TBD
Internet-Draft
Intended status: Standards Track
Expires: 4 October 2026

D. Hardt
Hell

2 April 2026

AAuth Protocol
draft-hardt-aauth-protocol-00

Abstract

This document defines the AAuth authorization protocol, in which agents obtain proof-of-possession tokens from auth servers to access resources on behalf of users and organizations. It specifies three token types (agent, resource, and auth), a unified token endpoint with deferred response support, and cross-domain federation between auth servers. It builds on the AAuth Headers specification ([I-D.hardt-aauth-headers]), which defines the AAuth-Requirement response header and HTTP Message Signatures profile.

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

Table of Contents

1. Introduction

OAuth 2.0 [RFC6749] was created to solve a security problem: users were sharing their passwords with third-party web applications so those applications could access their data at other sites. OAuth replaced this anti-pattern with a delegation model — the user's browser redirects to the authorization server, the user consents, and the application receives an access token without ever seeing the user's credentials. OpenID Connect extended this to federated login.

But the landscape has changed. New use cases have emerged that OAuth and OIDC were not designed to address:

* *On-demand authorization* where agents do not know what resources they will require until runtime. Long-running agents may execute tasks over hours or days and discover new authorization needs as they progress.
* *Multi-hop resource access* where a resource needs to obtain authorization to access a downstream resource to fulfill a request, with interaction requirements bubbling back to the user through the chain.
* *Cross-domain trust* where agents and resources have different auth servers. In OAuth, the client and resource share the same authorization server. In dynamic ecosystems, agents routinely access resources governed by a different auth server.
* *Authorization negotiation* where the user and agent engage in a back-and-forth during consent — the user asks why access is needed, the agent explains or adjusts its request — rather than a binary approve/deny decision.

AAuth introduces the following features to address these use cases:

* *Agent identity without pre-registration*: HTTPS URLs with self-published metadata and JWKS enable agents to establish identity without registering credentials at each authorization server.
* *Per-instance agent identity*: Each agent instance has its own identifier (local@domain) and signing key. Authorization grants are per-instance, not per-application.

* *Resource identity and resource-defined authorization*: Resources issue signed challenges binding the request to the resource's identity and the agent's key, defining authorization requirements at request time. This decouples resources from auth servers, and prevents MITM and confused deputy attacks.
* *Multi-hop resource access*: A resource acts as an agent to access downstream resources, with interaction requirements bubbling back to the user.
* *AS-to-AS federation*: An agent's auth server can call a resource's auth server to obtain an auth token on behalf of its agent, enabling cross-domain access without the agent or resource being aware of the federation.
* *Deferred responses*: 202 Accepted with polling is a first-class primitive across all endpoints, supporting headless agents, long-running consent, and clarification chat.
* *Clarification chat with justification*: Agents declare why access is needed, and users can ask questions during consent. The agent can explain or adjust its request.

AAuth also provides enhancements over OAuth:

* *Proof-of-possession by default*: OAuth bearer tokens can be stolen and replayed by any holder. AAuth binds every token to a signing key via HTTP Message Signatures.
* *Unified authentication and authorization*: OAuth and OIDC are separate protocols with separate flows and token types. AAuth uses a single auth token that carries both identity claims and authorized scopes.
* *No protocol artifacts in browser redirects*: Unlike OAuth, where browser redirects carry authorization codes that are vulnerable to interception, AAuth uses browser redirects only to transition the user between parties.
* *Reuse of OpenID Connect vocabulary*: AAuth reuses OpenID Connect scope values, identity claims, and enterprise extensions, lowering the adoption barrier.

AAuth complements OAuth and OIDC rather than replacing them — where pre-registered clients, browser redirects, bearer tokens, and static scopes work well, they remain the right choice. The AAuth Header specification ([I-D.hardt-aauth-headers]) defines how resources communicate authentication requirements via the AAuth-Requirement header and how agents present cryptographic identity using HTTP Message Signatures. This specification builds on that foundation to define the authorization protocol — how agents obtain auth tokens from auth servers to access protected resources.

2. Conventions and Definitions

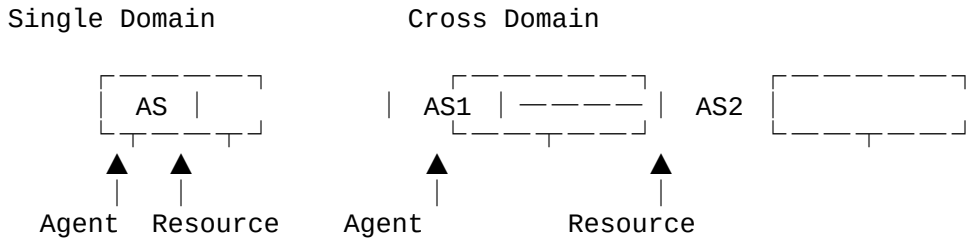
{::boilerplate bcp14-tagged}

3. Terminology

* *Agent*: An HTTP client ([RFC9110], Section 3.5) acting on behalf of a legal person (user or organization). Identified by an agent identifier of the form local@domain Section 8.2. An agent has exactly one auth server that it sends all token requests to.
* *Agent Server*: A server that manages agent identity and issues agent tokens to agents. Identified by an HTTPS URL Section 8.1 and publishes metadata at /.well-known/aauth-agent.json.
* *Agent Token*: A JWT issued by an agent server to an agent, binding the agent's signing key to the agent's identity Section 9.
* *Auth Server*: A server that authenticates users, obtains consent, evaluates authorization policies, and issues auth tokens. The auth server maintains the association between agents and their legal persons. Identified by an HTTPS URL Section 8.1 and publishes metadata at /.well-known/aauth-issuer.json.
* *Auth Token*: A JWT issued by an auth server that grants an agent access to a resource, containing user identity and/or authorized scopes Section 11.
* *Resource*: A server that requires authentication and/or authorization to protect access to its APIs and data. Identified by an HTTPS URL Section 8.1 and publishes metadata at /.well-known/aauth-resource.json. A resource has exactly one auth server that it accepts auth tokens from.
* *Resource Token*: A JWT issued by a resource binding the agent's identifier (sub) and key thumbprint to the resource's auth server (aud) Section 10.
* *Interaction*: User authentication, consent, or other action at an interaction endpoint. Triggered when a server returns 202 Accepted with requirement=interaction.
* *Markdown String*: A human-readable text value formatted as Markdown (CommonMark). Fields of this type MAY define recommended sections. Implementations MUST sanitize Markdown before rendering to users.
* *Justification*: A Markdown string provided by the agent declaring why access is needed, presented to the user by the auth server during consent. *TODO:* Define recommended sections.
* *Clarification*: A Markdown string containing a question posed to the agent by the user during consent via the auth server. The agent may respond with an explanation or an updated request.
* *Assessment*: A Markdown string containing an auth server's evaluation of a request during cross-domain federation, conveyed from AS2 to AS1. *TODO:* Define recommended sections.

4. Trust Model

Agents trust their auth server. Resources trust their auth server. When they are in different trust domains, their auth servers federate.



5. Bootstrapping

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

5.1. Entity Metadata

Each entity publishes metadata at a well-known URL:

* Agent servers publish at /.well-known/aauth-agent.json — including JWKS URI, display name, and capabilities Section 14.1.
* Auth servers publish at /.well-known/aauth-issuer.json — including token endpoint and supported scopes Section 14.2.
* Resources publish at /.well-known/aauth-resource.json — including auth server, required scopes, and resource token endpoint Section 14.3.

5.2. Agent Identity

An agent obtains an agent token from its agent server. The agent token binds the agent's signing key to its agent identifier (local@domain). See Appendix A for common provisioning patterns.

5.3. Auth Server Association

An agent has exactly one auth server that it sends all token requests to. How the agent learns its auth server is out of scope — this is determined by configuration during agent setup (e.g., set by the agent server or chosen by the person deploying the agent).

5.4. Person-Agent Association

The auth server maintains the association between an agent and its legal person (user or organization). This association is typically established when the person first authorizes the agent at the auth server via the interaction flow. An organization administrator may also pre-authorize agents for the organization.

The auth server 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.

