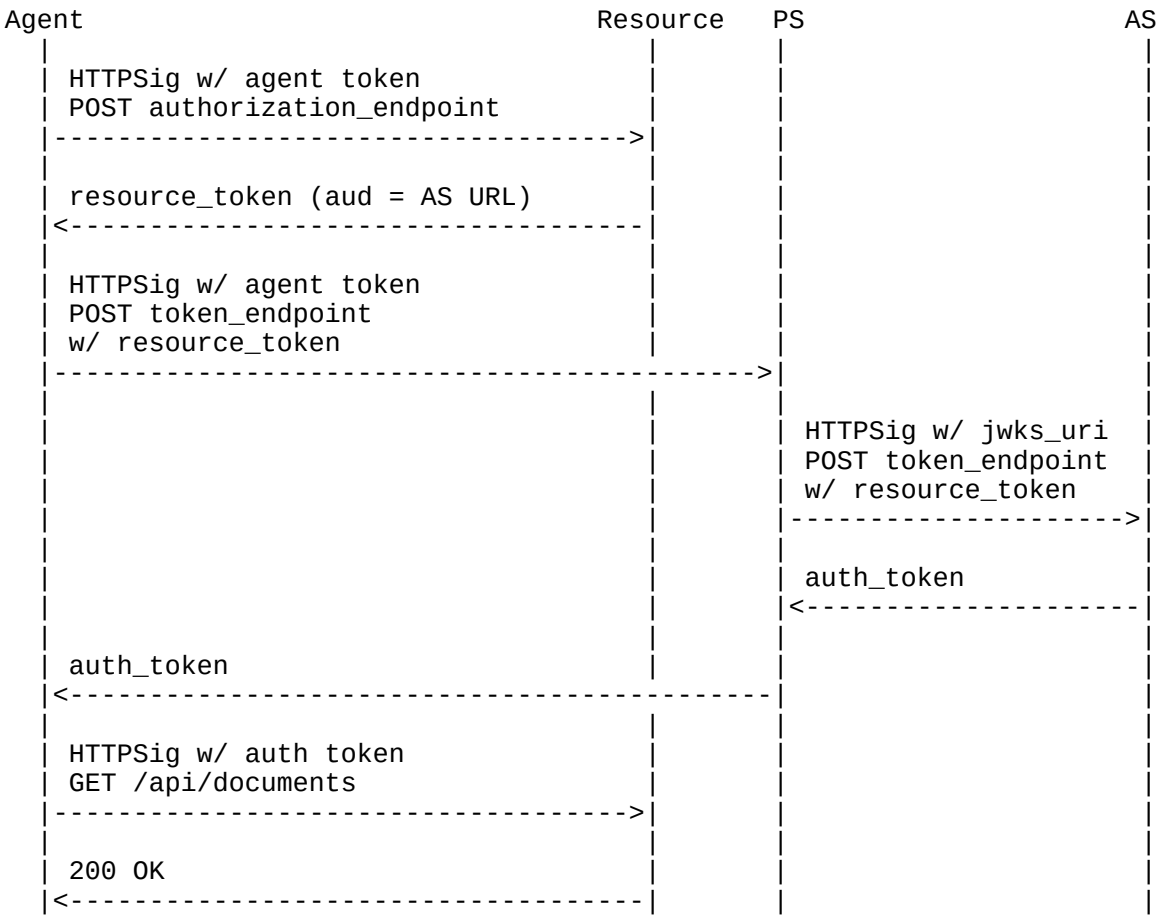


Figure 5: Federated Access (Four-Party)

4.1.5. Agent Server as Resource

An agent server MAY also act as a resource — publishing metadata at /.well-known/aauth-resource.json and issuing resource tokens. This enables the agent to obtain auth tokens from its PS for the agent server's own services or infrastructure, using the standard resource token flow. How the agent obtains the resource token from the agent server is out of scope of this specification. No mission is required.

4.2. Agent Governance

Agent governance is orthogonal to resource access modes. Any agent with a person server (ps claim in agent token) can use the PS for governance, regardless of which access modes the resources it accesses support.

4.2.1. Missions

When the person's PS requires governance over the agent's actions, the agent creates a mission — a Markdown description of what it intends to accomplish. The PS and user review, clarify, and approve the mission. The approved mission is immutable — bound by its s256 hash. Missions evolve through the *mission log* Section 8.3: the ordered record of all agentPS interactions within the mission. Missions are not required for all PS interactions — an agent can get auth tokens without a mission. See Section 8 for normative requirements.

4.2.1.1. Mission Creation

The agent proposes a mission at the PS. The PS and user may clarify and refine before approving.

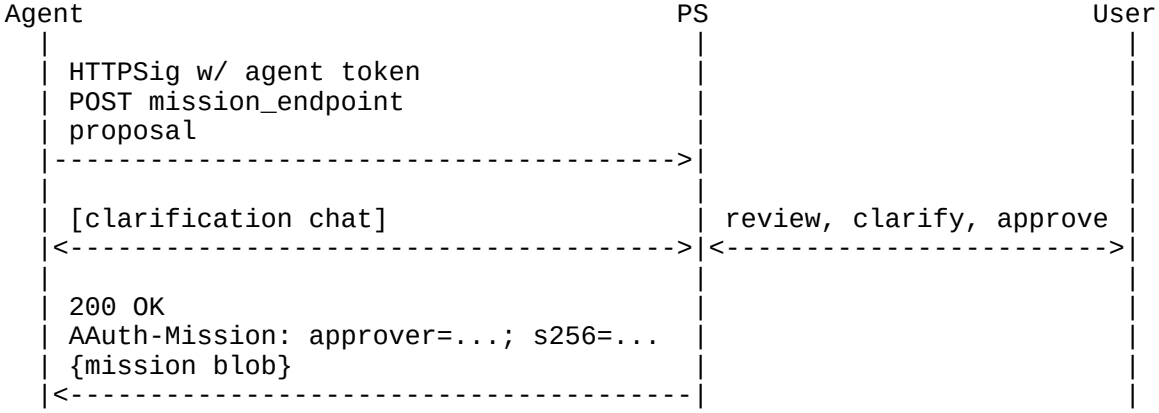


Figure 6: Mission Creation and Approval

4.2.1.2. Mission Context at Resources

The agent includes the AAuth-Mission header when sending requests to resources, unless the mission is already conveyed in an auth token. The resource includes the mission object in the resource token it issues:

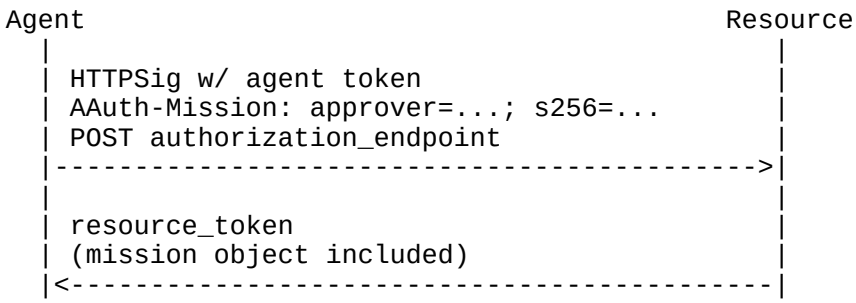


Figure 7: Mission Context at Resource

4.2.1.3. Mission Completion

When the agent believes the mission is complete, it proposes completion via the interaction endpoint with a summary. The PS presents the summary to the user. The user either accepts (mission terminates) or responds with follow-up questions (mission continues).

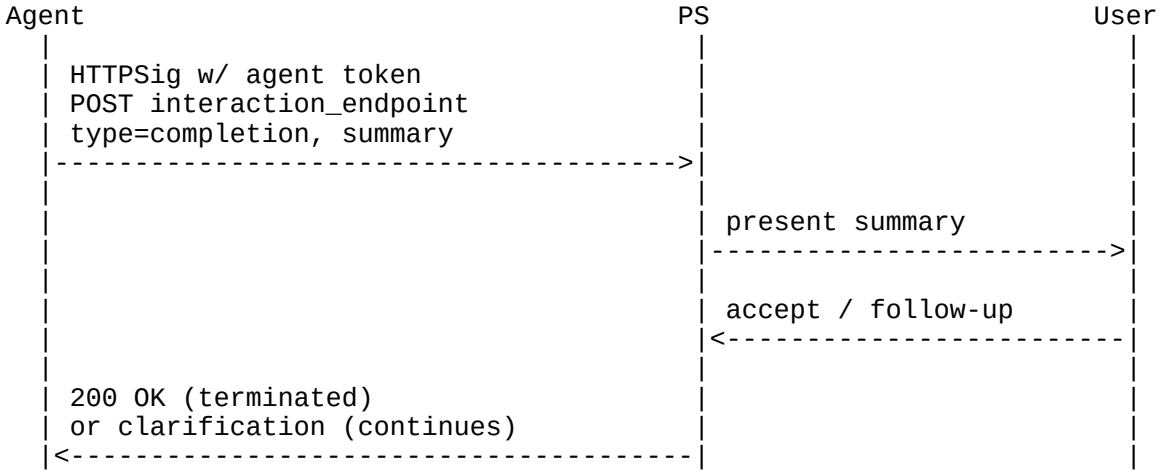


Figure 8: Mission Completion

4.2.2. PS Governance Endpoints

The PS provides three governance endpoints. The *permission* Section 7.4 and *interaction* Section 7.6 endpoints work with or without a mission. The *audit* endpoint Section 7.5 requires a mission.

* *Permission endpoint*: Request permission for actions not governed by a remote resource — tool calls, file writes, sending messages.

* *Audit endpoint*: Log actions performed, providing the PS with a complete record for the mission log.
* *Interaction endpoint*: Reach the user through the PS — relay interactions, ask questions, forward payment approvals, or propose mission completion.

4.3. Obtaining an Agent Token

The agent obtains an agent token from its agent server. The agent generates a signing key pair, proves its identity to the agent server through a platform-specific mechanism, and receives an agent token binding the signing key to the agent's identifier. The agent token MAY include a ps claim identifying the agent's person server. Agent token structure and normative requirements are defined in Section 5.2. Acquisition is platform-dependent; see [I-D.hardt-aauth-bootstrap] for common patterns.

4.4. Bootstrapping

Before protocol flows begin, each entity must be established with its identity, keys, and relationships. The requirements build incrementally.

All modes:

* Agent obtains an agent token from its agent server, binding its signing key to its identifier (aauth:local@domain). See [I-D.hardt-aauth-bootstrap].
* Agent servers publish metadata at /.well-known/aauth-agent.json Section 12.10.1.

Resource-managed access (two-party) and above:

* Resources MAY publish metadata at /.well-known/aauth-resource.json Section 12.10.4. Resources that do not publish metadata can still issue resource tokens and interaction requirements via 401 responses.

PS-managed access (three-party) and above:

* The agent's agent token includes the ps claim identifying its person server. This is configured during agent setup (e.g., set by the agent server or chosen by the person deploying the agent).
* The PS maintains the association between an agent and its person. This association is typically established when the person first authorizes the agent at the PS via the interaction flow. An organization administrator may also pre-authorize agents for the organization.

* The PS MAY establish a direct communication channel with the user (e.g., email, push notification, or messaging) to support out-of-band authorization, approval notifications, and revocation alerts.
* Person servers publish metadata at /.well-known/aauth-person.json Section 12.10.2.
* The resource discovers the agent's PS from the ps claim in the agent token and issues resource tokens with aud = PS URL.

Federated access (four-party):

* Access servers publish metadata at /.well-known/aauth-access.json Section 12.10.3.
* The resource issues resource tokens with aud = AS URL.
* The PS and the resource's AS must have a trust relationship before the AS will issue auth tokens. This trust may be pre-established (through a business relationship) or established dynamically through the AS's token endpoint responses — interaction, payment, or claims. When an organization controls both the PS and AS, trust is implicit. See Section 9.3 for details.

5. Agent Identity

This section defines agent identity — how agents are identified and how that identity is bound to signing keys via agent tokens. Agent identity is the foundation of AAuth: every signed request an agent makes carries its agent token, enabling any party to verify who the agent is and that the request was signed by the key bound to that identity.

5.1. Agent Identifiers

Agent identifiers are URIs using the aauth scheme, of the form aauth:local@domain where domain is the agent server's domain. The local part MUST consist of lowercase ASCII letters (a-z), digits (0-9), hyphen (-), underscore (_), plus (+), and period (.). The local part MUST NOT be empty and MUST NOT exceed 255 characters. The domain part MUST be a valid domain name conforming to the server identifier requirements Section 12.9.1 (without scheme).

Valid agent identifiers:

* aauth:assistant-v2@agent.example
* aauth:cli+instance.1@tools.example

Invalid agent identifiers:

* My Agent@agent.example (uppercase letters and space in local part)
* @agent.example (empty local part)

* agent@http://agent.example (domain includes scheme)

Implementations MUST perform exact string comparison on agent identifiers (case-sensitive).

5.2. Agent Token

5.2.1. Agent Token Acquisition

An agent MUST obtain an agent token from its agent server before participating in the AAuth protocol. The acquisition process follows these steps:

1. The agent generates an ephemeral signing key pair (EdDSA is RECOMMENDED).
2. The agent proves its identity to the agent server through a platform-specific mechanism.
3. The agent server verifies the agent's identity and issues an agent token binding the agent's ephemeral public key to the agent's identifier.

The mechanism for proving identity is platform-dependent. See [I-D.hardt-aauth-bootstrap] for common patterns including self-hosted agents, browser-based applications, mobile applications, and B2B SaaS agents.

5.2.2. Agent Token Structure

An agent token is a JWT with typ: aa-agent+jwt containing:

Header: - alg: Signing algorithm. EdDSA is RECOMMENDED. Implementations MUST NOT accept none. - typ: aa-agent+jwt - kid: Key identifier

Required payload claims: - iss: Agent server URL - dwk: aauth-agent.json — the well-known metadata document name for key discovery ([I-D.hardt-httpbis-signature-key]) - sub: Agent identifier (stable across key rotations) - jti: Unique token identifier for replay detection, audit, and revocation - cnf: Confirmation claim ([RFC7800]) with jwk containing the agent's public key - iat: Issued at timestamp - exp: Expiration timestamp. Agent tokens SHOULD NOT have a lifetime exceeding 24 hours.

Optional payload claims: - ps: The HTTPS URL of the agent's person server. Configured per agent instance. When present, resources can discover the agent's PS from the agent token. This claim is distinct from iss (which identifies the agent server that issued the token).

Agent servers MAY include additional claims in the agent token. Companion specifications may define additional claims for use by PSes or ASes in policy evaluation — for example, software attestation, platform integrity, secure enclave status, workload identity assertions, or software publisher identity. PSes and ASes MUST ignore unrecognized claims.

5.2.3. Agent Token Usage

Agents present agent tokens via the Signature-Key header ([I-D.hardt-httpbis-signature-key]) using scheme=jwt:

```
Signature-Key: sig=jwt;
    jwt="eyJhbGciOiJFZERTQSIsInR5cCI6Im..."
```

5.2.4. Agent Token Verification

Verify the agent token per [RFC7515] and [RFC7519]:

1. Decode the JWT header. Verify typ is aa-agent+jwt.
2. Verify dwk is aauth-agent.json. Discover the issuer’s JWKS via {iss}/.well-known/{dwk} per the HTTP Signature Keys specification ([I-D.hardt-httpbis-signature-key]). Locate the key matching the JWT header kid and verify the JWT signature.
3. Verify exp is in the future and iat is not in the future.
4. Verify iss is a valid HTTPS URL conforming to the Server Identifier requirements.
5. Verify cnf.jwk matches the key used to sign the HTTP request.
6. If ps is present, verify it is a valid HTTPS URL conforming to the Server Identifier requirements.

6. Resource Access and Resource Tokens

This section defines how agents request access to resources and how resources issue resource tokens.

A resource token can be returned in two ways:

1. *Authorization endpoint*: The agent proactively requests access at the resource’s authorization_endpoint. The resource responds with a resource token.
2. *AAuth-Requirement challenge*: The agent calls a resource endpoint directly. If the agent lacks sufficient authorization, the resource returns 401 with an AAuth-Requirement header containing a resource token Section 6.5.

A resource MAY return a 401 with AAuth-Requirement even when the agent presents a valid auth token — for example, when the endpoint requires additional scopes or a different authorization context beyond what the current auth token grants (nested authorization).

A resource token is a signed JWT that cryptographically binds the resource's identity, the agent's identity, and the requested scope. The resource sets the token's audience based on its configuration:

* If the resource has its own AS: aud = AS URL (four-party)
* If the resource has no AS but the agent has a PS (ps claim in agent token): aud = PS URL (three-party)
* If neither: the resource handles authorization itself — via an interaction response Section 7.2 or internal policy — and MAY return an AAuth-Access header Section 6.3

A resource MAY always handle authorization itself, regardless of whether the agent has a PS.

6.1. Authorization Endpoint Request

A resource MAY publish an authorization_endpoint in its metadata. The agent sends a signed POST to the authorization endpoint. The resource reads the agent token from the Signature-Key header and determines how to respond — it may return a resource token, handle authorization itself, or both.

Request parameters:

* scope (REQUIRED): A space-separated string of scope values the agent is requesting.

```
POST /authorize HTTP/1.1
Host: resource.example
Content-Type: application/json
Signature-Input: sig=("@method" "@authority"
    "@path" "signature-key");created=1730217600
Signature: sig=:...signature bytes...:
Signature-Key: sig=jwt;jwt="eyJhbGc..."

{
  "scope": "data.read data.write"
}
```

When the agent is operating in a mission context, it includes the AAuth-Mission header and adds aauth-mission to the signed components:

```
POST /authorize HTTP/1.1
Host: resource.example
Content-Type: application/json
AAuth-Mission:
    approver="https://ps.example";
    s256="dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk"
Signature-Input: sig=("@method" "@authority"
    "@path" "signature-key"
    "aauth-mission");created=1730217600
Signature: sig=:...signature bytes...:
Signature-Key: sig=jwt;jwt="eyJhbGc..."

{
  "scope": "data.read data.write"
}
```

6.2. Authorization Endpoint Responses

The resource can handle authorization itself, or it can issue a resource token when the resource has an AS or the agent token includes a ps claim.