6. Protocol Overview

6.1. Tokens

AAuth defines three proof-of-possession token types, all JWTs bound to a specific signing key:

* *Agent Token* (agent+jwt): Issued by an agent server to an agent, binding the agent's key to its identity Section 9.
* *Resource Token* (resource+jwt): Issued by a resource in response to a request, binding the access challenge to the resource's identity Section 10.
* *Auth Token* (auth+jwt): Issued by an auth server, granting an agent access to a specific audience Section 11.

6.2. Agent Identity

Every agent holds an agent token issued by its agent server. The agent token binds the agent's signing key to its agent identifier. The agent's auth server maintains the association between the agent and its legal person.

The agent always presents its agent token via the Signature-Key header when calling its auth server's token endpoint.

6.3. Scopes

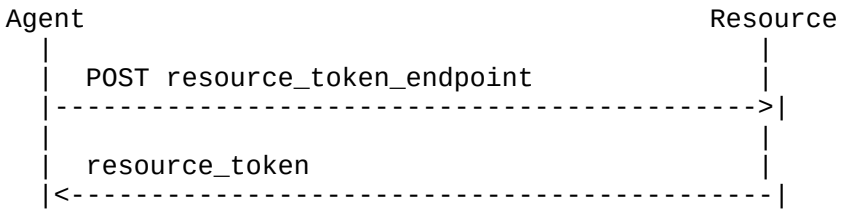
AAuth reuses the scope and claims vocabulary defined by OpenID Connect. Scopes may request identity claims (using OpenID Connect Core 1.0 [OpenID.Core] scope values such as openid, profile, email) or resource authorization (using scopes defined by the resource, such as data.read or calendar.write as defined in the resource's scope_descriptions metadata), or both.

6.4. Protocol Steps

This section describes the fundamental protocol steps. Detailed end-to-end flows combining these steps are in Appendix B.

6.4.1. Obtaining a Resource Token

When the agent knows the resource's requirements (from metadata or a previous request), it requests a resource token directly from the resource's resource_token_endpoint:

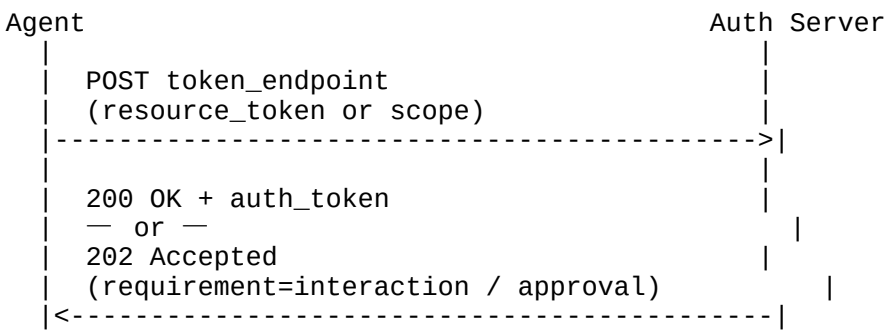


Alternatively, a resource MAY respond to any request with 401 and an AAuth-Requirement containing a resource token, indicating what authorization is required. This is the discovery path when the agent does not know the resource's requirements in advance.

A resource MAY also return 401 with a new resource token to a request that 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.

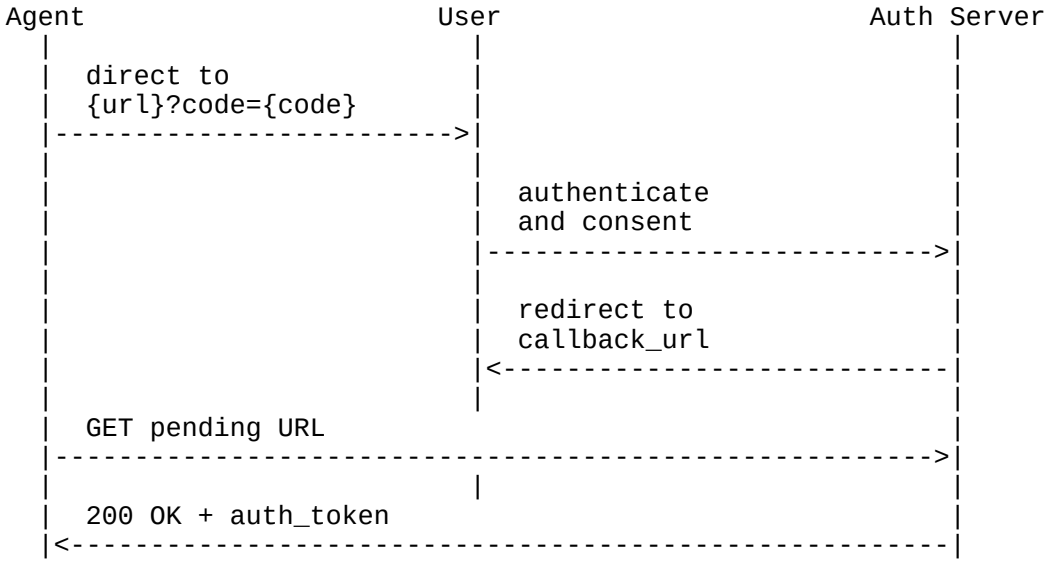
6.4.2. Obtaining an Auth Token

The agent presents a resource token (or scope for agent-as-audience) to its auth server's token endpoint. The auth server evaluates policy and returns an auth token immediately, or a 202 if user authorization is required.



6.4.3. Obtaining Authorization

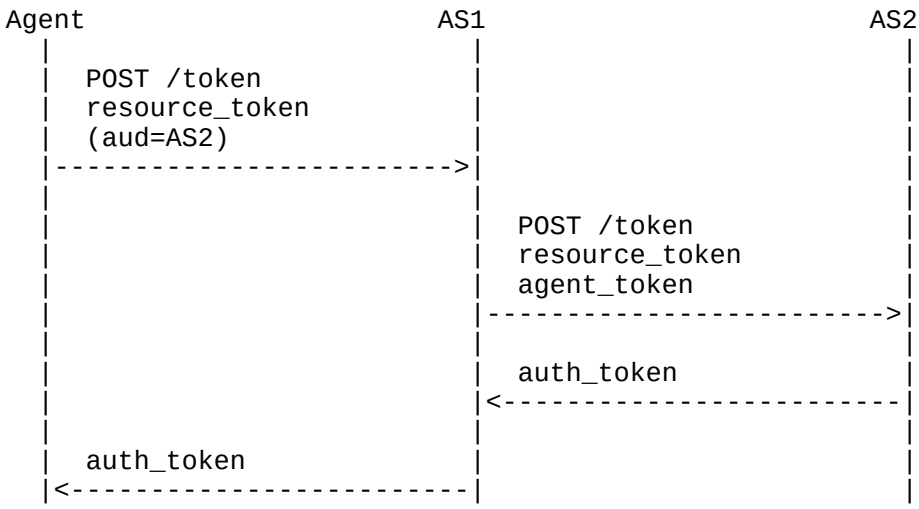
When the auth server requires user consent or authentication, it returns a 202 with an interaction url and code. The agent directs the user to {url}?code={code}. After the user completes the action, the agent polls for the result.



When the auth server can obtain authorization directly from the user without the agent's involvement Section 5.4, it returns requirement=approval and the agent simply polls.

6.4.4. Cross-Domain Federation

When the resource's auth server differs from the agent's, the agent's auth server (AS1) federates with the resource's auth server (AS2). The agent is unaware of the federation.



7. AAuth-Requirement Requirement Levels

This document defines the following requirement level for the AAuth-Requirement response header ([I-D.hardt-aauth-headers]). These levels extend the pseudonym and identity levels defined by the header specification.

7.1. Auth Token Required

When a resource requires an auth token, it responds with 401 Unauthorized and includes the AAuth-Requirement header with a resource token:

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

The agent presents the resource token to its auth server’s token endpoint to obtain an auth token. See Section 10 and Section 13 for details. A resource MAY also use 402 Payment Required with the same AAuth-Requirement header when payment is additionally required (see draft-hardt-aauth-headers).

When the auth server requires user interaction to complete an authorization request (e.g., authentication, consent), it returns 202 Accepted with requirement=interaction, a url, and a code ([I-D.hardt-aauth-headers]). The agent directs the user to the interaction URL with the code. See Section 13.5 for details.

When the auth server can obtain approval without the agent directing a user — for example, by contacting the user directly (push notification, email), or obtaining administrator approval — it returns 202 Accepted with requirement=approval. This is the recommended flow once the auth server has established a direct communication channel with the user Section 5.4. The agent polls the pending URL until a terminal response is received. See Section 12 for details.

8. Identifier and URL Requirements

8.1. Server Identifiers

The agent, resource, and issuer values that identify agents, resources, and auth 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.

8.2. Agent Identifiers

Agent identifiers are of the form 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 above (without scheme).

Valid agent identifiers:

* assistant-v2@agent.example
* 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).

8.3. Endpoint URLs

The token_endpoint, resource_token_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.

8.4. Other URLs

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

9. Agent Tokens

Agent tokens bind an agent's signing key to its identity.

9.1. Agent Token Structure

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

Header: - alg: Signing algorithm. EdDSA is RECOMMENDED. Implementations MUST NOT accept none. - typ: 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 and audit - 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: - aud: Audience restriction. When present, the agent MUST only present this agent token to the specified server(s). - aud_sub: The user identifier (sub value) from a previous auth token issued by the auth server in aud. This is a hint to the auth server to identify which user the agent is associated with. The auth server MUST NOT treat this claim as authoritative — the auth server maintains its own person-agent associations and uses aud_sub only to optimize lookup.

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

9.2. 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="eyJhbGciOiJFZERTQSIsInR5cCI6ImFnZW50K2p3dCJ9..."
```

10. Resource Tokens

Resource tokens provide cryptographic proof of resource identity, preventing confused deputy and MITM attacks.

10.1. Resource Token Structure

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

Header: - alg: Signing algorithm. EdDSA is RECOMMENDED. Implementations MUST NOT accept none. - typ: 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: Auth server URL - jti: Unique token identifier for replay detection and audit - 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 (optional), as a space-separated string of scope values

Resource tokens SHOULD NOT have a lifetime exceeding 5 minutes. Resource tokens are single-use; the auth server MUST reject a resource token whose jti has been seen before.

10.2. Resource Token Usage

Resources include resource tokens in the AAuth-Requirement header when requiring authorization:

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

10.3. Resource Token Endpoint

When a resource publishes a resource_token_endpoint in its metadata, agents MAY request a resource token proactively — without first making an API call and receiving a 401 challenge.

Request:

```
POST /resource-token HTTP/1.1
Host: resource.example
Content-Type: application/json
Signature-Input: sig=("@method" "@authority" "@path" "signature-key");created=17302176
00
Signature: sig=:...signature bytes...:
Signature-Key: sig=jwt;jwt="eyJhbGc..."

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

Response:

```
{
  "resource_token": "eyJhbGc...",
  "scope": "data.read data.write"
}
```

The resource's auth server is identified by the aud claim in the resource token. The agent sends the resource token to its own auth server's token endpoint.

10.4. Resource Token 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 1

Error responses use the same format as the token endpoint Section 17.2.

11. Auth Tokens

Auth tokens grant agents access to resources after authentication and authorization.

11.1. Auth Token Structure

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

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

Required payload claims: - iss: Auth server URL - dwk: aauth-issuer.json — the well-known metadata document name for key discovery ([I-D.hardt-httpbis-signature-key]) - aud: The URL of the resource the agent is authorized to access. When the agent is accessing its own resources (SSO or first-party use), the aud is the agent server's URL. - jti: Unique token identifier for replay detection and audit -

agent: Agent identifier - cnf: Confirmation claim with jwk containing the agent's public key - iat: Issued at timestamp - exp: Expiration timestamp. Auth tokens SHOULD NOT have a lifetime exceeding 1 hour and MUST NOT have a lifetime exceeding 24 hours.

Conditional payload claims (at least one MUST be present): - sub: User identifier - 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.

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.

11.2. 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="eyJhbGciOiJFZERTQSIsInR5cCI6ImF1dGgrand0In0..."
```

12. Deferred Responses

Any endpoint in AAuth — whether an auth server token endpoint or a 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.

12.1. 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=17302176
00
Signature: sig=:...signature bytes...:
Signature-Key: sig=jwt;jwt="eyJhbGc..."

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

12.2. 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",
  "location": "/pending/f7a3b9c"
}
```

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.

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.
* location (REQUIRED): The pending URL (echoes the Location header).
* requirement (OPTIONAL): "interaction" when the agent must direct the user to an interaction endpoint (with code). "approval" when the auth server is obtaining approval directly.
* code (OPTIONAL): The interaction code. Present only with requirement: "interaction".
* clarification (OPTIONAL): A question from the user during consent.

12.3. 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. Terminal Responses

A non-202 response terminates polling. The following table covers responses to both the initial request and subsequent GET polling requests at any AAuth endpoint:

Status	Meaning	Agent Behavior
200	Success	Process response body
400	Invalid request	Check error field; fix and retry
401	Invalid signature / auth token required	Check credentials; obtain auth token if resource challenge
402	Auth token + payment required	Obtain auth token and satisfy payment requirement
403	Denied or abandoned	Surface to user; check error field
408	Expired	MAY initiate a fresh request
410	Gone — permanently invalid	MUST NOT retry
429	Too many requests	Increase polling interval by 5 seconds
500	Internal server error	Start over
503	Temporarily unavailable	Back off per Retry-After, retry

Table 2

12.5. Deferred Response State Machine

The following state machine applies to any AAuth endpoint that returns a 202 Accepted response — including auth server token endpoints and resource endpoints during call chaining.

```
Initial request (with Prefer: wait=N)
    |
    +-- 200 --> done (process response)
    +-- 202 --> note Location URL, check require/code
    +-- 400 --> invalid request — check error field, fix and retry
    +-- 401 --> invalid signature — check credentials
    +-- 402 --> auth token + payment required (resource only)
    +-- 500 --> server error — start over
    +-- 503 --> back off (Retry-After), retry
                |
                GET Location (with Prefer: wait=N)
                |
                +-- 200 --> done (process response)
                +-- 202 --> continue polling (check status and clarification)
                |           status=interacting → stop prompting user
                +-- 403 --> denied or abandoned — surface to user
                +-- 408 --> expired — MAY retry with fresh request
                +-- 410 --> invalid_code — do not retry
                +-- 429 --> slow_down — increase interval by 5s
                +-- 500 --> server_error — start over
                +-- 503 --> temporarily_unavailable — back off (Retry-After)
```

13. Token Endpoint

The auth server's token_endpoint issues auth tokens to agents and to other auth servers during cross-domain federation.

13.1. Token Endpoint Modes

Mode	Key Parameters	Use Case
Resource access	resource_token	Agent needs auth token for a resource
Self-access (SSO/1P)	scope (no resource_token)	Agent needs auth token for itself
Call chaining	resource_token + upstream_token	Resource acting as agent
AS-to-AS	resource_token +	Auth server federating on

federation	agent_token + optional upstream_token	behalf of agent (includes upstream_token when call chaining cross-domain)
Token refresh	auth_token (expired)	Renew expired token

Table 3

13.2. Authorization Request

The agent MUST make a signed POST to the token_endpoint. The request MUST include HTTP Message Signatures and the agent MUST present its agent token via the Signature-Key header using scheme=jwt.

Request parameters:

* resource_token (CONDITIONAL): The resource token. Required when requesting access to another resource.
* scope (CONDITIONAL): Space-separated scope values. Used when the agent requests authorization to itself.
* upstream_token (OPTIONAL): An auth token from an upstream authorization, used in call chaining.
* agent_token (OPTIONAL): The agent's agent token. Used in AS-to-AS federation.
* justification (OPTIONAL): A Markdown string declaring why access is being requested. The auth server SHOULD present this value to the user during consent. The auth server MUST sanitize the Markdown before rendering to users. The auth server 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: auth.example
   Content-Type: application/json
   Prefer: wait=45
   Signature-Input: sig=("@method" "@authority" "@path" "signature-key");created=17302176
00
   Signature: sig=:...signature bytes...:
   Signature-Key: sig=jwt;jwt="eyJhbGc..."