6.2.1. Response without Resource Token

The resource handles authorization itself. It evaluates the request and returns a deferred response if user interaction is needed:

```
HTTP/1.1 202 Accepted
Location: https://resource.example/authorize/pending/abc123
Retry-After: 0
Cache-Control: no-store
AAuth-Requirement: requirement=interaction;
    url="https://resource.example/interaction"; code="A1B2-C3D4"
Content-Type: application/json

{
  "status": "pending"
}
```

The user completes interaction at the resource's own consent page. The agent polls the Location URL. When authorization is complete, the resource returns 200 OK and MAY include an AAuth-Access header Section 6.3 containing an opaque access token for subsequent calls.

```
HTTP/1.1 200 OK
AAuth-Access: wrapped-opaque-token-value
Content-Type: application/json

{
  "status": "authorized",
  "scope": "data.read data.write"
}
```

If the resource can authorize immediately (e.g., the agent's key is already authorized), it returns 200 OK directly with the optional AAuth-Access header.

6.2.2. Response with Resource Token

Alternatively, the resource MAY return a resource token. The resource sets the aud claim based on its configuration:

* If the resource has its own AS: aud = AS URL (four-party)
* If the resource has no AS but the agent has a PS (ps claim): aud = PS URL (three-party)

When the AAuth-Mission header is present, the resource includes the mission object (approver and s256) in the resource token.

```
{
  "resource_token": "eyJhbGc..."
}
```

The agent sends the resource token to its PS's token endpoint.

6.2.3. Authorization Endpoint Error Responses

Error	Status	Meaning
invalid_request	400	Missing or invalid parameters
invalid_signature	401	HTTP signature verification failed
invalid_scope	400	Requested scope not recognized by the resource
server_error	500	Internal error

Table 2

Error responses use the same format as the token endpoint Section 12.5.2.

6.3. AAuth-Access Response Header

The AAuth-Access response header carries an opaque access token from a resource to an agent. The token is opaque to the agent — the resource wraps its internal authorization state (which MAY be an existing OAuth access token or other credential). The agent passes the token back to the resource via the Authorization header on subsequent requests:

```
GET /api/data HTTP/1.1
Host: resource.example
Authorization: AAuth wrapped-opaque-token-value
Signature-Input: sig=("@method" "@authority" "@path" \
    "authorization" "signature-key");created=1730217600
Signature: sig=:...signature bytes...:
Signature-Key: sig=jwt;jwt="eyJhbGc..."
```

The agent MUST include authorization in the covered components of its HTTP signature, binding the access token to the signed request. This prevents the token from being stolen and replayed as a standalone bearer token — the token is useless without a valid AAuth signature from the agent.

A resource MAY return a new AAuth-Access header on any response, replacing the agent's current access token. This enables rolling refresh without an explicit refresh flow. When the agent receives a new AAuth-Access value, it MUST use the new value on subsequent requests.

6.4. Resource-Managed Authorization

When a resource manages authorization itself and requires user interaction, it returns a 202 Accepted response with an interaction requirement:

```
HTTP/1.1 202 Accepted
Location: https://resource.example/pending/abc123
Retry-After: 0
Cache-Control: no-store
AAuth-Requirement: requirement=interaction;
    url="https://resource.example/interaction"; code="A1B2-C3D4"
Content-Type: application/json

{
  "status": "pending"
}
```

The agent directs the user to the interaction URL Section 7.2 and polls the Location URL per the deferred response pattern Section 12.4. When the interaction completes, the resource returns 200 OK and MAY include an AAuth-Access header Section 6.3 with an opaque access token for subsequent calls.

A resource MAY also authorize the agent based solely on its identity (from the agent token) without any interaction — for example, when the agent's key is already known or the agent's domain is trusted.

6.5. Auth Token Required

A resource MUST use requirement=auth-token with a 401 Unauthorized response when an auth token is required. The header MUST include a resource-token parameter containing a resource token JWT Section 6.6.1.

```
HTTP/1.1 401 Unauthorized
AAuth-Requirement: requirement=auth-token; resource-token="eyJ..."
```

The agent MUST extract the resource-token parameter, verify the resource token Section 6.6.3, and present it to its PS's token endpoint to obtain an auth token Section 7.1. A resource MAY also use 402 Payment Required with the same AAuth-Requirement header when payment is additionally required Section 12.3.

A resource MAY return requirement=auth-token with a new resource token to a request that already includes an auth token — for example, when the request requires a higher level of authorization than the current token provides. Agents MUST be prepared for this step-up authorization at any time.

6.6. Resource Token

6.6.1. Resource Token Structure

A resource token is a JWT with typ: aa-resource+jwt containing:

Header: - alg: Signing algorithm. EdDSA is RECOMMENDED. Implementations MUST NOT accept none. - typ: aa-resource+jwt - kid: Key identifier

Payload: - iss: Resource URL - dwk: aauth-resource.json — the well-known metadata document name for key discovery ([I-D.hardt-httpbis-signature-key]) - aud: Token audience — the PS URL (when the resource delegates authorization to the agent's PS) or the AS URL (when the resource has its own access server) - jti: Unique token identifier for replay detection, audit, and revocation - agent: Agent identifier - agent_jkt: JWK Thumbprint ([RFC7638]) of the agent's current signing key - iat: Issued at timestamp - exp: Expiration timestamp - scope: Requested scopes, as a space-separated string of scope values. Companion specifications MAY define alternative authorization claims that replace scope.

Optional payload claims: - mission: Mission object (present when the resource is mission-aware and the agent sent an AAuth-Mission header). Contains: - approver: HTTPS URL of the entity that approved the mission - s256: SHA-256 hash of the approved mission JSON (base64url)

Resource tokens SHOULD NOT have a lifetime exceeding 5 minutes. The jti claim provides an audit trail for token requests; ASes are not required to enforce replay detection on resource tokens. If a resource token expires before the PS presents it to the AS (e.g., because user interaction was required), the agent MUST obtain a fresh resource token from the resource and submit a new token request to the PS. The PS SHOULD remember prior consent decisions within a mission so the user is not re-prompted when the agent resubmits a request for the same resource and scope.

6.6.2. Resource Token Verification

Verify the resource token per [RFC7515] and [RFC7519]:

1. Decode the JWT header. Verify typ is aa-resource+jwt.
2. Verify dwk is aauth-resource.json. Discover the issuer's JWKS via {iss}/.well-known/{dwk} per the HTTP Signature Keys specification ([I-D.hardt-httpbis-signature-key]). Locate the key matching the JWT header kid and verify the JWT signature.
3. Verify exp is in the future and iat is not in the future.
4. Verify aud matches the recipient's own identifier (the PS in three-party, or the AS in four-party).

5. Verify agent matches the requesting agent's identifier.
6. Verify agent_jkt matches the JWK Thumbprint of the key used to sign the HTTP request.
7. If mission is present, verify mission.approver matches the PS that sent the token request.

6.6.3. Resource Challenge Verification

When an agent receives a 401 response with AAuth-Requirement: requirement=auth-token:

1. Extract the resource-token parameter.
2. Decode and verify the resource token JWT.
3. Verify iss matches the resource the agent sent the request to.
4. Verify agent matches the agent's own identifier.
5. Verify agent_jkt matches the JWK Thumbprint of the agent's signing key.
6. Verify exp is in the future.
7. Send the resource token to the agent's PS's token endpoint.

7. Person Server

This section defines how agents obtain authorization from their person server. When accessing a remote resource, the agent sends a resource token to the PS's token endpoint. When performing local actions not governed by a remote resource, the agent requests permission from the PS's permission endpoint. In both cases, the PS evaluates the request against mission scope, handles user consent if needed, and uses the same requirement response patterns.

7.1. PS Token Endpoint

The PS's token_endpoint is where agents send token requests. The PS evaluates the request, handles user consent if needed, and either issues the auth token directly or federates with the resource's AS.

7.1.1. Token Endpoint Modes

Mode	Key Parameters	Use Case
Direct issuance	resource_token (aud = PS)	PS issues auth token directly (three-party)
Federated issuance	resource_token (aud = AS)	PS federates with AS for auth token (four-party)
Call chaining	resource_token +	Resource acting as agent

	upstream_token	

Table 3

7.1.2. Concurrent Token Requests

An agent MAY have multiple token requests pending at the PS simultaneously — for example, when a mission requires access to several resources. Each request has its own pending URL and lifecycle. The PS MUST handle concurrent requests independently. Some requests may be resolved without user interaction (e.g., within existing mission scope), while others may require consent. The PS is responsible for managing concurrent user interactions — for example, by batching consent prompts or serializing them.

7.1.3. Agent Token Request

The agent MUST make a signed POST to the PS's token_endpoint. The request MUST include an HTTP Sig Section 12.7 and the agent MUST present its agent token via the Signature-Key header using scheme=jwt.

Request parameters:

* resource_token (REQUIRED): The resource token.
* upstream_token (OPTIONAL): An auth token from an upstream authorization, used in call chaining Section 10.1.
* justification (OPTIONAL): A Markdown string declaring why access is being requested. The PS SHOULD present this value to the user during consent. The PS MUST sanitize the Markdown before rendering to users. The PS MAY log the justification for audit and monitoring purposes. *TODO:* Define recommended sections.
* login_hint (OPTIONAL): Hint about who to authorize, per [OpenID.Core] Section 3.1.2.1.
* tenant (OPTIONAL): Tenant identifier, per OpenID Connect Enterprise Extensions 1.0 [OpenID.Enterprise].
* domain_hint (OPTIONAL): Domain hint, per OpenID Connect Enterprise Extensions 1.0 [OpenID.Enterprise].

Example request:

```
POST /token HTTP/1.1
Host: ps.example
Content-Type: application/json
Prefer: wait=45
Signature-Input: sig=("@method" "@authority"
    "@path" "signature-key");created=1730217600
Signature: sig=:...signature bytes...:
Signature-Key: sig=jwt;jwt="eyJhbGc..."

{
  "resource_token": "eyJhbGc...",
  "justification": "Find available meeting times"
}
```

7.1.4. PS Response

When the resource token's aud matches the PS's own identifier (three-party), the PS evaluates the request and issues the auth token directly — no AS federation is needed. When aud identifies a different server (four-party), the PS federates with the AS per Section 9.3.

In both cases, the PS handles user consent if needed and returns one of:

Direct grant response (200):

```
{
  "auth_token": "eyJhbGc...",
  "expires_in": 3600
}
```

User interaction required response (202):

```
HTTP/1.1 202 Accepted
Location: /pending/abc123
Retry-After: 0
Cache-Control: no-store
AAuth-Requirement: requirement=interaction;
    url="https://ps.example/interaction"; code="ABCD1234"
Content-Type: application/json

{
  "status": "pending"
}
```

In four-party mode, the PS may also pass through a clarification from the AS to the agent via the 202 response Section 9.1.

7.2. User Interaction

When a server responds with 202 and AAuth-Requirement: requirement=interaction, the url and code parameters in the header tell the agent where to send the user Section 12.3. The agent constructs the user-facing URL as {url}?code={code} and directs the user using one of the methods defined in Section 12.3 (browser redirect, QR code, or display code).

When the agent has a browser, it MAY append a callback parameter:

{url}?code={code}&callback={callback_url}

The callback URL is constructed from the agent's callback_endpoint metadata. When present, the server redirects the user's browser to the callback URL after the user completes the action. If no callback parameter is provided, the server displays a completion page and the agent relies on polling to detect completion.

The code parameter is single-use: once the user arrives at the URL with a valid code, the code is consumed and cannot be reused.

7.3. Clarification Chat

During user consent, the user may ask questions about the agent's stated justification. The PS delivers these questions to the agent, and the agent responds. This enables a consent dialog without requiring the agent to have a direct channel to the user.

Agents that support clarification chat declare this via the AAuth-Capabilities request header Section 12.1 by including the clarification capability value.

7.3.1. Clarification Required

A server MUST use requirement=clarification with a 202 Accepted response when it needs the recipient to answer a question before proceeding. The response body MUST include a clarification field containing the question and MAY include timeout and options fields.

```
HTTP/1.1 202 Accepted
Location: /pending/abc123
Retry-After: 0
Cache-Control: no-store
AAuth-Requirement: requirement=clarification
Content-Type: application/json

{
  "status": "pending",
  "clarification": "Why do you need write access to my calendar?",
  "timeout": 120
}
```

Body fields:

* clarification (REQUIRED): A Markdown string containing the question.
* timeout (OPTIONAL): Seconds until the server times out the request. The recipient MUST respond before this deadline.
* options (OPTIONAL): An array of string values when the question has discrete choices.

The recipient MUST respond with one of the actions defined in Section 7.3.3: a clarification response, an updated request, or a cancellation. This requirement is used by both PSes (delivering user questions to agents) and ASes (requesting clarification from PSes).

7.3.2. Clarification Flow

When the user asks a question during consent, the PS returns a 202 with AAuth-Requirement: requirement=clarification.

7.3.3. Agent Response to Clarification

The agent MUST respond to a clarification with one of:

1. *Clarification response*: POST a clarification_response to the pending URL.
2. *Updated request*: POST a new resource_token to the pending URL, replacing the original request with updated scope or parameters.
3. *Cancel request*: DELETE the pending URL to withdraw the request.

7.3.3.1. Clarification Response

The agent responds by POSTing JSON with clarification_response to the pending URL:

```
POST /pending/abc123 HTTP/1.1
Host: ps.example
Content-Type: application/json
Signature-Input: sig=("@method" "@authority"
    "@path" "signature-key");created=1730217600
Signature: sig=:...signature bytes...:
Signature-Key: sig=jwt;jwt="eyJhbGc..."

{
  "clarification_response":
    "I need to create a meeting invite
     for the participants you listed."
}
```

The clarification_response value is a Markdown string. *TODO:* Define recommended sections. After posting, the agent resumes polling with GET.

7.3.3.2. Updated Request

The agent MAY obtain a new resource token from the resource (e.g., with reduced scope) and POST it to the pending URL:

```
POST /pending/abc123 HTTP/1.1
Host: ps.example
Content-Type: application/json
Signature-Input: sig=("@method" "@authority"
    "@path" "signature-key");created=1730217600
Signature: sig=:...signature bytes...:
Signature-Key: sig=jwt;jwt="eyJhbGc..."

{
  "resource_token": "eyJ...",
  "justification": "I've reduced my request to read-only access."
}
```

The new resource token MUST have the same iss, agent, and agent_jkt as the original. The PS presents the updated request to the user. A justification is OPTIONAL but RECOMMENDED to explain the change to the user.

7.3.3.3. Cancel Request

The agent MAY cancel the request by sending DELETE to the pending URL:

```
DELETE /pending/abc123 HTTP/1.1
Host: ps.example
Signature-Input: sig=("@method" "@authority"
    "@path" "signature-key");created=1730217600
Signature: sig=:...signature bytes...:
Signature-Key: sig=jwt;jwt="eyJhbGc..."
```

The PS terminates the consent session and informs the user that the agent withdrew its request. Subsequent requests to the pending URL return 410 Gone.

7.3.4. Clarification Limits

PSes SHOULD enforce limits on clarification rounds (recommended: 5 rounds maximum). Clarification responses from agents are untrusted input and MUST be sanitized before display to the user.

7.4. Permission Endpoint

The permission endpoint enables agents to request permission from the PS for actions not governed by a remote resource — for example, executing tool calls, writing files, or sending messages on behalf of the user. This enables governance before any resources support AAuth. The permission endpoint MAY be used with or without a mission.

When a mission is active, the mission approval MAY include a list of pre-approved tools in the approved_tools field. The agent calls the permission endpoint only for actions not covered by pre-approved tools.

7.4.1. Permission Request

The agent MUST make a signed POST to the PS’s permission_endpoint. The request MUST include an HTTP Sig Section 12.7 and the agent MUST present its agent token via the Signature-Key header.

Request parameters:

* action (REQUIRED): A string identifying the action the agent wants to perform (e.g., a tool name).
* description (OPTIONAL): A Markdown string describing what the action will do and why.
* parameters (OPTIONAL): A JSON object containing the parameters the agent intends to pass to the action.
* mission (OPTIONAL): Mission object with approver and s256 fields, binding the request to a mission. When present, the PS evaluates the request against the mission context and log history.

```
POST /permission HTTP/1.1
Host: ps.example
Content-Type: application/json
Signature-Input: sig=("@method" "@authority" "@path" \
    "signature-key");created=1730217600
Signature: sig=:...signature bytes...:
Signature-Key: sig=jwt;jwt="eyJhbGc..."

{
  "action": "SendEmail",
  "description": "Send the proposed itinerary to the user",
  "parameters": {
    "to": "user@example.com",
    "subject": "Japan trip itinerary"
  },
  "mission": {
    "approver": "https://ps.example",
    "s256": "dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk"
  }
}
```

7.4.2. Permission Response

If the PS can decide immediately, it returns 200 OK:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "permission": "granted"
}
```

The permission field is one of:

* granted: The agent MAY proceed with the action.
* denied: The agent MUST NOT proceed. The response MAY include a reason field with a Markdown string explaining why.

If the mission is no longer active, the PS returns a mission status error Section 8.6.

If the PS requires user input, it returns a deferred response Section 12.4 using the same pattern as other AAuth endpoints. The agent polls until the PS returns a final response.

The PS SHOULD record all permission requests and responses. When a mission is present, the PS records the permission request and response in the mission log.

7.5. Audit Endpoint

The audit endpoint enables agents to log actions they have performed, providing the PS with a record for governance and monitoring. The agent sends a signed POST to the PS's audit_endpoint after performing an action. The audit endpoint requires a mission — there is no audit outside a mission context.

7.5.1. Audit Request

The agent MUST make a signed POST to the PS's audit_endpoint. The request MUST include an HTTP Sig Section 12.7 and the agent MUST present its agent token via the Signature-Key header.

Request parameters:

* mission (REQUIRED): Mission object with approver and s256 fields.
* action (REQUIRED): A string identifying the action that was performed.
* description (OPTIONAL): A Markdown string describing what was done and the outcome.
* parameters (OPTIONAL): A JSON object containing the parameters that were used.
* result (OPTIONAL): A JSON object containing the result or outcome of the action.