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

13.3. Auth Server Response

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; code="ABCD1234"
   Content-Type: application/json

   {
     "status": "pending",
     "location": "/pending/abc123",
     "require": "interaction",
     "code": "ABCD1234"
   }
```

13.4. Clarification Chat

During user consent, the user may ask questions about the agent's stated justification. The auth server 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 SHOULD declare "clarification_supported": true in their agent server metadata. Individual requests MAY indicate clarification support by including "clarification_supported": true in the token endpoint request body.

13.4.1. Clarification Flow

When the user asks a question during consent, the auth server includes a clarification field in the next polling response:

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

- clarification (String): The user's question.
- timeout (Integer, OPTIONAL): Seconds until the auth server times out the user interaction. The agent MUST respond before this deadline.

13.4.2. 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.

13.4.2.1. Clarification Response

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

```
POST /pending/abc123 HTTP/1.1
Host: auth.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.

13.4.2.2. 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: auth.example
Content-Type: application/json
Signature-Input: sig=("@method" "@authority" "@path" "signature-key");created=17302176
00
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 auth server presents the updated request to the user. A justification is OPTIONAL but RECOMMENDED to explain the change to the user.

13.4.2.3. Cancel Request

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

```
DELETE /pending/abc123 HTTP/1.1
Host: auth.example
Signature-Input: sig=("@method" "@authority" "@path" "signature-key");created=17302176
00
Signature: sig=:...signature bytes...:
Signature-Key: sig=jwt;jwt="eyJhbGc..."
```

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

13.4.3. Clarification Limits

Auth servers 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.

13.5. 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 ([I-D.hardt-aauth-headers]). The agent constructs the user-facing URL as {url}?code={code} and directs the user using one of the methods defined in the header specification (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.

13.6. Third-Party Initiated Login

When a third party directs a user to the agent's login_endpoint, the agent initiates a standard "agent as audience" login flow.

Login endpoint parameters:

* issuer (REQUIRED): The auth server URL.
* domain_hint (OPTIONAL): Domain hint.
* tenant (OPTIONAL): Tenant identifier.
* start_path (OPTIONAL): Path on the agent's origin where the user should be directed after login completes.

13.7. Token Refresh

When an auth token expires, the agent requests a new one by presenting the expired auth token:

```
   POST /token HTTP/1.1
   Host: auth.example
   Content-Type: application/json
   Signature-Input: sig=("@method" "@authority" "@path" "signature-key");created=17302176
00
   Signature: sig=:...signature bytes...:
   Signature-Key: sig=jwt;jwt="eyJhbGc..."

   {
     "auth_token": "eyJhbGc..."
   }
```

The auth server verifies the agent's HTTP signature, validates the expired auth token, and issues a new auth token. The refreshed auth token MUST have the same aud, scope, and sub claims as the original.

The agent's signing key MAY have rotated since the original auth token was issued. The auth server identifies the agent by the agent identifier in the agent token presented via Signature-Key on the refresh request.

The auth server determines the maximum time after expiration that a token may be refreshed. The refresh request MAY return a 202 deferred response if user interaction or approval is required to reauthorize.

14. Metadata Documents

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

14.1. Agent Server Metadata

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

```
{
  "agent": "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",
  "login_endpoint": "https://agent.example/login",
  "callback_endpoint": "https://agent.example/callback",
  "localhost_callback_allowed": true,
  "clarification_supported": true,
  "tos_uri": "https://agent.example/tos",
  "policy_uri": "https://agent.example/privacy"
}
```

Fields:

* agent (REQUIRED): The agent server's HTTPS URL (the domain in agent identifiers it issues)
* 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
* login_endpoint (OPTIONAL): URL where third parties direct users to initiate a login flow
* callback_endpoint (OPTIONAL): The agent's HTTPS callback endpoint URL
* localhost_callback_allowed (OPTIONAL): Boolean. Default: false.
* clarification_supported (OPTIONAL): Boolean. Default: false.
* tos_uri (OPTIONAL): URL to terms of service (per [RFC7591])
* policy_uri (OPTIONAL): URL to privacy policy (per [RFC7591])

14.2. Auth Server Metadata

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

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

Fields:

* issuer (REQUIRED): The auth server's HTTPS URL
* token_endpoint (REQUIRED): Single endpoint for all agent-to-auth-server communication
* jwks_uri (REQUIRED): URL to the auth server's JSON Web Key Set

14.3. Resource Metadata

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

```
{
  "resource": "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",
  "resource_token_endpoint": "https://resource.example/resource-token",
  "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:

* resource (REQUIRED): The resource's HTTPS URL
* 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
* resource_token_endpoint (OPTIONAL): URL where agents can proactively request resource tokens Section 10.3
* scope_descriptions (OPTIONAL): Object mapping scope values to Markdown strings for consent display
* 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 AAuth HTTP Message Signatures profile ([I-D.hardt-aauth-headers])

15. Request Verification

15.1. JWT Verification

When a request includes a JWT (agent token or auth token) via scheme=jwt, the server MUST verify the JWT per [RFC7515], [RFC7519], and the Signature-Key specification ([I-D.hardt-httpbis-signature-key]):

1. Decode the JWT header. Verify typ matches the expected token type (agent+jwt, auth+jwt, or resource+jwt).
2. Extract the iss and dwk claims from the JWT payload. Fetch {iss}/.well-known/{dwk}, parse as JSON, and extract the jwks_uri. Fetch the JWKS and locate the key matching the JWT header kid.

3. Verify the JWT signature using the discovered issuer key.
4. Verify exp is in the future. Verify iat is not in the future.
5. *Key binding*: Verify that the public key in the JWT's cnf claim matches the key used to sign the HTTP request.

15.1.1. Agent Token Verification

1. Perform JWT Verification. Verify dwk is aauth-agent.json.
2. Verify iss is a valid HTTPS URL conforming to the Server Identifier requirements.
3. Verify cnf.jwk matches the key used to sign the HTTP request.
4. If aud is present, verify that the server's own identifier is listed.

15.1.2. Auth Token Verification

1. Perform JWT Verification. Verify dwk is aauth-issuer.json.
2. Verify iss is a valid HTTPS URL.
3. Verify aud matches the resource's own identifier.
4. Verify agent matches the agent identifier from the request's signing context.
5. Verify cnf.jwk matches the key used to sign the HTTP request.
6. Verify that at least one of sub or scope is present.

15.1.3. Resource Token Verification

1. Perform JWT Verification. Verify dwk is aauth-resource.json.
2. Verify aud matches the auth server's own identifier.
3. Verify agent matches the requesting agent's identifier.
4. Verify agent_jkt matches the JWK Thumbprint of the key used to sign the HTTP request.
5. Verify exp is in the future.
6. Verify jti has not been seen before (replay detection).

15.1.4. JWKS Discovery and Caching

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.

15.1.5. Upstream Token Verification

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

1. Perform Auth Token Verification on the upstream token.
2. Verify iss is a trusted auth server (the auth server's own identifier, or a federated auth server).
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 auth server evaluates its own policy based on the upstream token's claims. The resulting downstream authorization is not required to be a subset of the upstream scopes.

16. Response Verification

16.1. Auth Token Response Verification

When an agent receives an auth token:

1. Verify the auth token JWT using the auth server's JWKS.
2. Verify iss matches the auth server the agent sent the token request to.
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.

16.2. 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. Send the resource token to the agent's own auth server's token endpoint.
5. Verify agent matches the agent's own identifier.
6. Verify agent_jkt matches the JWK Thumbprint of the agent's signing key.
7. Verify exp is in the future.

17. Error Responses

17.1. Authentication Errors

A 401 response from any AAuth endpoint uses the AAuth-Error header as defined in ([I-D.hardt-aauth-headers]).

17.2. 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.

17.3. 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
invalid_auth_token	400	Auth token for refresh is malformed, signature verification failed, or beyond refresh window
server_error	500	Internal error

Table 4

17.4. 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 5

18. Security Considerations

18.1. 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.

18.2. 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

18.3. 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 404

18.4. Clarification Chat Security

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

18.5. Auth Server Discovery

The resource's auth server is identified by the aud claim in the resource token. Federation mechanics are described in Appendix B.7.

18.6. 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.

18.7. Token Revocation

This specification does not define a token revocation mechanism. Auth tokens are short-lived and bound to specific signing keys.

18.8. Third-Party Initiated Login Security

Agents MUST treat all login endpoint parameters as untrusted input. The agent MUST verify the issuer and MUST validate that start_path is a relative path on its own origin.

18.9. TLS Requirements

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