```
POST /audit HTTP/1.1
Host: ps.example
Content-Type: application/json
Signature-Input: sig=("@method" "@authority" "@path" \
    "signature-key");created=1730217600
Signature: sig=:...signature bytes...:
Signature-Key: sig=jwt;jwt="eyJhbGc..."

{
  "mission": {
    "approver": "https://ps.example",
    "s256": "dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk"
  },
  "action": "WebSearch",
  "description": "Searched for flights to Tokyo in May",
  "parameters": {
    "query": "flights to Tokyo May 2026"
  },
  "result": {
    "status": "completed",
    "summary": "Found 12 flight options"
  }
}
```

7.5.2. Audit Response

The PS returns 201 Created to acknowledge the record:

```
HTTP/1.1 201 Created
```

The audit endpoint is fire-and-forget — the agent SHOULD NOT block on the response. The PS records the audit entry in the mission log. The PS MAY use audit records to detect anomalous behavior, alert the user, or revoke the mission.

If the mission is no longer active, the PS returns a mission status error Section 8.6.

7.6. Interaction Endpoint

The interaction endpoint enables the agent to reach the user through the PS. The agent uses this endpoint to forward interaction requirements from resources that it cannot handle directly, to ask the user questions, to relay payment approvals, or to propose mission completion. The interaction_endpoint URL is published in the PS's well-known metadata Section 12.10.2. The interaction endpoint MAY be used with or without a mission.

7.6.1. Interaction Request

The agent MUST make a signed POST to the PS's interaction_endpoint. The request MUST include an HTTP Sig Section 12.7 and the agent MUST present its agent token via the Signature-Key header.

Request parameters:

* type (REQUIRED): The type of interaction. One of interaction, payment, question, or completion.
* description (OPTIONAL): A Markdown string providing context for the user.
* url (OPTIONAL): The interaction URL to relay to the user (for interaction and payment types).
* code (OPTIONAL): The interaction code associated with the URL.
* question (OPTIONAL): A Markdown string containing a question for the user (for question type).
* summary (OPTIONAL): A Markdown string summarizing what the agent accomplished (for completion type).
* mission (OPTIONAL): Mission object with approver and s256 fields, binding the request to a mission.

Relay interaction example:

```
POST /interaction HTTP/1.1
Host: ps.example
Content-Type: application/json
Signature-Input: sig=("@method" "@authority" "@path" \
    "signature-key");created=1730217600
Signature: sig=:...signature bytes...:
Signature-Key: sig=jwt;jwt="eyJhbGc..."

{
  "type": "interaction",
  "description": "The booking service needs you to confirm payment",
  "url": "https://booking.example/confirm",
  "code": "X7K2-M9P4",
  "mission": {
    "approver": "https://ps.example",
    "s256": "dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk"
  }
}
```

Completion example:

```
POST /interaction HTTP/1.1
Host: ps.example
Content-Type: application/json
Signature-Input: sig=("@method" "@authority" "@path" \
    "signature-key");created=1730217600
Signature: sig=:...signature bytes...:
Signature-Key: sig=jwt;jwt="eyJhbGc..."

{
  "type": "completion",
  "summary": "# Japan Trip Booked\n\n
    Booked round-trip flights on ANA and
    10 nights across three cities.
    Total cost: $4,850.
    Itinerary sent to your email.",
  "mission": {
    "approver": "https://ps.example",
    "s256": "dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk"
  }
}
```

7.6.2. Interaction Response

For interaction and payment types, the PS relays the interaction to the user and returns a deferred response Section 12.4. The agent polls until the user completes the interaction.

For question type, the PS delivers the question to the user and returns the answer:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "answer": "Yes, go ahead with the refundable option."
}
```

For completion type, the PS presents the summary to the user. The user either accepts — the PS terminates the mission and returns 200 OK — or responds with follow-up questions via clarification Section 7.3, keeping the mission active. The PS returns a deferred response while the user reviews.

If the PS cannot reach the user and the agent does not have the interaction capability, the PS returns interaction_required. If the mission is no longer active, the PS returns a mission status error Section 8.6. The PS SHOULD record all interaction requests and responses. When a mission is active, the PS records the interaction in the mission log.

7.7. Re-authorization

AAuth does not have a separate refresh token or refresh flow. When an auth token expires, the agent obtains a fresh resource token from the resource's authorization endpoint and submits it to the PS's token endpoint — the same flow as the initial authorization. This gives the resource a voice in every re-authorization: the resource can adjust scope, require step-up authorization, or deny access based on current policy.

When an agent rotates its signing key, all existing auth tokens are bound to the old key and can no longer be used. The agent MUST re-authorize by obtaining fresh resource tokens and submitting them to the PS.

Agents SHOULD proactively obtain a new agent token and refresh all auth tokens before the current agent token expires, to avoid service interruptions. Auth tokens MUST NOT have an exp value that exceeds the exp of the agent token used to obtain them — a resource MUST reject an auth token whose associated agent token has expired.

8. Mission

Missions are OPTIONAL. The protocol operates in all modes without missions. When used, missions provide scoped authorization contexts that guide an agent's work across multiple resource accesses — enabling scope pre-approval, reduced consent fatigue, and centralized audit. A mission is a natural-language description of what the agent intends to accomplish, proposed by the agent and approved by the PS. The PS uses the mission to evaluate every subsequent request in context — it is the only party with the mission content, the user relationship, and the full history of the agent's actions. Once approved, the mission's s256 identifier is included in subsequent resource interactions via the AAuth-Mission header.

8.1. Mission Creation

The agent creates a mission by sending a proposal to the PS's mission_endpoint. The agent MUST make a signed POST with an HTTP Sig Section 12.7, presenting its agent token via the Signature-Key header using scheme=jwt.

The proposal includes a Markdown description of what the agent intends to accomplish, and MAY include a list of tools the agent wants to use:

```
{
  "description": "# Plan Japan Vacation\n\n
    Plan and book a trip to Japan next month
    for 2 adults. Budget around $5k.
    Propose an itinerary before booking.",
  "tools": [
    {
      "name": "WebSearch",
      "description": "Search the web"
    },
    {
      "name": "BookFlight",
      "description": "Book flights"
    },
    {
      "name": "BookHotel",
      "description": "Book hotels"
    }
  ]
}
```

The PS MAY return a 202 Accepted deferred response Section 12.4 if human review, clarification, or approval is needed. During this phase, the PS and user may engage in clarification chat Section 7.3 with the agent to refine the mission scope, ask questions about the agent's intent, or negotiate which tools are needed. The PS or user may also modify the description — the approved mission MAY differ from the original proposal.

8.2. Mission Approval

When the PS approves the mission, the response body is a JSON object — the *mission blob* — containing the approved mission and session-specific information. The PS returns the AAuth-Mission header with the approver and s256 values:

```
HTTP/1.1 200 OK
Content-Type: application/json
AAuth-Mission: approver="https://ps.example";
    s256="dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk"

{
  "approver": "https://ps.example",
  "agent": "aauth:assistant@agent.example",
  "approved_at": "2026-04-07T14:30:00Z",
  "description": "# Plan Japan Vacation\n\n
    Plan and book a trip to Japan next month
    for 2 adults. Budget around $5k.
    Propose an itinerary before booking.",
  "approved_tools": [
    {
      "name": "WebSearch",
      "description": "Search the web"
    },
    {
      "name": "Read",
      "description": "Read files and web pages"
    }
  ],
  "capabilities": [
    "interaction",
    "payment"
  ]
}
```

The mission blob MUST include:

* approver: HTTPS URL of the entity that approved the mission. Currently this is always the PS.
* agent: The agent identifier (aauth:local@domain).
* approved_at: ISO 8601 timestamp of when the mission was approved. Ensures the s256 is globally unique.
* description: Markdown string describing the approved mission scope.

The mission blob MAY include:

* approved_tools: Array of tool objects (each with name and description) that the agent may use without per-call permission at the PS's permission endpoint Section 7.4.
* capabilities: Array of capability strings (e.g., interaction, payment) that the PS can provide on behalf of the user for this session. The PS determines these based on whether it can reach the specific user — for example, via push notification, email, or

an active session. The agent unions these with its own capabilities when constructing the AAuth-Capabilities request header Section 12.1.

The s256 in the AAuth-Mission header is the base64url-encoded SHA-256 hash of the response body bytes. The agent verifies the hash by computing SHA-256 over the exact response body bytes. The agent MUST store the mission body bytes exactly as received — no re-serialization.

The approved description MAY differ from the proposal — the PS or user may refine, constrain, or expand the mission during review. The approved tools MAY be a subset of the proposed tools. The agent MUST use the approver and s256 from the AAuth-Mission header in all subsequent AAuth-Mission request headers.

8.3. Mission Log

The approved mission description is immutable — the s256 hash binds it permanently. Missions do not change; they accumulate context.

All agent interactions with the PS within a mission context form the *mission log*: token requests (with justifications), permission requests and responses, audit records, interaction requests, and clarification chats. The PS maintains this log as an ordered record of the agent's actions and the governance decisions made. The mission log gives the PS the full history it needs to evaluate whether each new request is consistent with the mission's intent.

The agent includes the mission context in all resource interactions via the AAuth-Mission header. When the agent sends a resource token to its PS, the PS evaluates the request against the mission context and log history before federating with the resource's AS.

8.4. Mission Completion

When the agent believes the mission is complete, it sends a completion interaction to the PS's interaction endpoint Section 7.6 with a summary of what was accomplished. The PS presents the summary to the user. The user either accepts — the PS terminates the mission — or responds with follow-up questions via clarification, keeping the mission active. This is the most common mission lifecycle path.

8.5. Mission Management

A mission has one of two states:

* *active*: The mission is in progress. The agent can make requests against it.
* *terminated*: The mission is permanently ended. The PS MUST reject requests with mission_terminated.

The mechanisms for state transitions beyond completion — revocation, delegation tree queries, and administrative interfaces — will be defined in a companion specification.

8.6. Mission Status Errors

When an agent makes a request to any PS endpoint with a mission parameter referencing a mission that is no longer active, the PS MUST return an error:

```
HTTP/1.1 403 Forbidden
Content-Type: application/json

{
  "error": "mission_terminated",
  "mission_status": "terminated"
}
```

Error	Mission Status	Meaning
mission_terminated	terminated	The mission is permanently ended. The agent MUST stop acting on this mission.

Table 4

8.7. AAuth-Mission Request Header

The AAuth-Mission header is a request header sent by the agent on initial requests to a resource when operating in a mission context. It signals to the resource that the agent has a person server and is operating within a mission.

```
AAuth-Mission:
    approver="https://ps.example";
    s256="dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk"
```

Parameters:

* approver: The HTTPS URL of the entity that approved the mission

* s256: The base64url-encoded SHA-256 hash of the approved mission JSON

When a mission-aware resource receives a request with the AAuth-Mission header, it includes the mission object (approver and s256) in the resource token it issues. When a resource does not support missions, it ignores the header.

Agents operating in a mission context MUST include the AAuth-Mission header on requests to resources that do not include an auth token containing a mission claim.

9. Access Server Federation

This section defines auth tokens and the mechanisms by which they are issued. The auth token is the end result of the authorization flow — a JWT issued by an access server that grants an agent access to a specific resource. This section covers the AS token endpoint, PS-AS federation, and the auth token structure.

9.1. AS Token Endpoint

The AS evaluates resource policy and issues auth tokens. It accepts JSON POST requests.

9.1.1. PS-to-AS Token Request

The PS MUST make a signed POST to the AS’s token_endpoint. The PS authenticates via an HTTP Sig Section 12.7.

Request parameters:

* resource_token (REQUIRED): The resource token issued by the resource.
* agent_token (REQUIRED): The agent’s agent token.
* upstream_token (OPTIONAL): An auth token from an upstream authorization, used in call chaining Section 10.1.

Example request:

```
POST /token HTTP/1.1
Host: as.resource.example
Content-Type: application/json
Signature-Input: sig=("@method" "@authority"
    "@path" "signature-key");created=1730217600
Signature: sig=:...signature bytes...:
Signature-Key: sig=jwks_uri;
    jwks_uri="https://ps.example/.well-known/jwks.json"

{
  "resource_token": "eyJhbGc...",
  "agent_token": "eyJhbGc..."
}
```

9.1.2. AS Response

The PS calls the AS token endpoint and follows the standard deferred response loop Section 12.4: it handles 202 and 402 responses and continues until it receives a 200 with an auth token or a terminal error.

Direct grant response (200):

```
{
  "auth_token": "eyJhbGc...",
  "expires_in": 3600
}
```

The AS MAY return 202 Accepted with an AAuth-Requirement header indicating what is needed before it can issue an auth token:

- *requirement=claims* Section 9.2: The AS needs identity claims. The body includes required_claims. The PS MUST provide the requested claims (including a directed sub identifier for the resource) by POSTing to the Location URL. The AS cannot know what claims it needs until it has processed the resource token.
- *requirement=clarification* Section 7.3.1: The AS needs a question answered. The PS triages who answers: itself (if mission context has the answer), the user, or the agent. The PS MAY pass the clarification down to the agent via a 202 response.
- *requirement=interaction* Section 12.3: The AS requires user interaction — for example, the user must authenticate at the AS to bind their PS, or the resource owner must approve access. The PS directs the user to the AS's interaction URL, or passes the interaction requirement back to the agent.
- *requirement=approval* Section 12.3: The AS is obtaining approval without requiring user direction.

Payment required (402):

The AS MAY return 402 Payment Required when a billing relationship is required before it will issue auth tokens. The 402 response includes payment details per an applicable payment protocol such as x402 [x402] or the Machine Payment Protocol (MPP) ([I-D.ryan-httpauth-payment]). The response MUST include a Location header for the PS to poll after payment is settled.

```
HTTP/1.1 402 Payment Required
Location: https://as.resource.example/token/pending/xyz
WWW-Authenticate: Payment id="x7Tg2pLq", method="stripe",
    request="eyJhbW91bnQiOiIxMDAw..."
```

The PS settles payment per the indicated protocol and polls the Location URL. When payment is confirmed, the AS continues processing the token request — which may result in a 200 with an auth token, or a further 202 requiring claims, interaction, or approval.

The PS caches the billing relationship per AS. Future token requests from the same PS to the same AS skip the billing step. The payment protocol, settlement mechanism, and billing terms are out of scope for this specification.

9.1.3. Auth Token Delivery

When the AS issues an auth token (200 response), the PS MUST verify the auth token before returning it to the agent:

1. Verify the auth token JWT signature using the AS's JWKS Section 12.8.
2. Verify iss matches the AS the PS sent the token request to.
3. Verify aud matches the resource identified by the resource token's iss.
4. Verify agent matches the agent that submitted the token request.
5. Verify cnf.jwk matches the agent's signing key.
6. Verify act is present and accurately reflects the delegation chain — act.sub identifies the requesting agent, and any nested act claims match the upstream delegation context.
7. Verify scope is consistent with what was requested — not broader than the scope in the resource token.

After verification, the PS returns the auth token to the agent. The agent presents the auth token to the resource via the Signature-Key header Section 9.4.2. The resource verifies the auth token against the AS's JWKS Section 9.4.3.

The agent receives the auth token from its trusted PS, so signature verification is not strictly required. However, agents SHOULD verify the auth token's signature to detect errors early. Agents MUST verify that aud, cnf, agent, and act match their own values.

9.2. Claims Required

A server MUST use requirement=claims with a 202 Accepted response when it needs identity claims to process a request. The response body MUST include a required_claims field containing an array of claim names.

```
HTTP/1.1 202 Accepted
Location: https://as.resource.example/token/pending/xyz
Retry-After: 0
Cache-Control: no-store
AAuth-Requirement: requirement=claims
Content-Type: application/json

{
  "status": "pending",
  "required_claims": ["email", "org"]
}
```

The recipient MUST provide the requested claims (including a directed user identifier as sub) by POSTing to the Location URL. The recipient MUST include an HTTP Sig Section 12.7 on the POST. Claims not recognized by the recipient SHOULD be ignored. This requirement is used by ASes to request identity claims from PSes during token issuance.