19. IANA Considerations

19.1. 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 14.1
aauth-issuer.json	IETF	This document, Section 14.2
aauth-resource.json	IETF	This document, Section 14.3

Table 6

19.2. Media Type Registrations

This specification registers the following media types:

19.2.1. application/agent+jwt

- Type name: application
- Subtype name: 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 18
- Interoperability considerations: N/A
- Published specification: This document, Section 9
- Applications that use this media type: AAuth agents and auth servers
- Fragment identifier considerations: N/A

19.2.2. application/auth+jwt

- Type name: application
- Subtype name: 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 18
- Interoperability considerations: N/A
- Published specification: This document, Section 11
- Applications that use this media type: AAuth auth servers, agents, and resources
- Fragment identifier considerations: N/A

19.2.3. application/resource+jwt

- Type name: application
- Subtype name: 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 18
- Interoperability considerations: N/A
- Published specification: This document, Section 10
- Applications that use this media type: AAuth resources and auth servers
- Fragment identifier considerations: N/A

19.3. JWT Type Registrations

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

Type Value	Reference
agent+jwt	This document, Section 9
auth+jwt	This document, Section 11
resource+jwt	This document, Section 10

Table 7

19.4. 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
agent	Agent identifier	IETF	This document
agent_jkt	JWK Thumbprint of the agent’s signing key	IETF	This document

Table 8

19.5. AAuth Requirement Level Registry

This specification registers the following additional entry in the AAuth Requirement Level Registry established by the AAuth-Requirement specification ([I-D.hardt-aauth-headers]):

Value	Reference
auth-token	This document

Table 9

20. Design Rationale

20.1. 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.

20.2. Why No Authorization Code

AAuth eliminates authorization codes entirely. 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.

20.3. Why Every Agent Has a Legal Person

AAuth requires every agent to be associated with a legal person — a user or an organization. There are no truly autonomous agents. The auth server maintains this association. This ensures there is always an accountable party for an agent's actions, which is essential for authorization decisions, audit, and liability.

20.4. Why HTTPS-Based Agent Identity

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

20.5. Why No Refresh Token

Every AAuth request includes an HTTP Message Signature that proves the agent holds the private key. The expired auth token provides authorization context. A separate refresh token would be redundant.

20.6. Why JSON Instead of Form-Encoded

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

20.7. Why Callback URL Has No Security Role

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

20.8. Why Reuse OpenID Connect Vocabulary

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

20.9. 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 pseudonymous access, others verified identity, and others an auth token. AAuth's 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.

20.10. 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 pseudonym and identity levels, 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.

20.11. Why Not Extend GNAP

GNAP (RFC 9635) 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 authorization 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 authorization 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 authorization server directly. AAuth's "agents always go up" principle — where the agent only ever talks to its own auth server, and auth servers federate horizontally — 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, AS-to-AS federation, 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.

21. 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 at the time of writing:

* *Hell* (https://hello.coop): (https://hello.coop):) Auth server implementation of the AAuth protocol, including token endpoint, and deferred responses.

* *@aauth npm packages* (https://www.npmjs.com/org/aauth): (https://www.npmjs.com/org/aauth):) JavaScript/TypeScript libraries for AAuth agents and resources.

* *aauth-implementation* (https://github.com/christian-posta/aauth-implementation): (https://github.com/christian-posta/aauth-implementation):) Python library implementing key pair generation (Ed25519), HTTP Message Signatures, AAuth request signing, and Signature-Key header support. Author: Christian Posta.

* *keycloak-aauth-extension* (https://github.com/christian-posta/keycloak-aauth-extension): (https://github.com/christian-posta/keycloak-aauth-extension):) Java Keycloak SPI extension implementing the auth server role — HTTP Message Signature verification, AAuth token issuance with agent identity binding, consent flows, token refresh, token exchange for delegation chains, and /.well-known/aauth-issuer metadata. Author: Christian Posta.

* *aauth-full-demo* (https://github.com/christian-posta/aauth-full-demo): (https://github.com/christian-posta/aauth-full-demo):) End-to-end Python/JavaScript demo with multiple agents communicating via A2A protocol with AAuth authentication, including autonomous authorization, user-delegated consent via Keycloak, multi-hop token exchange, and JWKS discovery. Author: Christian Posta.

22. Document History

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

* draft-hardt-aauth-protocol-00
  - Initial submission

23. 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, Nate Barbettini, Wils Dawson.

24. References

24.1. Normative References

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

[I-D.hardt-httpbis-signature-key]
Hardt, D. and T. Meunier, "HTTP Signature-Key Header", Work in Progress, Internet-Draft, draft-hardt-httpbis-signature-key-02, 2 March 2026, <https://datatracker.ietf.org/doc/html/draft-hardt-httpbis-signature-key-02>.

[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>.

[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>.

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

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

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

[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>.

[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>.

24.2. Informative References

[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>.

[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>.

[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>.

[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>.

Appendix A. Agent Token Acquisition Patterns

This appendix describes common patterns for how agents obtain agent tokens from their agent server. In all patterns, the agent generates an ephemeral signing key pair, proves its identity to the agent server, and receives an agent token binding the ephemeral key to an agent identifier. Each pattern differs in how the agent proves its identity and what trust assumption the agent server relies on.

A.1. Server Workloads

1. The agent generates an ephemeral signing key pair (e.g., Ed25519).
2. The agent obtains a platform attestation from its runtime environment — such as a SPIFFE SVID from a SPIRE agent, a WIMSE workload identity token, or a cloud provider instance identity document (AWS IMDSv2, GCP metadata, Azure IMDS).
3. The agent presents the attestation and its ephemeral public key to the agent server.
4. The agent server verifies the attestation against the platform's trust root and issues an agent token with the ephemeral key in the cnf claim.

On managed infrastructure, the platform may additionally attest the software identity (container image hash, binary signature) alongside the workload identity, allowing the agent server to restrict tokens to known software.

Trust assumption: The agent server trusts the platform's attestation that the workload is what it claims to be.

A.2. Mobile Applications

1. The app generates an ephemeral signing key pair, optionally backed by the device's secure enclave (iOS Secure Enclave, Android StrongBox).
2. The app obtains a platform attestation — iOS App Attest assertion or Android Play Integrity verdict — binding the app identity and the ephemeral public key.
3. The app sends the attestation and public key to the agent server.
4. The agent server verifies the attestation against Apple's or Google's attestation service and issues an agent token.

The platform attestation proves the app is a genuine installation from the app store, running on a real device, and has not been tampered with. If the key is hardware-backed, the attestation also proves the key cannot be exported.

Trust assumption: The agent server trusts the platform's attestation that the app is a genuine, untampered installation running on a real device.

A.3. Desktop and CLI Applications

Desktop platforms generally do not provide application-level attestation comparable to mobile platforms. Several patterns are available depending on the deployment context:

A.3.1. User Login

1. The agent opens a browser and redirects the user to the agent server's web interface.
2. The user authenticates at the agent server.
3. The agent generates an ephemeral signing key pair, stores the private key in a platform vault (macOS Keychain, Windows TPM, Linux Secret Service), and sends the public key to the agent server.
4. The agent server issues an agent token binding the ephemeral key to an agent identifier and returns it to the agent (e.g., via localhost callback).

This is the most common pattern for user-facing desktop and CLI tools.

The agent may also hold a stable key in hardware (TPM, secure enclave) or a platform keychain. During the initial user login flow, the agent server records the stable public key alongside the agent identity. When the agent token expires, the agent can renew it by sending its new ephemeral public key in a scheme=jkt-jwt request signed by the stable key, without requiring the user to log in again.

Trust assumption: The agent server trusts the user's authentication but cannot verify which software is running — only that the user authorized the agent. For renewal via stable key, the agent server trusts that the key registered at enrollment continues to represent the same agent.