9.3. PS-AS Federation

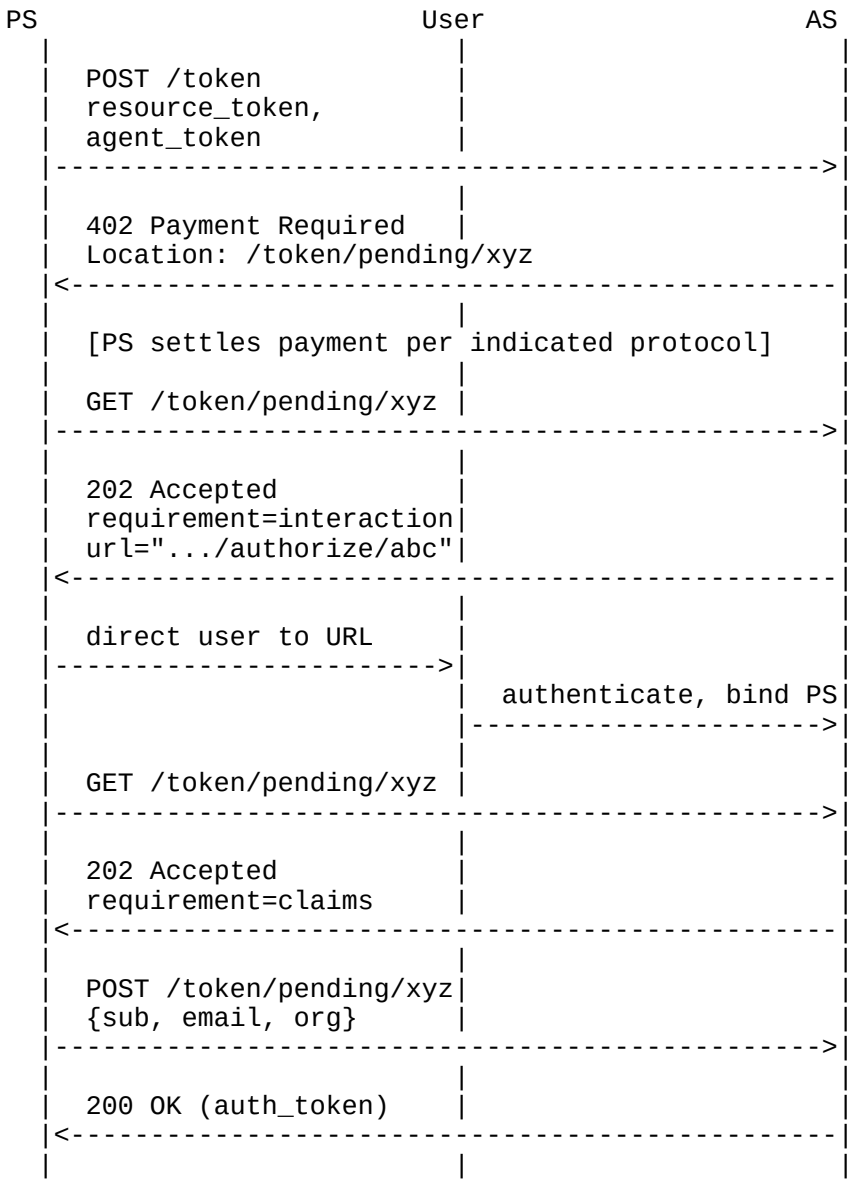
The PS is the only entity that calls AS token endpoints. When the PS receives a resource token from an agent, the resource token's aud claim identifies where to send the token request. If aud matches the PS's own identifier, the PS issues the auth token directly (three-party). If aud identifies a different server (an AS), the PS discovers the AS's metadata at {aud}/.well-known/aauth-access.json Section 12.10.3 and calls the AS's token_endpoint Section 9.1 (four-party).

9.3.1. PS-AS Trust Establishment

Trust between the PS and AS is not a separate registration step — it emerges from the AS's response to the PS's first token request. The AS evaluates the token request and responds based on its current policy:

* *Pre-established*: A business relationship configured between the PS and AS, potentially including payment terms, SLA, and compliance requirements. The AS recognizes the PS and processes the token request directly.
* *Interaction*: The AS returns 202 with requirement=interaction, directing the user to authenticate at the AS and confirm their PS. After this one-time binding, the AS trusts future requests from that PS for that user. This is the primary mechanism for establishing trust dynamically.
* *Payment*: The AS returns 402, requiring the PS to establish a billing relationship before tokens will be issued. The PS settles payment per the indicated protocol and polls for completion. After billing is established, the AS trusts future requests from that PS.
* *Claims only*: The AS may trust any PS that can provide sufficient identity claims for a policy decision, without requiring a prior relationship.

These mechanisms may compose: for example, the AS may first require payment (402), then interaction for user binding (202), then claims (202) before issuing an auth token. Each step uses the same Location URL for polling.



{: #fig-mm-as-trust title="PS-AS Trust Establishment (all steps shown — most requests skip some)"}

9.3.2. AS Decision Logic (Non-Normative)

The following is a non-normative description of how an AS might evaluate a token request:

1. *PS = AS (same entity)*: Grant directly. When an organization controls both the PS and AS, the federation call is internal and trust is implicit.
2. *User has bound this PS at the AS*: Apply the user's configured policy for this PS.
3. *PS is pre-established (enterprise agreement)*: Apply the organization's configured policy.
4. *Resource is open or has a free tier*: Grant with restricted scope or rate limits.
5. *Resource requires billing*: Return 402 with payment details.
6. *Resource requires user binding*: Return 202 with requirement=interaction.
7. *AS needs identity claims to decide*: Return 202 with requirement=claims.
8. *Insufficient trust for requested scope*: Return 403.

The AS is not required to follow this order. The decision logic is entirely at the AS's discretion based on resource policy.

9.3.3. Organization Visibility

Organizations benefit from the trust model: an organization's agents share a single PS, and internal resources may share a single AS. The PS provides centralized audit across all agents and missions. Federation is only incurred at the boundary, when an internal agent accesses an external resource. When an organization controls both the PS and AS, the federation call is internal and trust is implicit — this is the degenerate case of the four-party model collapsing to fewer parties.

9.4. Auth Token

9.4.1. Auth Token Structure

An auth token is a JWT with typ: aa-auth+jwt containing:

Header: - alg: Signing algorithm. EdDSA is RECOMMENDED. Implementations MUST NOT accept none. - typ: aa-auth+jwt - kid: Key identifier

Required payload claims: - iss: The URL of the server that issued the auth token — an AS (four-party) or a PS (three-party) - dwk: The well-known metadata document name for key discovery ([I-D.hardt-httpbis-signature-key]). aauth-access.json when issued by an AS, aauth-person.json when issued by a PS. - aud: The URL of the resource the agent is authorized to access. - jti: Unique token identifier for replay detection, audit, and revocation - agent: Agent identifier - cnf: Confirmation claim with jwk containing the agent's

public key - act: Actor claim ([RFC8693], Section 4.1) identifying the entity that requested this auth token. In direct authorization, act.sub is the agent identifier. In call chaining, act nests to record the full delegation chain — each intermediary's identity is preserved as a nested act claim within the outer act. This enables resources to see the complete chain of delegation and make authorization decisions accordingly. - iat: Issued at timestamp - exp: Expiration timestamp. Auth tokens MUST NOT have a lifetime exceeding 1 hour.

Conditional payload claims (at least one MUST be present): - sub: Directed user identifier. An opaque string that identifies the user. The PS SHOULD provide a pairwise pseudonymous identifier per resource (aud), preserving user privacy — different resources see different sub values for the same user. - scope: Authorized scopes, as a space-separated string of scope values consistent with [RFC9068] Section 2.2.3

At least one of sub or scope MUST be present.

Optional payload claims: - mission: Mission object. Present when the auth token was issued in the context of a mission. Contains: - approver: HTTPS URL of the entity that approved the mission - s256: SHA-256 hash of the approved mission JSON (base64url)

The auth token MAY include additional claims registered in the IANA JSON Web Token Claims Registry [RFC7519] or defined in OpenID Connect Core 1.0 [OpenID.Core] Section 5.1.

9.4.2. Auth Token Usage

Agents present auth tokens via the Signature-Key header ([I-D.hardt-httpbis-signature-key]) using scheme=jwt:

```
Signature-Key: sig=jwt;
    jwt="eyJhbGciOiJFZERTQSIsInR5cCI6ImF1dGgr..."
```

9.4.3. Auth Token Verification

When a resource receives an auth token, verify per [RFC7515] and [RFC7519]:

1. Decode the JWT header. Verify typ is aa-auth+jwt.
2. Verify dwk is aauth-access.json (AS-issued) or aauth-person.json (PS-issued). Discover the issuer's JWKS via {iss}/.well-known/{dwk} per the HTTP Signature Keys specification ([I-D.hardt-httpbis-signature-key]). Locate the key matching the JWT header kid and verify the JWT signature.

3. Verify exp is in the future and iat is not in the future.
4. Verify iss is a valid HTTPS URL.
5. Verify aud matches the resource's own identifier.
6. Verify agent matches the agent identifier from the request's signing context.
7. Verify cnf.jwk matches the key used to sign the HTTP request.
8. Verify act is present and act.sub matches the agent identifier from the request's signing context.
9. Verify that at least one of sub or scope is present.

9.4.4. Auth Token Response Verification

When an agent receives an auth token:

1. SHOULD verify the auth token JWT signature using the issuer's JWKS (the AS in four-party, or the PS in three-party). The agent trusts its PS, so signature verification is not required but is RECOMMENDED to detect errors early.
2. Verify iss matches the resource token's aud claim.
3. Verify aud matches the resource the agent intends to access.
4. Verify cnf.jwk matches the agent's own signing key.
5. Verify agent matches the agent's own identifier.
6. Verify act.sub matches the agent's own identifier.

9.4.5. Upstream Token Verification

When the PS receives an upstream_token parameter in a call chaining request:

1. Perform Auth Token Verification Section 9.4.3 on the upstream token.
2. Verify iss is a trusted AS (an AS whose auth token the PS previously brokered).
3. Verify the aud in the upstream token matches the resource that is now acting as an agent (i.e., the upstream token was issued for the intermediary resource).
4. The PS constructs the act claim for the downstream auth token by nesting the upstream token's act claim inside a new act object identifying the intermediary resource's agent identity. This preserves the complete delegation chain.
5. The PS evaluates its own policy based on the upstream token's claims and mission context. The resulting downstream authorization is not required to be a subset of the upstream scopes — see Section 10.1.

10. Multi-Hop Resource Access

This section defines how resources act as agents to access downstream resources on behalf of the original caller. In multi-hop scenarios, a resource that receives an authorized request needs to access another resource to fulfill that request. The resource acts as an agent — it has its own agent identity and signing key — and routes the downstream authorization to obtain an auth token for the downstream resource.

10.1. Call Chaining

When a resource needs to access a downstream resource on behalf of the caller, it acts as an agent. The resource determines where to send the downstream token request based on the upstream auth token it received:

* *Mission present* (mission.approver in the upstream auth token): The resource sends the downstream resource token to the PS identified by mission.approver, along with its own agent token and the upstream auth token as the upstream_token. The PS has mission context and evaluates the downstream request against the mission scope. This is the governed path — the PS sees the full delegation chain for audit.

* *No mission, iss is a PS* (three-party upstream): The resource sends the downstream resource token to the PS identified by iss, along with its own agent token and the upstream_token. The PS evaluates the request without mission context.

* *No mission, iss is an AS* (four-party upstream, no governance): The resource sends the downstream resource token to the AS identified by iss, along with its own agent token and the upstream_token. The AS evaluates the request based on resource policy. No PS is involved — no governance context is available.

The recipient (PS or AS) evaluates the downstream request per Section 9.4.5.

Note that downstream authorization is not required to be a subset of the upstream scopes. A downstream resource may have capabilities that are orthogonal to the upstream resource — for example, a flight booking API that calls a payment processor needs the payment processor to charge a card, an operation the user and original agent could never perform directly. The downstream resource's scope is constrained by its own AS policy and the PS's evaluation of the mission context, not by the upstream token's scope. The PS provides the governance constraint — it evaluates each hop independently and can deny requests that fall outside the mission or the user's intent.

Because the resource acts as an agent, it MUST have its own agent identity — it MUST publish agent metadata at /.well-known/aauth-agent.json so that downstream resources and ASes can verify its identity.

10.2. Interaction Chaining

When the PS or AS requires user interaction for the downstream access, it returns a 202 with requirement=interaction. Resource 1 chains the interaction back to the original agent by returning its own 202.

When a resource acting as an agent receives a 202 Accepted response with AAuth-Requirement: requirement=interaction, and the resource needs to propagate this interaction requirement to its caller, it MUST return a 202 Accepted response to the original agent with its own AAuth-Requirement header containing requirement=interaction and its own interaction code. The resource MUST provide its own Location URL for the original agent to poll. When the user completes interaction and the resource obtains the downstream auth token, the resource completes the original request and returns the result at its pending URL.

11. Third-Party Login

A third party — such as a PS, enterprise portal, app marketplace, or partner site — can direct a user to an agent's or resource's login_endpoint to initiate authentication. The agent or resource creates a resource token and sends it to the PS's token endpoint, obtaining an auth token with user identity.

This enables use cases where the user's journey starts outside the agent or resource — for example, an enterprise portal launching an agent for a specific user, an app marketplace connecting a user to a new service, or a PS dashboard directing a user to an agent.

11.1. Login Endpoint

Agents and resources MAY publish a login_endpoint in their metadata. The login_endpoint accepts the following query parameters:

* ps (REQUIRED): The PS URL to authenticate with. The agent or resource MUST verify this is a valid PS by fetching its metadata at {ps}/.well-known/aauth-person.json Section 12.10.2.
* login_hint (OPTIONAL): Hint about who to authorize, per [OpenID.Core] Section 3.1.2.1.
* domain_hint (OPTIONAL): Domain hint, per OpenID Connect Enterprise Extensions 1.0 [OpenID.Enterprise].
* tenant (OPTIONAL): Tenant identifier, per OpenID Connect Enterprise Extensions 1.0 [OpenID.Enterprise].
* start_path (OPTIONAL): Path on the agent's or resource's origin where the user should be directed after login completes. The recipient MUST validate that start_path is a relative path on its own origin.

Example login URL:

```
https://agent.example/login
    ?ps=https://ps.example
    &tenant=corp
    &login_hint=user@corp.example
    &start_path=/projects/tokyo-trip
```

11.2. Login Flow

Upon receiving a request at its login_endpoint, the agent or resource:

1. Validates the ps parameter by fetching the PS's metadata.
2. Creates a resource token with aud = PS URL, binding the request to its own identity.
3. POSTs to the PS's token_endpoint with the resource token and any provided login_hint, domain_hint, or tenant parameters.
4. Proceeds with the standard deferred response flow Section 12.4 — directing the user to the PS's interaction endpoint with the interaction code.
5. After obtaining the auth token, redirects the user to start_path if provided, or to a default landing page.

If the user is already authenticated at the PS, the interaction step resolves near-instantly — the PS recognizes the user from its own session. If not, the user completes a normal authentication and consent flow.

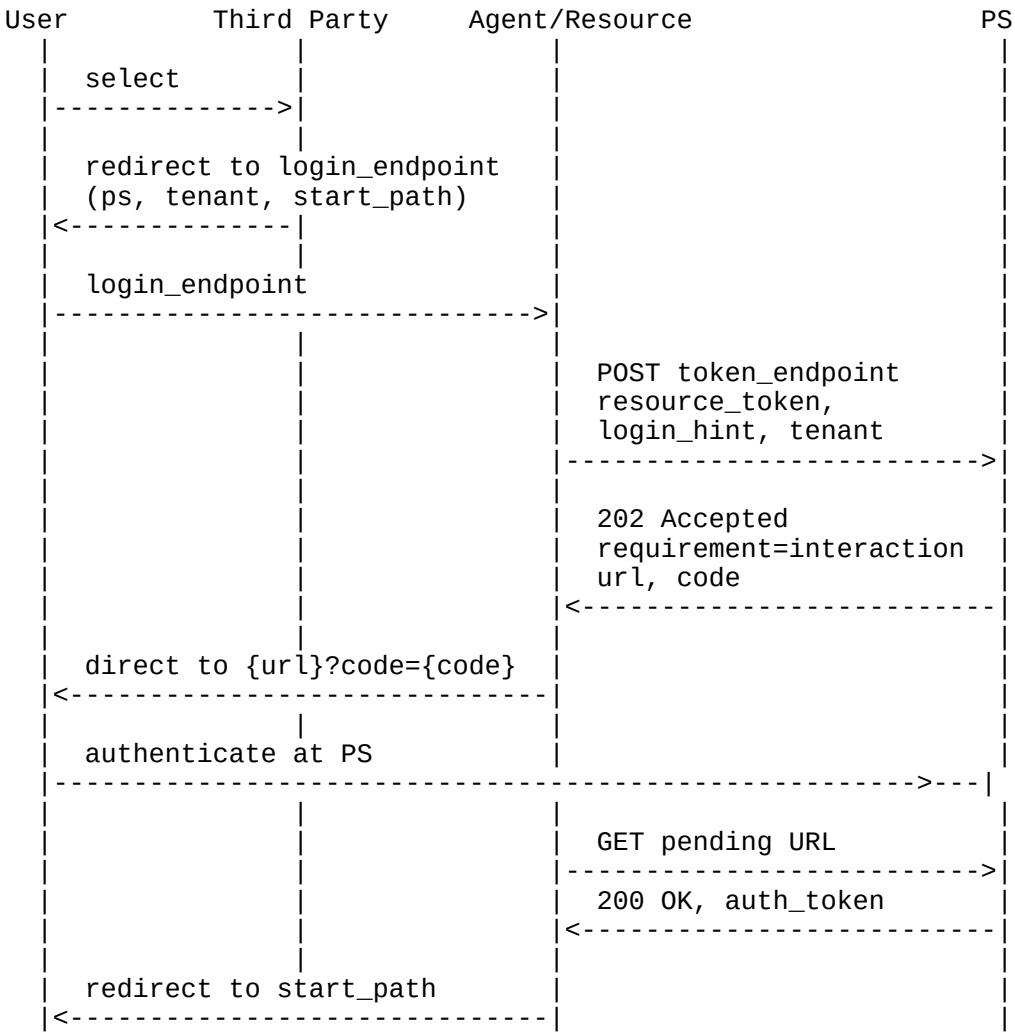


Figure 9: Third-Party Login Flow

The third party does not need to be the PS. Any party that knows the agent's or resource's login_endpoint (from metadata) can initiate the flow. The agent or resource treats the redirect as untrusted input — it verifies the PS through metadata discovery and initiates a signed flow.

11.3. Security Considerations for Third-Party Login

* The login_endpoint does not carry any tokens, codes, or pre-authorized state. The agent or resource initiates a standard signed flow with the PS, which independently authenticates the user.
* The start_path parameter MUST be validated as a relative path on the recipient's own origin to prevent open redirect attacks.
* The ps parameter is untrusted input. The agent or resource MUST discover and verify the PS via its well-known metadata before proceeding.

12. Protocol Primitives

This section defines the common mechanisms used across all AAuth endpoints: requirement responses, capabilities, deferred responses, error responses, scopes, token revocation, HTTP message signatures, key discovery, identifiers, and metadata documents.

12.1. AAuth-Capabilities Request Header

Agents use the AAuth-Capabilities request header to declare which protocol capabilities they can handle. This allows resources and PSes to tailor their responses — for example, a resource that sees interaction in the capabilities knows it can send requirement=interaction, whereas a resource that does not see interaction knows it must use an alternative path (such as issuing a resource token for three-party mode).

The AAuth-Capabilities header field is a List ([RFC8941], Section 3.1) of Tokens.

AAuth-Capabilities: interaction, clarification, payment

This specification defines the following capability values:

Value	Meaning
interaction	Agent can get a user to a URL — either directly (user is present) or via its PS's interaction endpoint
clarification	Agent can engage in back-and-forth clarification chat
payment	Agent can handle 402 payment flows — either directly or via its PS's interaction endpoint

Table 5

The agent determines its capabilities by combining what it can do directly with what its PS can do on its behalf. When the agent has a PS and has created a mission, the mission approval response includes a capabilities array listing what the PS can handle for this user/session Section 8.2. The agent unions its own capabilities with the PS's capabilities to produce the AAuth-Capabilities header value.

Agents SHOULD include the AAuth-Capabilities header on signed requests to resources. The header is not used on requests to PS endpoints — the PS learns the agent's capabilities through the mission approval flow. Recipients MUST ignore unrecognized capability values. When the header is absent, recipients MUST NOT assume any capabilities — the agent may not support interaction, clarification, or payment flows.

12.2. Scopes

Scopes define what an agent is authorized to do at a resource. AAuth uses two categories of scope values:

* *Resource scopes*: Resource-specific authorization grants (e.g., data.read, data.write, data.delete). Each resource defines its own scope values and publishes human-readable descriptions in its metadata (scope_descriptions). Resources that already define OAuth scopes SHOULD use the same scope values in AAuth.
* *Identity scopes*: Requests for user identity claims following [OpenID.Core] (e.g., openid, profile, email, address, phone). When identity scopes are present, the auth token includes the corresponding identity claims. Enterprise identity extensions (e.g., org, groups, roles) follow [OpenID.Enterprise].

A resource token MUST only include resource scopes that the resource has defined in its scope_descriptions metadata, and identity scopes that the PS has declared in its scopes_supported metadata. This ensures all parties can interpret and present the requested scopes.

Scopes appear in three places in the protocol:

1. *Resource token* (scope): The scope the resource is willing to grant, as determined by the resource based on the agent's request at the authorization endpoint.
2. *Auth token* (scope): The scope actually granted. The auth token's scope MUST NOT be broader than the resource token's scope.
3. *Authorization endpoint request* (scope): The scope the agent is requesting from the resource.

The PS evaluates requested scopes against mission context (if present) and user consent. The AS evaluates scopes against resource policy. Either party may narrow the granted scope.

12.3. Requirement Responses

Servers use the AAuth-Requirement response header to indicate protocol-level requirements to agents. The header MAY be sent with 401 Unauthorized or 202 Accepted responses. A 401 response indicates that authorization is required. A 202 response indicates that the request is pending and additional action is required — either user interaction (requirement=interaction) or third-party approval (requirement=approval).

AAuth-Requirement and WWW-Authenticate are independent header fields; a response MAY include both. A client that understands AAuth processes AAuth-Requirement; a legacy client processes WWW-Authenticate. Neither header's presence invalidates the other.

The header MAY also be sent with 402 Payment Required when a server requires both authorization and payment. The AAuth-Requirement conveys the authorization requirement; the payment requirement is conveyed by a separate mechanism such as x402 [x402] or the Machine Payment Protocol (MPP) ([I-D.ryan-httpauth-payment]).

12.3.1. AAuth-Requirement Header Structure

The AAuth-Requirement header field is a Dictionary ([RFC8941], Section 3.2). It MUST contain the following member:

* requirement: A Token ([RFC8941], Section 3.3.4) indicating the requirement type.

Additional members are defined per requirement value. Recipients MUST ignore unknown members.

Example:

```
AAuth-Requirement: requirement=auth-token; resource-token="eyJ..."
```

12.3.2. Requirement Values

The requirement value is an extension point. This document defines the following values:

Value	Status Code	Meaning	Resource	PS	AS
auth-token	401	Auth token required for resource access	Y		
interaction	202	User action required at an interaction endpoint	Y	Y	Y
approval	202	Approval pending, poll for result	Y	Y	Y
clarification	202	Question posed to the recipient	Y	Y	Y
claims	202	Identity claims required			Y

Table 6

The auth-token requirement is defined in Section 6.5; the interaction and approval requirements are defined in this section; clarification in Section 7.3.1; and claims in Section 9.2.

12.3.3. Interaction Required

When a server requires user action — such as authentication, consent, payment approval, or any decision requiring a human in the loop — it returns a 202 Accepted response:

```
HTTP/1.1 202 Accepted
AAuth-Requirement:
    requirement=interaction;
    url="https://example.com/interact";
    code="A1B2-C3D4"
Location: /pending/f7a3b9c
Retry-After: 0
```

The AAuth-Requirement header MUST include the following parameters:

* url (String): The interaction URL where the user completes the required action. MUST use the https scheme and MUST NOT contain query or fragment components.
* code (String): An interaction code that links the agent's pending request to the user's session at the interaction URL.

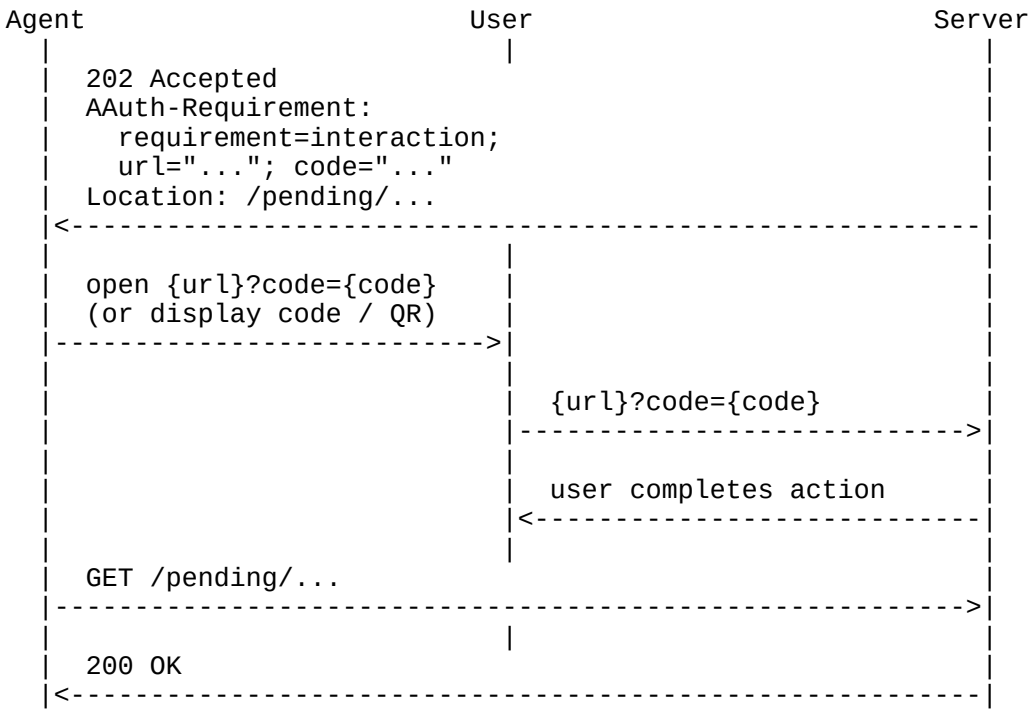
The response MUST also include:

* Location: A URL the agent polls (with GET) for a terminal response.
* Retry-After: Recommended polling interval in seconds.

The agent constructs a user-facing URL by appending the code as a query parameter: {url}?code={code}. The agent then directs the user to this URL using one of:

* *Browser redirect*: The agent opens the URL in the user's browser.
* *Display code*: The agent displays the url and code for the user to enter manually. The agent MAY also render the constructed URL as a QR code for the user to scan with their phone.

After directing the user, the agent polls the Location URL with GET requests, respecting the Retry-After interval. A 202 response means the request is still pending. A non-202 response is terminal — 200 indicates success, 403 indicates denial, and 408 indicates timeout.



Use cases: User login, consent, payment confirmation, document review, CAPTCHA, any workflow requiring human action.

12.3.4. Approval Pending

When a server is obtaining approval from another party without requiring the agent to direct a user — for example, via push notification, email, or administrator review:

```
HTTP/1.1 202 Accepted
AAuth-Requirement: requirement=approval
Location: /pending/f7a3b9c
Retry-After: 30
```

The response MUST include Location and Retry-After. The agent polls the Location URL with GET requests until a terminal response is received. No user action is required at the agent side. The same terminal response codes apply as for interaction.

Use cases: Administrator approval, resource owner consent, compliance review, direct user authorization via established communication channel.

12.4. Deferred Responses

Any endpoint in AAuth — whether a PS token endpoint, AS token endpoint, or resource endpoint — MAY return a 202 Accepted response ([RFC9110]) when it cannot immediately resolve a request. This is a first-class protocol primitive, not a special case. Agents MUST handle 202 responses regardless of the nature of the original request.

12.4.1. Initial Request

The agent makes a request and signals its willingness to wait using the Prefer header ([RFC7240]):

```
POST /token HTTP/1.1
Host: auth.example
Content-Type: application/json
Prefer: wait=45
Signature-Input: sig=("@method" "@authority"
    "@path" "signature-key");created=1730217600
Signature: sig=:...signature bytes...:
Signature-Key: sig=jwt;jwt="eyJhbGc..."

{
  "resource_token": "eyJhbGc..."
}
```

12.4.2. Pending Response

When the server cannot resolve the request within the wait period:

```
HTTP/1.1 202 Accepted
Location: /pending/f7a3b9c
Retry-After: 0
Cache-Control: no-store
Content-Type: application/json

{
  "status": "pending"
}
```

Headers:

* Location (REQUIRED): The pending URL. The Location URL MUST be on the same origin as the responding server.
* Retry-After (REQUIRED): Seconds the agent SHOULD wait before polling. 0 means retry immediately.

* Cache-Control: no-store (REQUIRED): Prevents caching of pending responses.
* AAuth-Requirement (OPTIONAL): Present when user interaction or approval is required. The url and code parameters are defined in Section 12.3.

Body fields:

* status (REQUIRED): "pending" while the request is waiting. "interacting" when the user has arrived at the interaction endpoint. Agents MUST treat unrecognized status values as "pending" and continue polling.

Additional body fields may be present depending on the AAuth-Requirement value — for example, clarification and timeout with requirement=clarification, or required_claims with requirement=claims. See the specific requirement definitions for details.

12.4.3. Polling with GET

After receiving a 202, the agent switches to GET for all subsequent requests to the Location URL. The agent does NOT resend the original request body. *Exception*: During clarification chat, the agent uses POST to deliver a clarification response.

The agent MUST respect Retry-After values. If a Retry-After header is not present, the default polling interval is 5 seconds. If the server responds with 429 Too Many Requests, the agent MUST increase its polling interval by 5 seconds (linear backoff, following the pattern in [RFC8628], Section 3.5). The Prefer: wait=N header ([RFC7240]) MAY be included on polling requests to signal the agent's willingness to wait for a long-poll response.

12.4.4. Deferred Response State Machine

The following state machine applies to any AAuth endpoint that returns a 202 Accepted response — including PS token endpoints, AS token endpoints, and resource endpoints during call chaining. A non-202 response terminates polling.

```
Initial request (with Prefer: wait=N)
    |
    +-- 200 --> done — process response body
    +-- 202 --> note Location URL, check requirement/code
    +-- 400 --> invalid request — check error field, fix and retry
    +-- 401 --> invalid signature — check credentials;
    |           obtain auth token if resource challenge
    +-- 402 --> payment required (settle payment, poll Location)
    +-- 500 --> server error — start over
    +-- 503 --> back off per Retry-After, retry
                |
                GET Location (with Prefer: wait=N)
                |
                +-- 200 --> done — process response body
                +-- 202 --> continue polling (check status/clarification)
                |           status=interacting → stop prompting user
                +-- 403 --> denied or abandoned — surface to user
                +-- 408 --> expired — MAY initiate a fresh request
                +-- 410 --> gone — MUST NOT retry
                +-- 429 --> slow down — increase interval by 5s
                +-- 500 --> server error — start over
                +-- 503 --> temporarily unavailable
                            back off per Retry-After
```

12.5. Error Responses

12.5.1. Authentication Errors

A 401 response from any AAuth endpoint uses the Signature-Error header as defined in ([I-D.hardt-httpbis-signature-key]).

12.5.2. Token Endpoint Error Response Format

Token endpoint errors use Content-Type: application/json ([RFC8259]) with the following members:

* error (REQUIRED): String. A single error code.
* error_description (OPTIONAL): String. A human-readable description.

12.5.3. Token Endpoint Error Codes

Error	Status	Meaning
invalid_request	400	Malformed JSON, missing required fields

invalid_agent_token	400	Agent token malformed or signature verification failed
expired_agent_token	400	Agent token has expired
invalid_resource_token	400	Resource token malformed or signature verification failed
expired_resource_token	400	Resource token has expired
interaction_required	403	User interaction is needed but no interaction channel is available — the PS cannot reach the user and the agent does not have the interaction capability
server_error	500	Internal error

Table 7

12.5.4. Polling Error Codes

Error	Status	Meaning
denied	403	User or approver explicitly denied the request
abandoned	403	Interaction code was used but user did not complete
expired	408	Timed out
invalid_code	410	Interaction code not recognized or already consumed
slow_down	429	Polling too frequently — increase interval by 5 seconds
server_error	500	Internal error

Table 8

12.6. Token Revocation

Any AAuth server that issues tokens MAY provide a revocation endpoint. The endpoint accepts a signed POST with the jti of the token to revoke. The server identifies the token from the jti and its own records — no token type is needed since the jti is unique within the issuer's namespace.

Request:

```
POST /revoke HTTP/1.1
Host: ps.example
Content-Type: application/json
Signature-Input: sig=("@method" "@authority"
    "@path" "signature-key");created=1730217600
Signature: sig=:...signature bytes...:
Signature-Key: sig=jwt;jwt="eyJhbGc..."

{
  "jti": "unique-token-identifier"
}
```

Response: 200 OK if the token was revoked or was already invalid. 404 if the jti is not recognized.

Revocation provides real-time termination of access. The PS or AS calls the revocation endpoint of the resource that a token was issued for, passing the jti of the auth token to revoke. The following revocation scenarios are supported:

* *PS revokes an auth token it issued* (three-party): The PS calls the resource's revocation endpoint with the auth token's jti.
* *PS revokes an auth token it provided* (four-party): The PS calls the resource's revocation endpoint with the auth token's jti. The PS MAY also notify the AS.
* *AS revokes an auth token it issued*: The AS calls the resource's revocation endpoint with the auth token's jti.
* *PS revokes a mission*: The PS marks the mission as revoked. All subsequent token requests referencing that mission's s256 are denied. The PS SHOULD revoke outstanding auth tokens issued under the mission.
* *Agent server stops issuing agent tokens*: The agent server decides not to issue new agent tokens to the agent. Existing agent tokens expire naturally. This is part of the regular token lifecycle — all tokens have limited lifetimes and require periodic re-issuance, which provides a natural policy re-evaluation point.

Revocation endpoints are advertised in server metadata as revocation_endpoint. Resources that accept revocation requests MUST verify the caller's identity via HTTP Message Signatures and MUST only accept revocation from the issuer of the token being revoked or from a trusted PS.

Auth tokens are short-lived (maximum 1 hour) and proof-of-possession (useless without the bound signing key). All AAuth tokens have limited lifetimes — agent tokens, resource tokens, and auth tokens all expire and require re-issuance. Each re-issuance is a policy evaluation point where the issuer can deny renewal. This natural expiration cycle, combined with real-time revocation, provides layered access control.

12.7. HTTP Message Signatures Profile

This section profiles HTTP Message Signatures ([RFC9421]) for use with AAuth. Signing requirements (what the agent does) and verification requirements (what the server does) are specified separately.

12.7.1. Signature Algorithms

Agents and resources MUST support EdDSA using Ed25519 ([RFC8032]). Agents and resources SHOULD support ECDSA using P-256 with deterministic signatures ([RFC6979]). The alg parameter in the JWK ([RFC7517]) key representation identifies the algorithm. See the IANA JSON Web Signature and Encryption Algorithms registry ([RFC7518], Section 7.1) for the full list of algorithm identifiers.

12.7.2. Keying Material

The signing key is conveyed in the Signature-Key header ([I-D.hardt-httpbis-signature-key]). The Signature-Key scheme determines how the server obtains the public key:

* For pseudonym: the agent uses scheme=hwk (inline public key) or scheme=jkt-jwt (delegation from a hardware-backed key).
* For identity: the agent uses scheme=jwks_uri (JWKS endpoint) or scheme=jwt (JWT with public key in cnf claim).

See the Signature-Key specification ([I-D.hardt-httpbis-signature-key]) for scheme definitions, key discovery, and verification procedures.

12.7.3. Signing (Agent)

The agent creates an HTTP Message Signature ([RFC9421]) on each request, including the following headers:

* Signature-Key: Public key or key reference for signature verification
* Signature-Input: Signature metadata including covered components
* Signature: The HTTP message signature

12.7.3.1. Covered Components

The signature MUST cover the following derived components and header fields:

* @method: The HTTP request method ([RFC9421], Section 2.2.1)
* @authority: The target host ([RFC9421], Section 2.2.3)
* @path: The request path ([RFC9421], Section 2.2.6)
* signature-key: The Signature-Key header value

Servers MAY require additional covered components (e.g., content-digest ([RFC9530]) for request body integrity). The agent learns about additional requirements from server metadata or from an invalid_input error response that includes required_input.

12.7.3.2. Signature Parameters

The Signature-Input header ([RFC9421], Section 4.1) MUST include the following parameters:

* created: Signature creation timestamp as an Integer (Unix time). The agent MUST set this to the current time.

12.7.4. Verification (Server)

When a server receives a signed request, it MUST perform the following steps. Any failure MUST result in a 401 response with the appropriate Signature-Error header ([I-D.hardt-httpbis-signature-key]).

1. Extract the Signature, Signature-Input, and Signature-Key headers. If any are missing, return invalid_request.
2. Verify that the Signature-Input covers the required components defined in Section 12.7.3.1. If the server requires additional components, verify those are covered as well. If not, return invalid_input with required_input.

3. Verify the created parameter is present and within the server's signature validity window of the server's current time. The default window is 60 seconds. Servers MAY advertise a different window via their metadata (e.g., signature_window in resource metadata). Reject with invalid_signature if outside this window. Servers and agents SHOULD synchronize their clocks using NTP ([RFC5905]).
4. Determine the signature algorithm from the alg parameter in the key. If the algorithm is not supported, return unsupported_algorithm.
5. Obtain the public key from the Signature-Key header according to the scheme, as specified in ([I-D.hardt-httpbis-signature-key]). Return invalid_key if the key cannot be parsed, unknown_key if the key is not found at the jwks_uri, invalid_jwt if a JWT scheme fails verification, or expired_jwt if the JWT has expired.
6. Verify the HTTP Message Signature ([RFC9421]) using the obtained public key and determined algorithm. Return invalid_signature if verification fails.

12.8. JWKS Discovery and Caching

All AAuth token verification — agent tokens, resource tokens, and auth tokens — requires discovering the issuer's signing keys via the {iss}/.well-known/{dwk} pattern defined in the HTTP Signature Keys specification ([I-D.hardt-httpbis-signature-key]).

Implementations MUST cache JWKS responses and SHOULD respect HTTP cache headers (Cache-Control, Expires) returned by the JWKS endpoint. When an implementation encounters an unknown kid in a JWT header, it SHOULD refresh the cached JWKS for that issuer to support key rotation. To prevent abuse, implementations MUST NOT fetch a given issuer's JWKS more frequently than once per minute. If a JWKS fetch fails, implementations SHOULD use the cached JWKS if available and SHOULD retry with exponential backoff. Cached JWKS entries SHOULD be discarded after a maximum of 24 hours regardless of cache headers, to ensure removed keys are no longer trusted.

12.9. Identifiers

12.9.1. Server Identifiers

The issuer values in metadata documents that identify agent servers, resources, access servers, and person servers MUST conform to the following:

* MUST use the https scheme
* MUST contain only scheme and host (no port, path, query, or fragment)

* MUST NOT include a trailing slash
* MUST be lowercase
* Internationalized domain names MUST use the ASCII-Compatible Encoding (ACE) form (A-labels) as defined in [RFC5890]

Valid identifiers:

* https://agent.example
* https://xn--nxasmq6b.example (internationalized domain in ACE form)

Invalid identifiers:

* http://agent.example (not HTTPS)
* https://Agent.Example (not lowercase)
* https://agent.example:8443 (contains port)
* https://agent.example/v1 (contains path)
* https://agent.example/ (trailing slash)

Implementations MUST perform exact string comparison on server identifiers.

12.9.2. Endpoint URLs

The token_endpoint, authorization_endpoint, mission_endpoint, and callback_endpoint values MUST conform to the following:

* MUST use the https scheme
* MUST NOT contain a fragment
* MUST NOT contain a query string

When localhost_callback_allowed is true in the agent's metadata, the agent MAY use a localhost callback URL as the callback parameter to the interaction endpoint.

12.9.3. Other URLs

The jwks_uri, tos_uri, policy_uri, logo_uri, and logo_dark_uri values MUST use the https scheme.

12.10. Metadata Documents

Participants publish metadata at well-known URLs ([RFC8615]) to enable discovery.

12.10.1. Agent Server Metadata

Published at /.well-known/aauth-agent.json:

```
{
  "issuer": "https://agent.example",
  "jwks_uri": "https://agent.example/.well-known/jwks.json",
  "client_name": "Example AI Assistant",
  "logo_uri": "https://agent.example/logo.png",
  "logo_dark_uri": "https://agent.example/logo-dark.png",
  "callback_endpoint": "https://agent.example/callback",
  "localhost_callback_allowed": true,
  "tos_uri": "https://agent.example/tos",
  "policy_uri": "https://agent.example/privacy"
}
```

Fields:

* issuer (REQUIRED): The agent server's HTTPS URL (the domain in agent identifiers it issues). This is the value placed in the iss claim of agent tokens.
* jwks_uri (REQUIRED): URL to the agent server's JSON Web Key Set
* client_name (OPTIONAL): Human-readable agent name (per [RFC7591])
* logo_uri (OPTIONAL): URL to agent logo (per [RFC7591])
* logo_dark_uri (OPTIONAL): URL to agent logo for dark backgrounds
* callback_endpoint (OPTIONAL): The agent's HTTPS callback endpoint URL
* login_endpoint (OPTIONAL): URL where third parties can direct users to initiate authentication Section 11
* localhost_callback_allowed (OPTIONAL): Boolean. Default: false.
* tos_uri (OPTIONAL): URL to terms of service (per [RFC7591])
* policy_uri (OPTIONAL): URL to privacy policy (per [RFC7591])

12.10.2. Person Server Metadata

Published at /.well-known/aauth-person.json:

```
{
  "issuer": "https://ps.example",
  "token_endpoint": "https://ps.example/token",
  "mission_endpoint": "https://ps.example/mission",
  "permission_endpoint": "https://ps.example/permission",
  "audit_endpoint": "https://ps.example/audit",
  "interaction_endpoint": "https://ps.example/interaction",
  "mission_control_endpoint": "https://ps.example/mission-control",
  "jwks_uri": "https://ps.example/.well-known/jwks.json"
}
```

Fields:

* issuer (REQUIRED): The PS's HTTPS URL. MUST match the URL used to fetch the metadata document. This is the value placed in the iss claim of JWTs issued by the PS.
* token_endpoint (REQUIRED): URL where agents send token requests
* mission_endpoint (OPTIONAL): URL for mission lifecycle operations (proposal, status). Present when the PS supports missions.
* permission_endpoint (OPTIONAL): URL where agents request permission for actions not governed by a remote resource Section 7.4
* audit_endpoint (OPTIONAL): URL where agents log actions performed Section 7.5
* interaction_endpoint (OPTIONAL): URL where agents relay interactions to the user through the PS Section 7.6
* mission_control_endpoint (OPTIONAL): URL for mission administrative interface
* revocation_endpoint (OPTIONAL): URL where authorized parties can revoke tokens Section 12.6
* jwks_uri (REQUIRED): URL to the PS's JSON Web Key Set
* scopes_supported (RECOMMENDED): Array of scope values the PS supports, including identity scopes (e.g., openid, profile, email) and enterprise scopes (e.g., org, groups, roles)
* claims_supported (RECOMMENDED): Array of identity claim names the PS can provide (e.g., sub, email, name, org)

12.10.3. Access Server Metadata

Published at /.well-known/aauth-access.json:

```
{
  "issuer": "https://as.resource.example",
  "token_endpoint": "https://as.resource.example/token",
  "jwks_uri": "https://as.resource.example/.well-known/jwks.json"
}
```

Fields:

* issuer (REQUIRED): The AS's HTTPS URL. MUST match the URL used to fetch the metadata document. This is the value placed in the iss claim of auth tokens.
* token_endpoint (REQUIRED): URL where PSes send token requests
* revocation_endpoint (OPTIONAL): URL where authorized parties can revoke tokens Section 12.6
* jwks_uri (REQUIRED): URL to the AS's JSON Web Key Set

12.10.4. Resource Metadata

Published at /.well-known/aauth-resource.json:

```
{
  "issuer": "https://resource.example",
  "jwks_uri": "https://resource.example/.well-known/jwks.json",
  "client_name": "Example Data Service",
  "logo_uri": "https://resource.example/logo.png",
  "logo_dark_uri": "https://resource.example/logo-dark.png",
  "authorization_endpoint": "https://resource.example/authorize",
  "scope_descriptions": {
    "data.read": "Read access to your data and documents",
    "data.write": "Create and update your data and documents",
    "data.delete": "Permanently delete your data and documents"
  },
  "additional_signature_components": ["content-type", "content-digest"]
}
```

Fields:

* issuer (REQUIRED): The resource's HTTPS URL. This is the value placed in the iss claim of resource tokens.
* jwks_uri (REQUIRED): URL to the resource's JSON Web Key Set
* client_name (OPTIONAL): Human-readable resource name (per [RFC7591])
* logo_uri (OPTIONAL): URL to resource logo (per [RFC7591])
* logo_dark_uri (OPTIONAL): URL to resource logo for dark backgrounds
* authorization_endpoint (OPTIONAL): URL where agents request authorization Section 6.1. When absent, the resource issues resource tokens and interaction requirements via 401 responses (#requirement-auth-token, #resource-managed-auth).
* login_endpoint (OPTIONAL): URL where third parties can direct users to initiate authentication Section 11
* scope_descriptions (OPTIONAL): Object mapping scope values to Markdown strings for consent display. Scope values are resource-specific; resources that already define OAuth scopes SHOULD use the same scope values in AAuth. Identity-related scopes (e.g., openid, profile, email) follow [OpenID.Core].
* signature_window (OPTIONAL): Integer. The signature validity window in seconds for the created timestamp. Default: 60. Resources serving agents with poor clock synchronization (mobile, IoT) MAY advertise a larger value. High-security resources MAY advertise a smaller value.

* additional_signature_components (OPTIONAL): Array of HTTP message component identifiers ([RFC9421]) that agents MUST include in the Signature-Input covered components when signing requests to this resource, in addition to the base components required by the HTTP Message Signatures profile ([I-D.hardt-httpbis-signature-key])
* revocation_endpoint (OPTIONAL): URL where authorized parties can revoke auth tokens for this resource Section 12.6

13. Incremental Adoption

AAuth is designed for incremental adoption. Each party — agent, resource, PS, AS — can independently add support. The system works at every partial adoption state. No coordination is required between parties.

13.1. Agent Adoption Path

Each step builds on the previous one. An agent that adopts any step gains immediate value.

1. *Sign requests with HTTP Message Signatures*: The agent signs requests using the Signature-Key specification ([I-D.hardt-httpbis-signature-key]). Resources that recognize signatures can verify the agent's key and respond with Accept-Signature headers. Resources that don't recognize signatures ignore the headers — existing auth mechanisms continue to work.
2. *Obtain an agent token* (scheme=jwt, typ: aa-agent+jwt): The agent has a full AAuth identity with an aauth:local@domain identifier issued by an agent server, providing a stable, managed identity lifecycle. The agent token is presented via the Signature-Key header using scheme=jwt.
3. *Add a person server* (include ps claim in agent token): The agent can obtain auth tokens from its PS directly. Resources in three-party and four-party modes can issue resource tokens targeting the PS. Enables PS-issued auth tokens with user identity, organization membership, and group information.
4. *Add governance* (create a mission): The agent creates a mission at its PS, gaining permissions, audit, PS-relayed interactions, and consent-managed resource access. The mission can be as simple as the user's prompt.

13.2. Resource Adoption Path

Each step builds on the previous one. A resource that adopts any step works with agents at all identity levels.

1. *Recognize AAuth signatures*: Verify HTTP Message Signatures and respond with Accept-Signature headers ([I-D.hardt-httpbis-signature-key]). Resources that don't recognize AAuth ignore the signature headers — existing auth mechanisms continue to work. This is identity-based access.
2. *Manage authorization*: Handle authorization with interaction, consent, or existing infrastructure — via 401 responses, an authorization endpoint, or both. Return AAuth-Access headers Section 6.3 for subsequent calls. This is resource-managed access (two-party).
3. *Issue resource tokens to PS*: Read the ps claim from the agent token and issue resource tokens with aud = PS URL. This is PS-managed access (three-party).
4. *Deploy an access server*: Issue resource tokens with aud = AS URL. The PS federates with the AS. This is federated access (four-party).

13.3. Adoption Matrix

Agent	Resource	Mode	What Works
Signed requests	Recognizes signatures	Identity-based	Identity verification, access control by agent identity
Agent token	Manages authorization	Resource-managed	Resource-handled auth, interaction, AAuth-Access
Agent token + ps	Issues resource tokens	PS-managed	PS-issued auth tokens with user identity, org, groups
Agent token + ps	AS deployed	Federated	Full federation, AS policy enforcement
Agent token + ps + mission	Any or none	+ governance	Tool-call permissions, audit, PS-relayed interaction, consent-managed access

Table 9

14. Security Considerations

14.1. Proof-of-Possession

All AAuth tokens are proof-of-possession tokens. The holder must prove possession of the private key corresponding to the public key in the token's cnf claim.

14.2. Token Security

* Agent tokens bind agent keys to agent identity
* Resource tokens bind access requests to resource identity, preventing confused deputy attacks
* Auth tokens bind authorization grants to agent keys

14.3. Pending URL Security

* Pending URLs MUST be unguessable and SHOULD have limited lifetime
* Pending URLs MUST be on the same origin as the server that issued them
* Servers MUST verify the agent's identity on every poll
* Once a terminal response is returned, the pending URL MUST return 410 Gone

14.4. Clarification Chat Security

* PSes MUST enforce a maximum number of clarification rounds
* Clarification responses from agents are untrusted input and MUST be sanitized before display

14.5. Untrusted Input

All protocol inputs — JSON request bodies, clarification responses, justification strings, mission descriptions, and token claims — are untrusted input from potentially adversarial parties. This is consistent with standard web security practice where HTTP request bodies, headers, and query parameters are always treated as untrusted. Implementations MUST sanitize all values before rendering to users and MUST validate all values before processing. Markdown fields MUST be sanitized before rendering to prevent script injection.

14.6. Interaction Code Misdirection

An attacker could attempt to trick a user into approving an authorization request by directing them to an interaction URL with the attacker's code. The PS mitigates this by displaying the full request context — the agent's identity, the resource being accessed, and the requested scope — so the user can recognize requests they did not initiate. A stronger mitigation is for the PS to interact directly with the user via a pre-established channel (push notification, email, or existing session) using requirement=approval, which eliminates the possibility of misdirection through attacker-supplied links entirely.

14.7. AS Discovery

The resource's AS is identified by the aud claim in the resource token. In three-party mode, aud identifies the PS; in four-party mode, it identifies the AS. Federation mechanics are described in Section 9.3.

14.8. AAuth-Access Security

The AAuth-Access header carries an opaque wrapped token that is meaningful only to the issuing resource. The token MUST NOT be usable as a standalone bearer token — the resource wraps its internal authorization state so that the token is meaningless without a valid AAuth signature from the agent. The agent MUST include authorization in the signed components when presenting the token, binding it to the signed request.

14.9. PS as Auth Token Issuer

In three-party mode, the PS issues auth tokens directly without AS federation. The PS MUST protect its signing keys with the same rigor as an AS. Resources that accept PS-issued auth tokens are trusting the agent's PS — the trust basis differs from four-party mode where the resource trusts its own AS.

14.10. Agent-Person Binding

The PS MUST ensure that each agent is associated with exactly one person. This one-to-one binding is a trust invariant — it ensures that every action an agent takes is attributable to a single accountable party.

The binding is typically established when the person first authorizes the agent at the PS via the interaction flow. An organization administrator may pre-authorize agents for the organization. Once

established, the PS MUST NOT allow a different person to claim the same agent. If an agent's association needs to change (e.g., an employee leaves an organization), the existing binding MUST be revoked and a new binding established.

This invariant enables:

* *Accountability*: Every authorization decision traces to a single person.
* *Consent integrity*: Consent granted by one person cannot be exercised by a different person through the same agent.
* *Audit*: The PS can provide a complete record of an agent's actions on behalf of its person.
* *Revocation*: Revoking an agent's association with its person immediately prevents the agent from obtaining new auth tokens.

14.11. PS as High-Value Target

The PS is a centralized authority that sees every authorization in a mission. PS implementations MUST apply appropriate security controls including access control, audit logging, and monitoring. Compromise of a PS could affect all agents and missions it manages.

Several architectural properties mitigate this centralization risk. The person chooses their PS — no other party in the protocol imposes a PS, and the person can migrate to a different PS at any time. The PS MAY delegate authentication to an identity provider chosen by the person or organization (e.g., an enterprise IdP via OIDC federation), reducing the PS's role in credential management. The PS MAY also delegate policy evaluation to external services selected by the person, so that consent and authorization decisions are not solely determined by the PS operator. To the rest of the protocol, the PS presents a single interface regardless of how it is composed internally.

14.12. Call Chaining Identity

When a resource acts as an agent in call chaining, it uses its own signing key and presents its own credentials. The resource MUST publish agent metadata so downstream parties can verify its identity.

14.13. Token Revocation and Lifecycle

Real-time revocation Section 12.6 and short token lifetimes provide layered access control. Organizations have multiple control points — agent server, PS, and AS — each of which can deny renewal or revoke tokens independently. Shorter auth token lifetimes reduce the window between a control action and natural expiration.

14.14. TLS Requirements

All HTTPS connections MUST use TLS 1.2 or later, following the recommendations in BCP 195 [RFC9325].

15. Privacy Considerations

15.1. Directed Identifiers

The PS SHOULD provide a pairwise pseudonymous user identifier (sub) per resource, preventing resources from correlating users across trust domains. Each resource sees a different sub for the same user, preserving user privacy.

15.2. PS Visibility

In three-party and four-party modes, the PS sees every authorization request made by its agents — including the resource being accessed, the requested scope, and the mission context. This centralized visibility enables governance and audit, but it also means the PS is a sensitive data aggregation point. The person chooses to trust their PS with this visibility — no other party imposes the choice. PS implementations MUST apply appropriate access controls and data retention policies.

In two-party mode, no PS is involved and there is no centralized visibility — the resource handles authorization directly with the agent.

15.3. Mission Content Exposure

The mission JSON is visible to the PS and, when included in resource tokens and auth tokens via the s256 hash, its integrity is verifiable by any party that holds it. The approved mission JSON is shared between the agent and PS. Resources and ASes see only the s256 hash and the approver URL, not the full mission content.

16. IANA Considerations

16.1. HTTP Header Field Registration

This specification registers the following HTTP header fields in the "Hypertext Transfer Protocol (HTTP) Field Name Registry" established by [RFC9110]:

* Header Field Name: AAuth-Requirement

* Status: permanent

* Structured Type: Dictionary
* Reference: This document, Section 12.3
* Header Field Name: AAuth-Access
* Status: permanent
* Reference: This document, Section 6.3
* Header Field Name: AAuth-Capabilities
* Status: permanent
* Structured Type: List
* Reference: This document, Section 12.1
* Header Field Name: AAuth-Mission
* Status: permanent
* Structured Type: Dictionary
* Reference: This document, Section 8.7

16.2. HTTP Authentication Scheme Registration

This specification registers the following HTTP authentication scheme in the "Hypertext Transfer Protocol (HTTP) Authentication Scheme Registry" established by [RFC9110]:

* Authentication Scheme Name: AAuth
* Reference: This document, Section 6.3
* Notes: Used with opaque access tokens returned via the AAuth-Access header. The token MUST be bound to an HTTP Message Signature — the authorization field MUST be included in the signature's covered components.

16.3. Well-Known URI Registrations

This specification registers the following well-known URIs per [RFC8615]:

URI Suffix	Change Controller	Reference
aauth-agent.json	IETF	This document, Section 12.10.1
aauth-person.json	IETF	This document, Section 12.10.2
aauth-access.json	IETF	This document, Section 12.10.3
aauth-resource.json	IETF	This document, Section 12.10.4

Table 10

16.4. Media Type Registrations

This specification registers the following media types:

16.4.1. application/aa-agent+jwt

- Type name: application
- Subtype name: aa-agent+jwt
- Required parameters: N/A
- Optional parameters: N/A
- Encoding considerations: binary; a JWT is a sequence of Base64url-encoded parts separated by period characters
- Security considerations: See Section 14
- Interoperability considerations: N/A
- Published specification: This document, Section 5.2
- Applications that use this media type: AAuth agents, PSes, and ASes
- Fragment identifier considerations: N/A

16.4.2. application/aa-auth+jwt

- Type name: application
- Subtype name: aa-auth+jwt
- Required parameters: N/A
- Optional parameters: N/A
- Encoding considerations: binary; a JWT is a sequence of Base64url-encoded parts separated by period characters
- Security considerations: See Section 14
- Interoperability considerations: N/A
- Published specification: This document, Section 9.4

* Applications that use this media type: AAuth ASes, agents, and resources
* Fragment identifier considerations: N/A

16.4.3. application/aa-resource+jwt

* Type name: application
* Subtype name: aa-resource+jwt
* Required parameters: N/A
* Optional parameters: N/A
* Encoding considerations: binary; a JWT is a sequence of Base64url-encoded parts separated by period characters
* Security considerations: See Section 14
* Interoperability considerations: N/A
* Published specification: This document, Section 6
* Applications that use this media type: AAuth resources and ASes
* Fragment identifier considerations: N/A

16.5. JWT Type Registrations

This specification registers the following JWT typ header parameter values in the "JSON Web Token Types" sub-registry:

Type Value	Reference
aa-agent+jwt	This document, Section 5.2
aa-auth+jwt	This document, Section 9.4
aa-resource+jwt	This document, Section 6

Table 11

16.6. JWT Claims Registrations

This specification registers the following claims in the IANA "JSON Web Token Claims" registry established by [RFC7519]:

Claim Name	Claim Description	Change Controller	Reference
dwk	Discovery Well-Known document name	IETF	This document
ps	Person Server URL	IETF	This document
agent	Agent identifier	IETF	This document
agent_jkt	JWK Thumbprint of the agent's signing key	IETF	This document
mission	Mission object (approver, s256) in resource tokens and auth tokens	IETF	This document

Table 12

16.7. AAuth Requirement Value Registry

This specification establishes the AAuth Requirement Value Registry. The registry policy is Specification Required ([RFC8126]).

Value	Reference
interaction	This document
approval	This document
auth-token	This document
clarification	This document
claims	This document

Table 13

16.8. AAuth Capability Value Registry

This specification establishes the AAuth Capability Value Registry. The registry policy is Specification Required ([RFC8126]).

Value	Reference
interaction	This document
clarification	This document
payment	This document

Table 14

16.9. URI Scheme Registration

This specification registers the aauth URI scheme in the "Uniform Resource Identifier (URI) Schemes" registry ([RFC7595]):

* Scheme name: aauth
* Status: Permanent
* Applications/protocols that use this scheme: AAuth Protocol
* Contact: IETF
* Change controller: IETF
* Reference: This document, Section 5.1

The aauth URI scheme follows the pattern established by the acct scheme ([RFC7565]). An aauth URI identifies an agent instance and has the syntax aauth:local@domain, where local is the agent-specific part and domain is the agent server's domain name. The aauth URI is used in the sub claim of agent tokens, the agent field of resource tokens and mission objects, and the act.sub claim of auth tokens.

17. Implementation Status

Note: This section is to be removed before publishing as an RFC.

This section records the status of known implementations of the protocol defined by this specification at the time of posting of this Internet-Draft, and is based on a proposal described in [RFC7942]. The description of implementations in this section is intended to assist the IETF in its decision processes in progressing drafts to RFCs.

The following implementations are known:

* *TypeScript* — github.com/hellocoop/AAuth (https://github.com/hellocoop/AAuth). Organization: Hell. Coverage: agent token issuance, HTTP Message Signatures, resource token exchange, PS token endpoint. Level of maturity: exploratory.
* *Python* — github.com/christian-posta/aauth-full-demo (https://github.com/christian-posta/aauth-full-demo). Contact: Christian Posta. Coverage: agent-to-resource flows with Keycloak as AS. Level of maturity: exploratory.
* *Java (Keycloak SPI)* — github.com/christian-posta/keycloak-aauth-extension (https://github.com/christian-posta/keycloak-aauth-extension). Contact: Christian Posta. Coverage: AAuth access server extension for Keycloak 26.2.5. Level of maturity: exploratory.

18. Document History

Note: This section is to be removed before publishing as an RFC.

* draft-hardt-oauth-aauth-protocol-00
  - Initial draft. Replaces draft-hardt-aauth-protocol-02 (https://datatracker.ietf.org/doc/draft-hardt-aauth-protocol/02/); no technical changes.

19. Acknowledgments

The author would like to thank reviewers for their feedback on concepts and earlier drafts: Aaron Pareki, Christian Posta, Frederik Krogsdal Jacobsen, Jared Hanson, Karl McGuinness, Mark Hendrickson, Nate Barbettini, Scott Motte, Wils Dawson.

20. References

20.1. Normative References

[I-D.hardt-aauth-bootstrap]
Hardt, D., "AAuth Bootstrap", 2026, <https://github.com/dickhardt/AAuth>.

[I-D.hardt-httpbis-signature-key]
Hardt, D. and T. Meunier, "HTTP Signature Keys", 2026, <https://dickhardt.github.io/signature-key/draft-hardt-httpbis-signature-key.html>.

[I-D.ryan-httpauth-payment]
Ryan, B., Moxey, J., Meagher, T., Weinstein, J., and S. Kaliski, "The "Payment" HTTP Authentication Scheme", Work in Progress, Internet-Draft, draft-ryan-httpauth-payment-01, 17 March 2026, <https://datatracker.ietf.org/doc/html/draft-ryan-httpauth-payment-01>.

[OpenID.Core]
Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0", November 2014, <https://openid.net/specs/openid-connect-core-1_0.html>.

[RFC5890] Klensin, J., "Internationalized Domain Names for Applications (IDNA): Definitions and Document Framework", RFC 5890, DOI 10.17487/RFC5890, August 2010, <https://www.rfc-editor.org/info/rfc5890>.

[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>.

[RFC6979] Pornin, T., "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)", RFC 6979, DOI 10.17487/RFC6979, August 2013, <https://www.rfc-editor.org/info/rfc6979>.

[RFC7240] Snell, J., "Prefer Header for HTTP", RFC 7240, DOI 10.17487/RFC7240, June 2014, <https://www.rfc-editor.org/info/rfc7240>.

[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>.

[RFC7595] Thaler, D., Ed., Hansen, T., and T. Hardie, "Guidelines and Registration Procedures for URI Schemes", BCP 35, RFC 7595, DOI 10.17487/RFC7595, June 2015, <https://www.rfc-editor.org/info/rfc7595>.

[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>.

[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>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[RFC8615] Nottingham, M., "Well-Known Uniform Resource Identifiers (URIs)", RFC 8615, DOI 10.17487/RFC8615, May 2019, <https://www.rfc-editor.org/info/rfc8615>.

[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>.

[RFC8941] Nottingham, M. and P. Kamp, "Structured Field Values for HTTP", RFC 8941, DOI 10.17487/RFC8941, February 2021, <https://www.rfc-editor.org/info/rfc8941>.

[RFC9068] Bertocci, V., "JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens", RFC 9068, DOI 10.17487/RFC9068, October 2021, <https://www.rfc-editor.org/info/rfc9068>.

[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/info/rfc9110>.

[RFC9325] Sheffer, Y., Saint-Andre, P., and T. Fossati, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 9325, DOI 10.17487/RFC9325, November 2022, <https://www.rfc-editor.org/info/rfc9325>.

[RFC9421] Backman, A., Ed., Richer, J., Ed., and M. Sporny, "HTTP Message Signatures", RFC 9421, DOI 10.17487/RFC9421, February 2024, <https://www.rfc-editor.org/info/rfc9421>.

20.2. Informative References

[CommonMark] MacFarlane, J., "CommonMark Spec", 2024, <https://spec.commonmark.org/0.31.2/>.

[OpenID.Enterprise] Hardt, D. and K. McGuinness, "OpenID Connect Enterprise Extensions 1.0", 2025, <https://openid.net/specs/openid-connect-enterprise-extensions-1_0.html>.

[RFC5905] Mills, D., Martin, J., Ed., Burbank, J., and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification", RFC 5905, DOI 10.17487/RFC5905, June 2010, <https://www.rfc-editor.org/info/rfc5905>.

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

[RFC7565] Saint-Andre, P., "The 'acct' URI Scheme", RFC 7565, DOI 10.17487/RFC7565, May 2015, <https://www.rfc-editor.org/info/rfc7565>.

[RFC7591] Richer, J., Ed., Jones, M., Bradley, J., Machulak, M., and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol", RFC 7591, DOI 10.17487/RFC7591, July 2015, <https://www.rfc-editor.org/info/rfc7591>.

[RFC7636] Sakimura, N., Ed., Bradley, J., and N. Agarwal, "Proof Key for Code Exchange by OAuth Public Clients", RFC 7636, DOI 10.17487/RFC7636, September 2015, <https://www.rfc-editor.org/info/rfc7636>.

[RFC7942] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", BCP 205, RFC 7942, DOI 10.17487/RFC7942, July 2016, <https://www.rfc-editor.org/info/rfc7942>.

[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>.

[RFC8628] Denniss, W., Bradley, J., Jones, M., and H. Tschofenig, "OAuth 2.0 Device Authorization Grant", RFC 8628, DOI 10.17487/RFC8628, August 2019, <https://www.rfc-editor.org/info/rfc8628>.

[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>.

[RFC9530] Polli, R. and L. Pardue, "Digest Fields", RFC 9530, DOI 10.17487/RFC9530, February 2024, <https://www.rfc-editor.org/info/rfc9530>.

[RFC9635] Richer, J., Ed. and F. Imbault, "Grant Negotiation and Authorization Protocol (GNAP)", RFC 9635, DOI 10.17487/RFC9635, October 2024, <https://www.rfc-editor.org/info/rfc9635>.

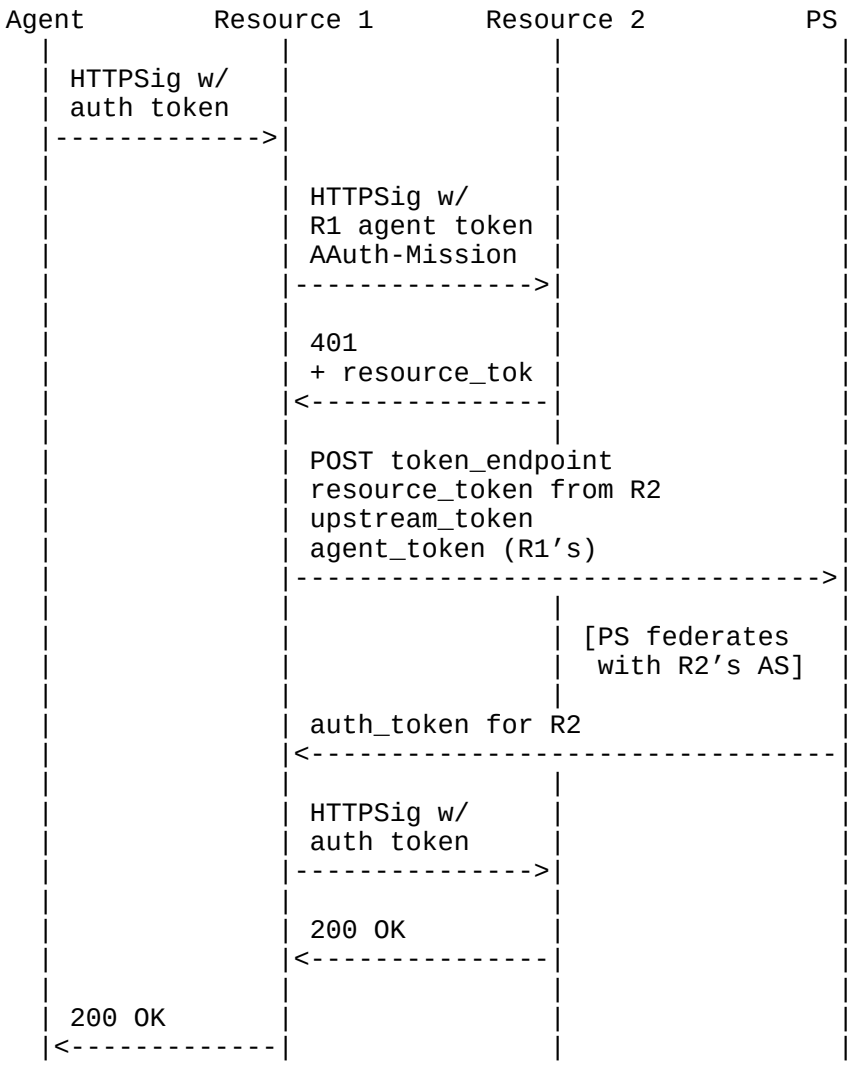
[x402] x402 Foundation, "x402: HTTP 402 Payment Protocol", 2025, <https://docs.x402.org>.

Appendix A. Detailed Flows

This appendix provides flow diagrams for the chaining patterns defined in the main specification, where the choreography is hard to follow from prose alone.

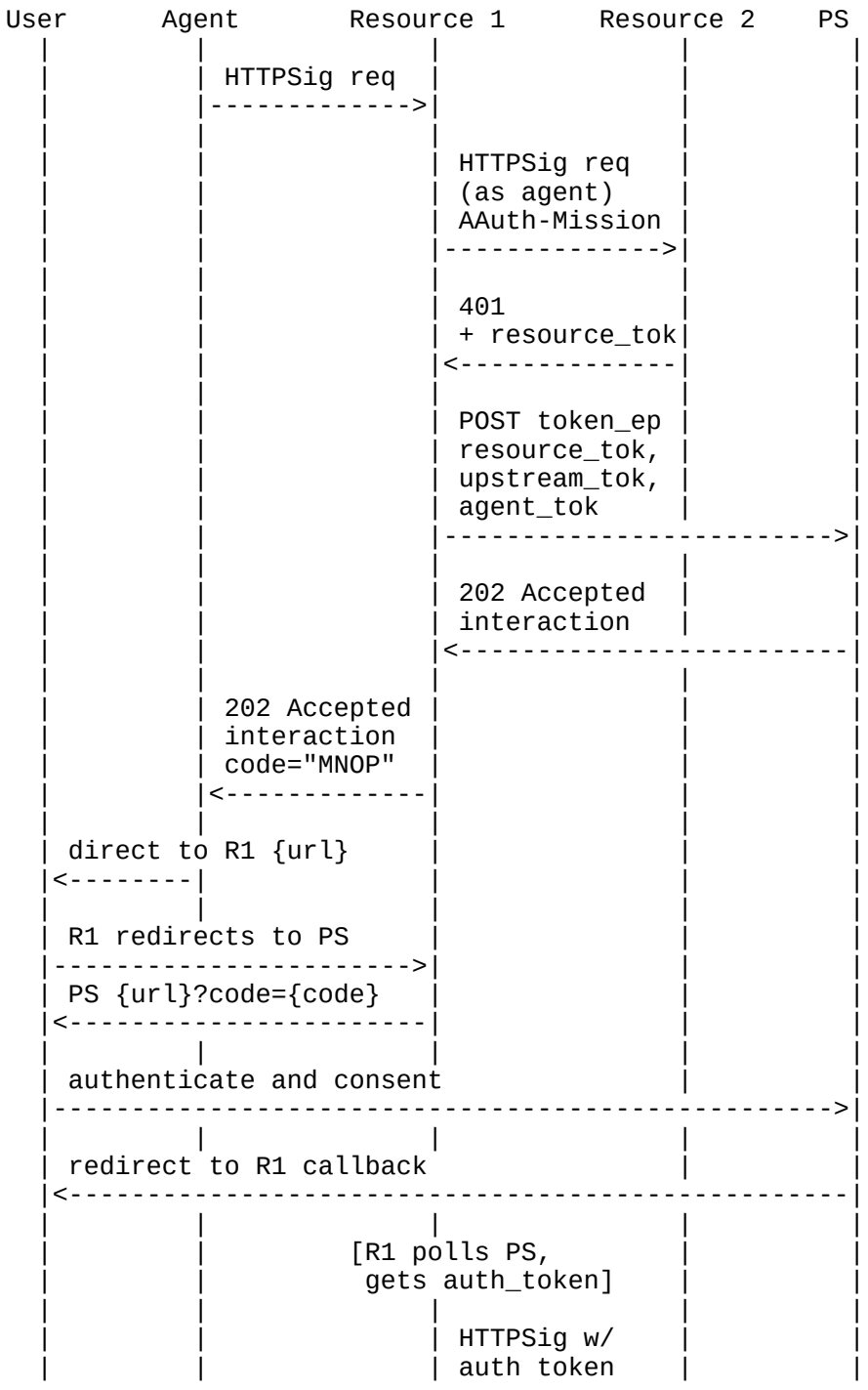
A.1. Four-Party: Call Chaining

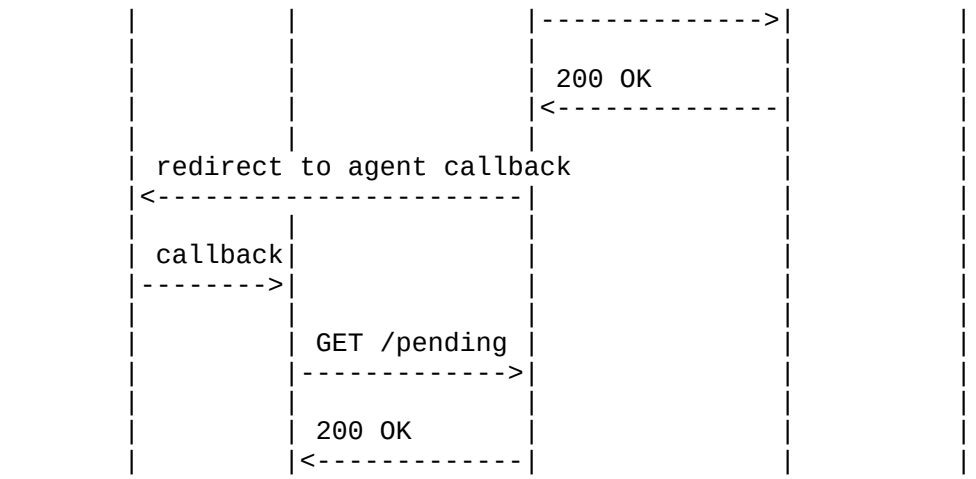
See Section 10.1 for normative requirements. Resource 1 acts as an agent, sending the downstream resource token plus its own agent token and the upstream auth token to the PS.



A.2. Interaction Chaining

See Section 10.2 for normative requirements. When the PS requires user interaction for the downstream access, Resource 1 chains the interaction back to the original agent.


User
Agent
Resource 1
Resource 2
PS
HTTPSig req
HTTPSig req
(as agent)
AAuth-Mission
401
+ resource_tok
POST token_ep
resource_tok,
upstream_tok,
agent_tok
202 Accepted
interaction
202 Accepted
interaction
code="MNOP"
direct to R1 {url}
R1 redirects to PS
PS {url}?code={code}
authenticate and consent
redirect to R1 callback
[R1 polls PS,
gets auth_token]
HTTPSig w/
auth token



Appendix B. Design Rationale

B.1. Identity and Foundation

B.1.1. Why HTTPS-Based Agent Identity

HTTPS URLs as agent identifiers enable dynamic ecosystems without pre-registration.

B.1.2. Why Per-Instance Agent Identity

OAuth's client_id identifies an application — every instance of the same app shares a single identifier and typically a single set of credentials. AAuth's aauth:local@domain agent identifier identifies a specific instance with its own signing key. This enables per-instance authorization (grant access to this specific agent process, not all instances of the app), per-instance revocation (revoke one compromised instance without affecting others), and per-instance audit (trace every action to the specific instance that performed it). The agent server controls which instances receive agent tokens, providing centralized governance over a distributed agent fleet.

B.1.3. Why Every Agent Has a Person

Every agent acts on behalf of a person — the entity accountable for the agent's actions. AAuth enables a person server to maintain this link, making it visible and enforceable across the protocol. When present, the PS ensures there is always an accountable party for authorization decisions, audit, and liability.

B.1.4. Why the ps Claim in Agent Tokens

Resources need to discover the agent's PS to issue resource tokens in three-party mode. The ps claim in the agent token provides this discovery without requiring the AAuth-Mission header, which is only present when the agent is operating within a mission. This separates PS discovery from mission governance — an agent can use three-party mode without missions.

B.2. Protocol Mechanics

B.2.1. Why .json in Well-Known URIs

AAuth well-known metadata URIs use the .json extension (e.g., /.well-known/aauth-agent.json) rather than the extensionless convention used by OAuth and OpenID Connect. The .json extension makes the content type immediately obvious — no content negotiation is needed. More importantly, it enables static file hosting: a .json file served from GitHub Pages, S3, or a CDN works without server-side configuration. This aligns with AAuth's self-hosted agent model (see [I-D.hardt-aauth-bootstrap]), where an agent's metadata can be published as static files with no active server.

B.2.2. Why Standard HTTP Async Pattern

AAuth uses standard HTTP async semantics (202 Accepted, Location, Prefer: wait, Retry-After). This applies uniformly to all endpoints, aligns with RFC 7240, replaces OAuth device flow, supports headless agents, and enables clarification chat.

B.2.3. Why JSON Instead of Form-Encoded

JSON is the standard format for modern APIs. AAuth uses JSON for both request and response bodies.

B.2.4. Why No Authorization Code

AAuth eliminates authorization codes entirely. OAuth authorization codes require PKCE ([RFC7636]) to prevent interception attacks, adding complexity for both clients and servers. AAuth avoids the problem: the user redirect carries only the callback URL, which has no security value to an attacker. The auth token is delivered exclusively via polling, authenticated by the agent's HTTP Message Signature.

B.2.5. Why Callback URL Has No Security Role

Tokens never pass through the user's browser. The callback URL is purely a UX optimization.

B.2.6. Why No Refresh Token

AAuth has no refresh tokens. When an auth token expires, the agent obtains a fresh resource token and submits it through the standard authorization flow. This gives the resource a voice in every re-authorization — the resource can adjust scope, require step-up authorization, or deny access based on current policy. A separate refresh token would bypass the resource entirely, and is unnecessary given that the standard flow is a single additional request.

B.2.7. Why Reuse OpenID Connect Vocabulary

AAuth reuses OpenID Connect scope values, identity claims, and enterprise parameters. This lowers the adoption barrier.

B.3. Architecture

B.3.1. Why a Separate Person Server

The PS is distinct from the AS because they serve different parties with different concerns. The PS represents the agent and its user — it handles consent, identity, mission governance, and audit. The AS represents the resource — it evaluates policy and issues tokens. Combining these into a single entity would conflate the interests of the requesting party with the interests of the resource owner, which is the same conflation that makes OAuth insufficient for cross-domain agent ecosystems.

B.3.2. Why Four Adoption Modes

The protocol supports identity-based, resource-managed (two-party), PS-managed (three-party), and federated (four-party) resource access modes, with agent governance as an orthogonal layer. A resource that only verifies agent signatures can start using AAuth today without deploying a PS or AS. As the ecosystem matures, the same resource can issue resource tokens to the agent's PS (three-party) and eventually deploy its own AS (four-party). Each mode is self-contained and useful — not a stepping stone to the "real" protocol. Agent governance (missions, permissions, audit) works independently of resource access modes.

B.3.3. Why Resource Tokens

In GNAP and OAuth, the resource server is a passive consumer of tokens — it verifies them but never produces signed artifacts. AAuth inverts this: the resource cryptographically asserts what is being requested by issuing a resource token that binds the resource's own identity, the agent's key thumbprint, the requested scope, and the mission context into a single signed JWT. This prevents confused deputy attacks — an attacker cannot substitute a different resource in the authorization flow because the resource token is signed by the resource. It also gives the resource a voice in every authorization and re-authorization, and provides a complete audit artifact linking the request to a specific resource, agent, scope, and mission.

B.3.4. Why Opaque AAuth-Access Tokens

In two-party mode, the resource returns an opaque wrapped token via the AAuth-Access header rather than a JWT auth token. This allows the resource to wrap its existing authorization infrastructure (OAuth access tokens, session tokens, etc.) without exposing internal structure. The token is bound to the AAuth signature — the agent includes it in the Authorization header as a covered component — so it cannot be stolen and replayed as a standalone bearer token.

B.3.5. Why Missions Are Not a Policy Language

Missions are intentionally not a machine-evaluable policy language. AAuth separates two kinds of authorization decisions:

* *Deterministic policy* is handled by scopes, resource tokens, and AS policy evaluation. These are mechanically evaluable — "does this agent have data.read scope for this resource?" A policy engine (Cedar, OPA/Rego, or any other) can answer this question consistently and automatically.

* *Contextual governance* is handled by missions, justifications, and clarification at the PS. These are the contextual decisions that policy engines cannot answer — "is booking a $10,000 flight reasonable for planning a weekend trip?" or "should this agent access the HR database given what it's trying to accomplish?" The mission description, the agent's justification for each resource access, and the clarification dialog between user and agent provide the context for these decisions.

Prior attempts to make authorization semantics machine-evaluable across domains have not scaled. OAuth Rich Authorization Requests (RAR) require clients and servers to agree on domain-specific type values and JSON structures — workable within a single API but

combinatorially explosive across arbitrary services. UMA attempted cross-domain resource sharing with machine-readable permission tickets, but adoption stalled because resource owners, requesting parties, and authorization servers could not converge on shared semantics for what permissions meant across organizational boundaries. The fundamental problem is that the meaning of "appropriate access" is contextual, evolving, and domain-specific — it cannot be captured in a predefined vocabulary that all parties share.

Missions solve this differently. Rather than requiring all parties to agree on machine-evaluable semantics, AAuth concentrates governance evaluation at the PS — the only party with full context. The PS has the mission description, the user's identity and organizational context, the agent's justification for each request, the history of what the agent has done so far, and a channel to the user for clarification. No other party in the protocol has this context, and no predefined policy language can substitute for it.

This context can be presented to humans or to agents acting as decision-makers. The PS does not need to evaluate missions deterministically — it presents the mission context, the justification, and the resource request to whatever decision-maker is appropriate: a human reviewing a consent screen, an AI agent evaluating policy on behalf of an organization, or an automated system applying heuristics. As AI decision-making matures, governance can shift from human review to agent evaluation — without changing the protocol. AAuth standardizes how context is conveyed to the decision-maker; it does not prescribe how the decision is made.

The mission's description is Markdown because it represents human intent, not machine policy. The approved_tools array provides structured machine-evaluable elements where appropriate. Resources and access servers do not need the mission content — they enforce their own deterministic policies independently. The mission is a further restriction applied by the PS, and only the PS has sufficient context to evaluate it. Distributing mission semantics to other parties would be both a privacy leak and a false promise of enforcement, since those parties lack the context to evaluate the mission meaningfully.

B.3.6. Why Missions Have Only Two States

Missions are either *active* or *terminated*. There is no suspended state. A suspended state would require the agent to learn that the mission has resumed, but AAuth has no push channel from the PS to the agent — the agent can only poll. For short pauses (minutes), the deferred response mechanism already provides natural waiting via 202 polling. For long pauses (hours or more), the agent would need to poll indefinitely with no indication of when to stop, making suspension operationally equivalent to termination. Terminating the mission and creating a new one is cleaner — the PS retains the old mission's log for audit, and the new mission can be scoped appropriately for the changed circumstances that prompted the pause. This keeps mission lifecycle simple: a mission is alive until it is done.

B.3.7. Why Downstream Scope Is Not Constrained by Upstream Scope

In multi-hop scenarios, downstream authorization is intentionally not required to be a subset of upstream scopes. A flight booking API that calls a payment processor needs the payment processor to charge a card — an operation orthogonal to the upstream scope. Formal subset rules would prevent legitimate delegation chains. Instead, the PS evaluates each hop against the mission context, providing governance-based constraints that are more flexible than algebraic attenuation rules while maintaining a complete audit trail.

B.4. Comparisons with Alternatives

B.4.1. Why Not mTLS?

Mutual TLS (mTLS) authenticates the TLS connection, not individual HTTP requests. Different paths on the same resource may have different requirements — some paths may require no signature, others a signed request, others verified identity, and others an auth token. Per-request signatures allow resources to vary requirements by path. Additionally, mTLS requires PKI infrastructure (CA, certificate provisioning, revocation), cannot express progressive requirements, and is stripped by TLS-terminating proxies and CDNs. mTLS remains the right choice for infrastructure-level mutual authentication (e.g., service mesh). AAuth addresses application-level identity where progressive requirements and intermediary compatibility are needed.

B.4.2. Why Not DPoP?

DPoP ([RFC9449]) binds an existing OAuth access token to a key, preventing token theft. AAuth differs in that agents can establish identity from zero — no pre-existing token, no pre-registration. At the signature level ([I-D.hardt-httpbis-signature-key]), AAuth requires no tokens at all, only a signed request. DPoP has a single mode (prove you hold the key bound to this token), while AAuth supports progressive requirements from pseudonymous access through verified identity to authorized access with interactive consent. DPoP is the right choice for adding proof-of-possession to existing OAuth deployments.

B.4.3. Why Not Extend GNAP

GNAP ([RFC9635]) shares several motivations with AAuth — proof-of-possession by default, client identity without pre-registration, and async authorization. A natural question is whether AAuth's capabilities could be achieved as GNAP extensions rather than a new protocol. There are several reasons they cannot.

Resource tokens require an architectural change, not an extension. In GNAP, as in OAuth, the resource server is a passive consumer of tokens — it verifies them but never produces signed artifacts that the access server consumes. AAuth's resource tokens invert this: the resource cryptographically asserts what is being requested, binding its own identity, the agent's key thumbprint, and the requested scope into a signed JWT. Adding this to GNAP would require changing its core architectural assumption about the role of the resource server.

Interaction chaining requires a different continuation model. GNAP's continuation mechanism operates between a single client and a single access server. When a resource needs to access a downstream resource that requires user consent, GNAP has no mechanism for that consent requirement to propagate back through the call chain to the original user. Supporting this would require rethinking GNAP's continuation model to support multi-party propagation through intermediaries.

The federation model is fundamentally different. In GNAP, the client must discover and interact with each access server directly. AAuth's model — where the agent only ever talks to its PS, and the PS federates with resource ASes — is a different trust topology, not a configuration option. Retrofitting this into GNAP would produce a profile so constrained that it would be a distinct protocol in practice.

GNAP's generality is a liability for this use case. GNAP is designed to be maximally flexible — interaction modes, key proofing methods, token formats, and access structures are all pluggable. This means implementers must make dozens of profiling decisions before arriving at an interoperable system. AAuth makes these decisions prescriptively: one token format (JWT), one key proofing method (HTTP Message Signatures), one interaction pattern (interaction codes with polling), and one identity model (local@domain with HTTPS metadata). For the agent-to-resource ecosystem, this prescriptiveness is a feature — it enables interoperability without bilateral agreements.

In summary, AAuth's core innovations — resource-signed challenges, interaction chaining through multi-hop calls, PS-to-AS federation, mission-scoped authorization, and clarification chat during consent — are architectural choices that would require changing GNAP's foundations rather than extending them. The result would be a heavily constrained GNAP profile that shares little with other GNAP deployments.

B.4.4. Why Not Extend WWW-Authenticate?

WWW-Authenticate ([RFC9110], Section 11.6.1) tells the client which authentication scheme to use. Its challenge model is "present credentials" — it cannot express progressive requirements, authorization, or deferred approval, and it cannot appear in a 202 Accepted response.

AAuth-Requirement and Accept-Signature coexist with WWW-Authenticate. A 401 response MAY include multiple headers, and the client uses whichever it understands:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="api"
Accept-Signature: sig=("@method" "@authority" "@path");sigkey=uri
```

A 402 response MAY include WWW-Authenticate for payment (e.g., the Payment scheme defined by the Micropayment Protocol ([I-D.ryan-httpauth-payment])) alongside Accept-Signature for authentication or AAuth-Requirement for authorization:

```
HTTP/1.1 402 Payment Required
WWW-Authenticate: Payment id="x7Tg2pLq", method="example",
    request="eyJhbW91bnQiOiIxMDAw..."
Accept-Signature: sig=("@method" "@authority" "@path");sigkey=jkt
```

B.4.5. Why Not Extend OAuth?

OAuth 2.0 ([RFC6749]) was designed for delegated access — a user authorizes a pre-registered client to act on their behalf at a specific server. Extending OAuth for agent-to-resource authorization would require changing its foundational assumptions:

* *Client identity*: OAuth clients have no independent identity. A client_id is issued by each authorization server — it is meaningless outside that relationship. AAuth agents have self-sovereign identity (aauth:local@domain) verifiable by any party.
* *Pre-registration*: OAuth requires clients to register with each authorization server before use. AAuth agents call resources they have never contacted before — the first API call is the registration.
* *Bearer tokens*: OAuth access tokens are bearer credentials — anyone who holds the token can use it. AAuth binds every token to a signing key via HTTP Message Signatures — a stolen token is useless without the private key.
* *No resource identity*: OAuth does not cryptographically identify the resource. AAuth resources sign resource tokens, binding their identity to the authorization flow.
* *No governance layer*: OAuth has no concept of missions, permission endpoints, audit logging, or interaction relay. These would need to be built on top as extensions, losing the coherence of a protocol designed around them.
* *No federation model*: OAuth's authorization server is always the resource owner's server. AAuth separates the person server (user's choice) from the access server (resource's choice) and defines how they federate.

The Model Context Protocol (MCP) illustrates these limitations. MCP adopted OAuth 2.1 for agent-to-server authorization and immediately needed Dynamic Client Registration ([RFC7591]) because agents cannot pre-register with every server. But Dynamic Client Registration gives the agent a different client_id at each server — the agent still has no portable identity. Tokens are bearer credentials, so a stolen token grants full access. There is no resource identity — the server does not cryptographically prove who it is. There is no governance layer — no missions, no permission management, no audit trail. And the entire authorization model is per-server: each MCP server has its own authorization server, and the agent must discover and register with each one independently. MCP's experience demonstrates that OAuth can be made to work for the first API call, but it cannot provide the identity, governance, and federation that agents need as they operate across trust domains.

Rather than layer these changes onto OAuth — which would break backward compatibility and produce something unrecognizable — AAuth is a new protocol designed for the agent model from the ground up. AAuth complements OAuth: resources can wrap existing OAuth infrastructure behind the AAuth-Access token, and PSes can delegate user authentication to OpenID Connect providers.

Author's Address

Dick Hardt
Hell
Email: dick.hardt@gmail.com