A.3.2. Managed Desktops

On managed desktops (e.g., corporate MDM-enrolled devices), the management platform may provide device and software attestation similar to server workloads. The agent presents the platform attestation alongside its ephemeral key, and the agent server verifies the device is managed and the software is approved.

Trust assumption: The agent server trusts the management platform's attestation that the device is managed and the software is approved.

A.3.3. Self-Hosted Agent Metadata

A user publishes agent metadata and a JWKS at a domain they control (e.g., username.github.io/.well-known/aauth-agent.json) — no active server is required, only static files. The agent's public key is included in the published JWKS. The corresponding private key is held on the user's machine — potentially in a secure enclave or hardware token. Agents generate ephemeral signing keys and use scheme=jwt to obtain agent tokens signed by the private key. Auth servers verify agent tokens against the published JWKS.

Trust assumption: The trust anchor is the published JWKS and the private key held by the user. No server-side logic is involved — verification relies entirely on the static metadata and key material.

A.4. Browser-Based Applications

1. The web server — which acts as the agent server — authenticates the user. The recommended mechanism is WebAuthn, which binds authentication to the device and origin, preventing scripts or headless browsers from impersonating the web page to obtain an agent token.
2. The web app generates an ephemeral signing key pair using the Web Crypto API (non-extractable if supported) and sends it to the web server.
3. The web server issues an agent token binding the web app's ephemeral public key to an agent identifier and returns it.

The key pair and agent token exist only for the lifetime of the browser session. The web server controls both the agent identity and the issuance.

Trust assumption: The web server is the agent server and controls the entire lifecycle. The agent token lifetime is tied to the browser session. When WebAuthn is used, authentication is bound to the device and origin rather than relying solely on session credentials.

Appendix B. Detailed Flows

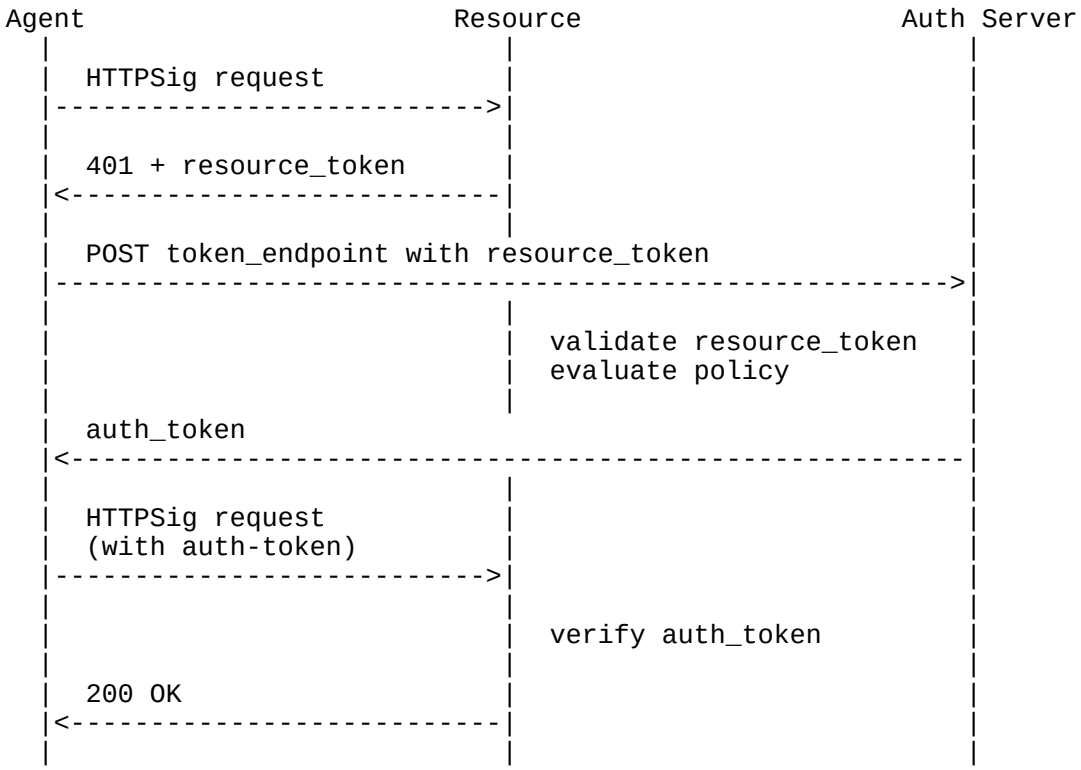
This appendix provides complete end-to-end flows combining the key interactions defined in the Protocol Overview.

B.1. Autonomous Agent

A machine-to-machine agent obtains authorization directly without user interaction.

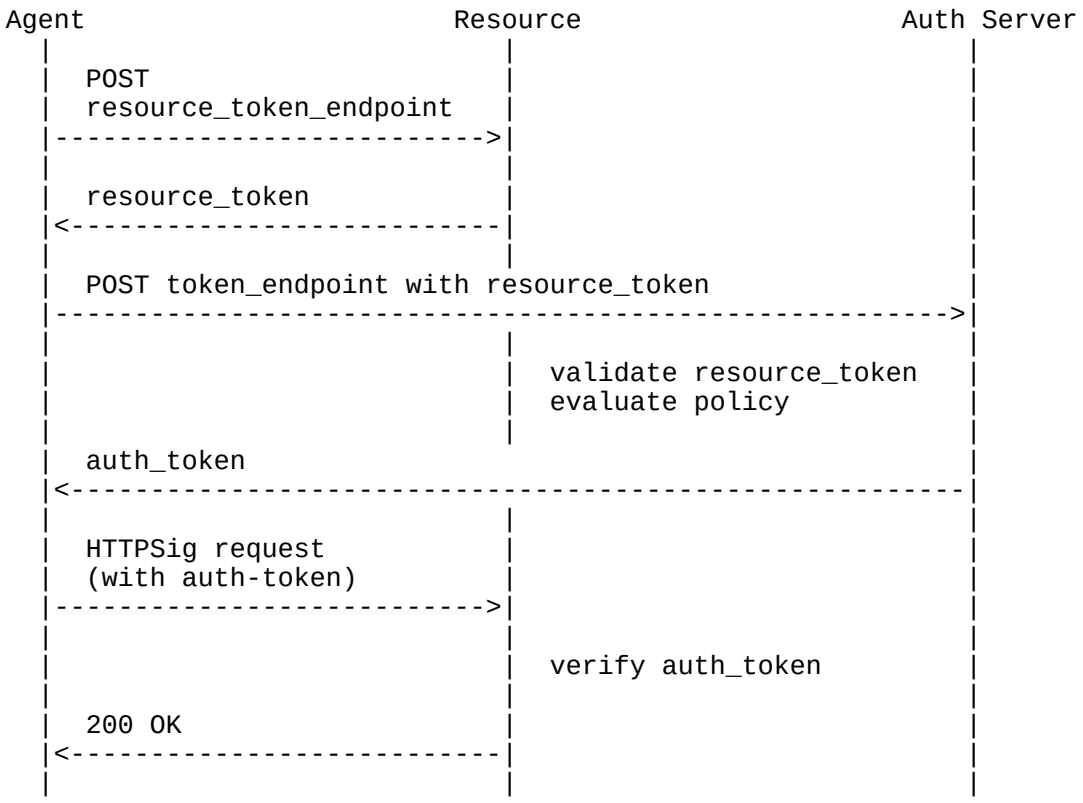
B.1.1. Resource Challenge

The resource challenges the agent with a 401 response containing a resource token:



B.1.2. Proactive Token Request

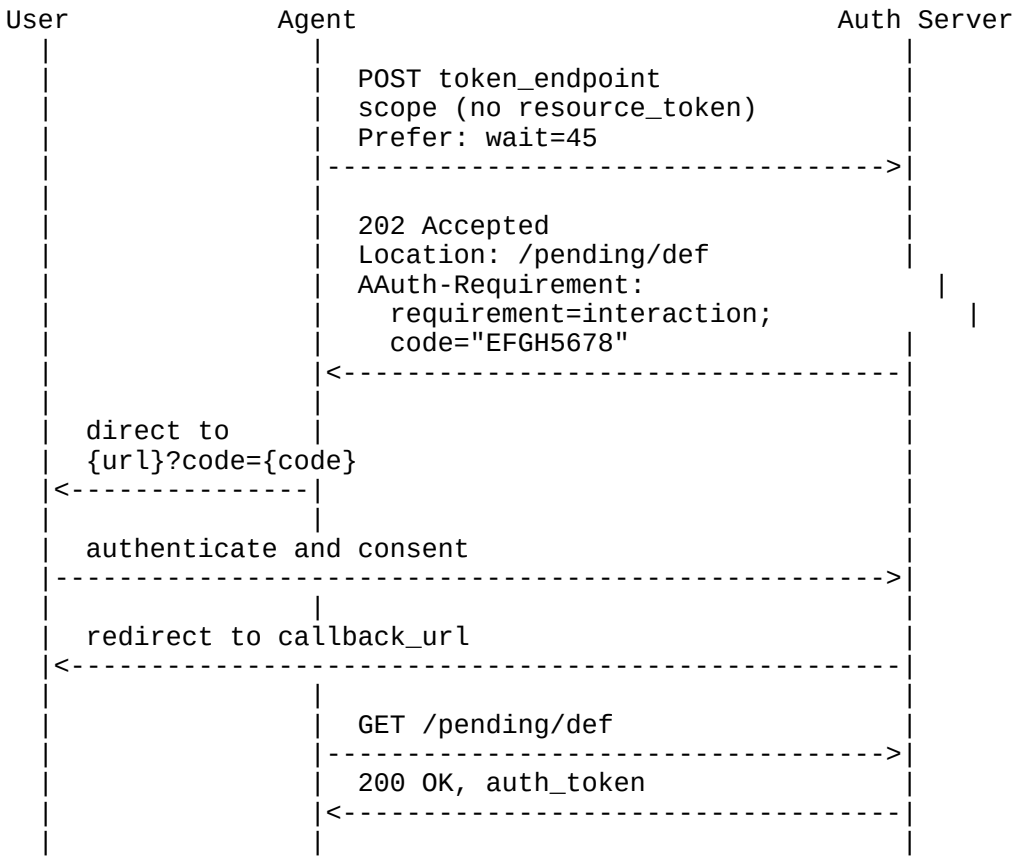
When the agent knows the resource's requirements from metadata, it can request a resource token proactively via the resource_token_endpoint:



Use cases: Machine-to-machine API calls, automated pipelines, cron jobs, service-to-service communication where no user is involved.

B.2. Agent as Audience

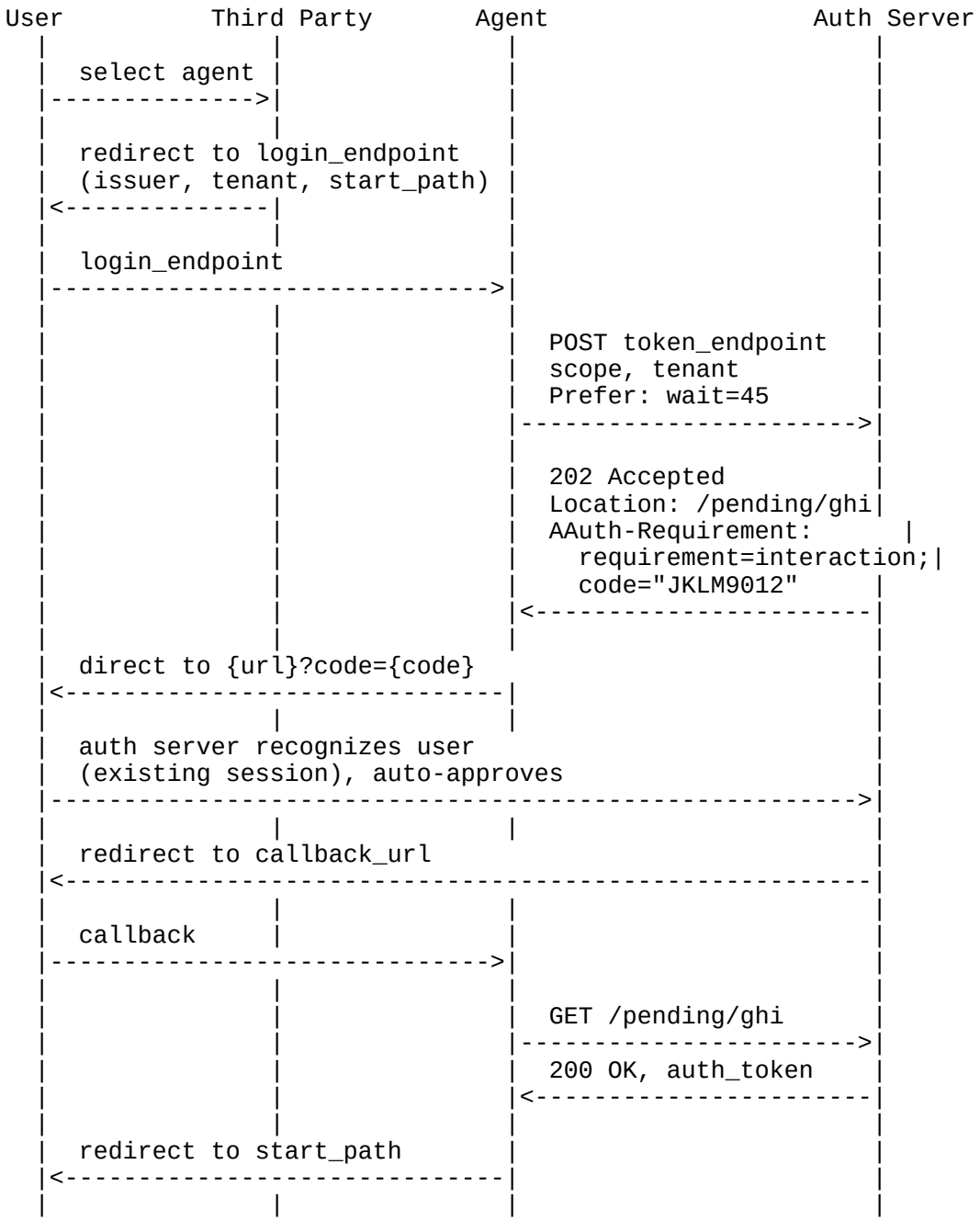
An agent requests an auth token where it is the audience — either for SSO (obtaining user identity) or for first-party resource access. The agent calls the token endpoint with scope (and no resource_token), since the agent itself is the resource.



Use cases: Single sign-on, user login, enabling agent to access protected resources at the agent on behalf of the user.

B.3. Third-Party Initiated Login

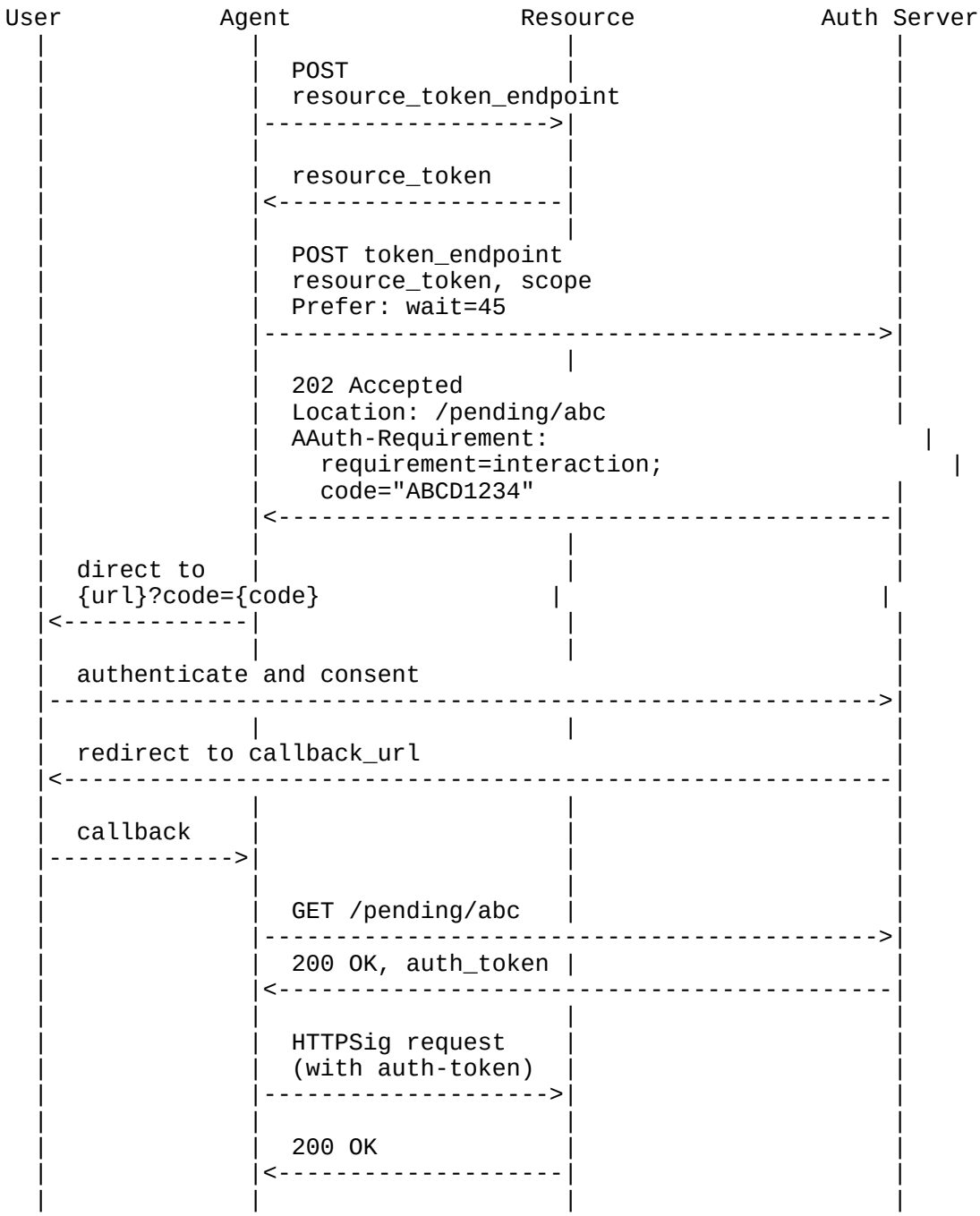
A third party — such as an auth server, organization portal, app marketplace, or partner site — directs the user to the agent's login_endpoint with enough context to start a login flow. The agent then initiates a standard "agent as audience" flow.


User
Third Party
Agent
Auth Server
select agent
redirect to login_endpoint
(issuer, tenant, start_path)
login_endpoint
POST token_endpoint
scope, tenant
Prefer: wait=45
202 Accepted
Location: /pending/ghi
AAuth-Requirement:
requirement=interaction;
code="JKLM9012"
direct to {url}?code={code}
auth server recognizes user
(existing session), auto-approves
redirect to callback_url
callback
GET /pending/ghi
200 OK, auth_token
redirect to start_path

Use cases: Organization portal SSO, app marketplace "connect" buttons, partner site deep links, auth server dashboard launching an agent.

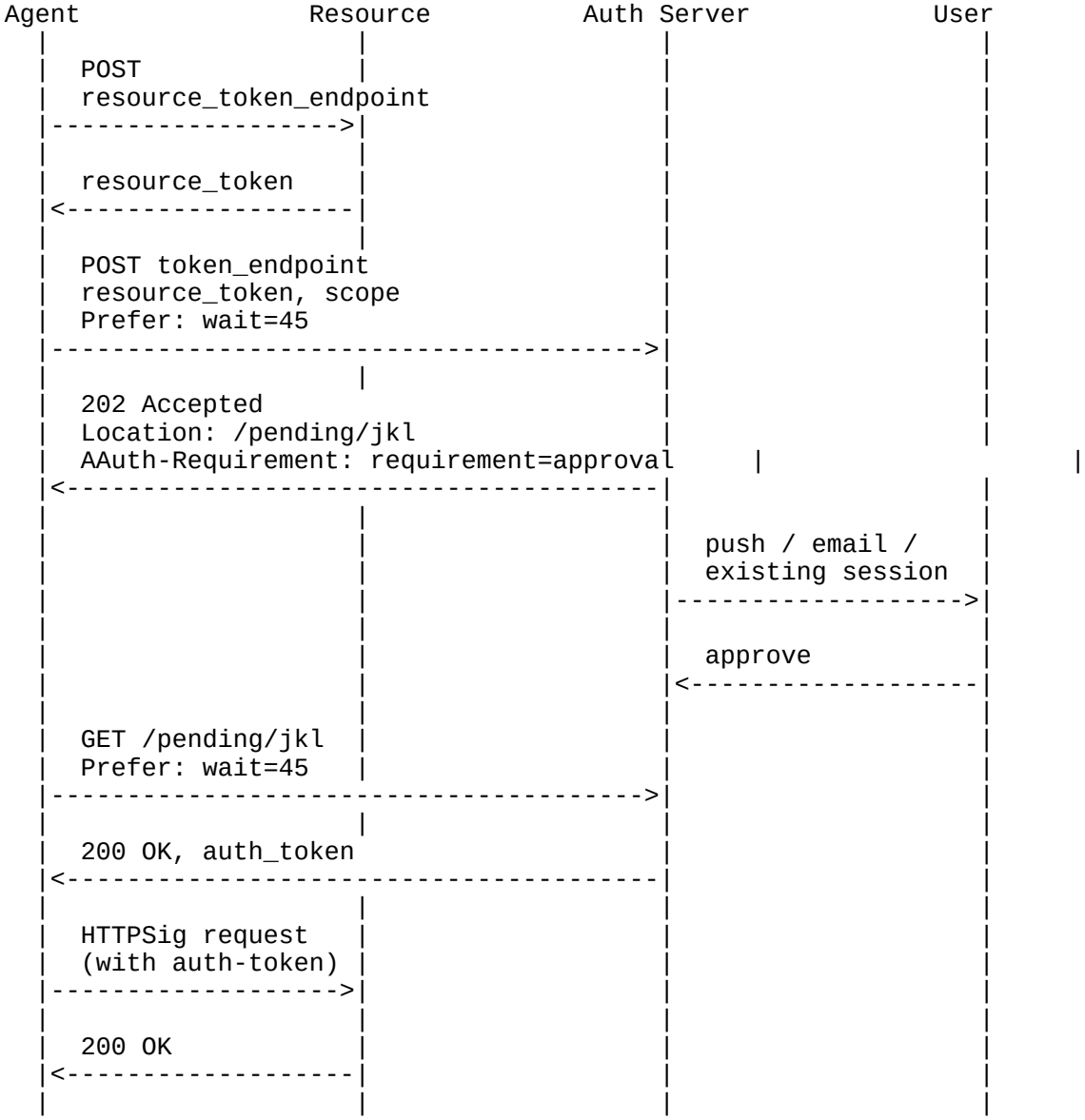
B.4. User Authorization

Full flow with user-authorized access. The agent obtains a resource token from the resource’s resource_token_endpoint, then requests authorization from the auth server.


User
Agent
Resource
Auth Server
POST
resource_token_endpoint
resource_token
POST token_endpoint
resource_token, scope
Prefer: wait=45
202 Accepted
Location: /pending/abc
AAuth-Requirement:
requirement=interaction;
code="ABCD1234"
direct to
{url}?code={code}
authenticate and consent
redirect to callback_url
callback
GET /pending/abc
200 OK, auth_token
HTTPSig request
(with auth-token)
200 OK

B.5. Direct Approval

The auth server obtains approval directly — from a user (e.g., push notification, existing session, email) — without the agent facilitating a redirect.



B.6. Call Chaining

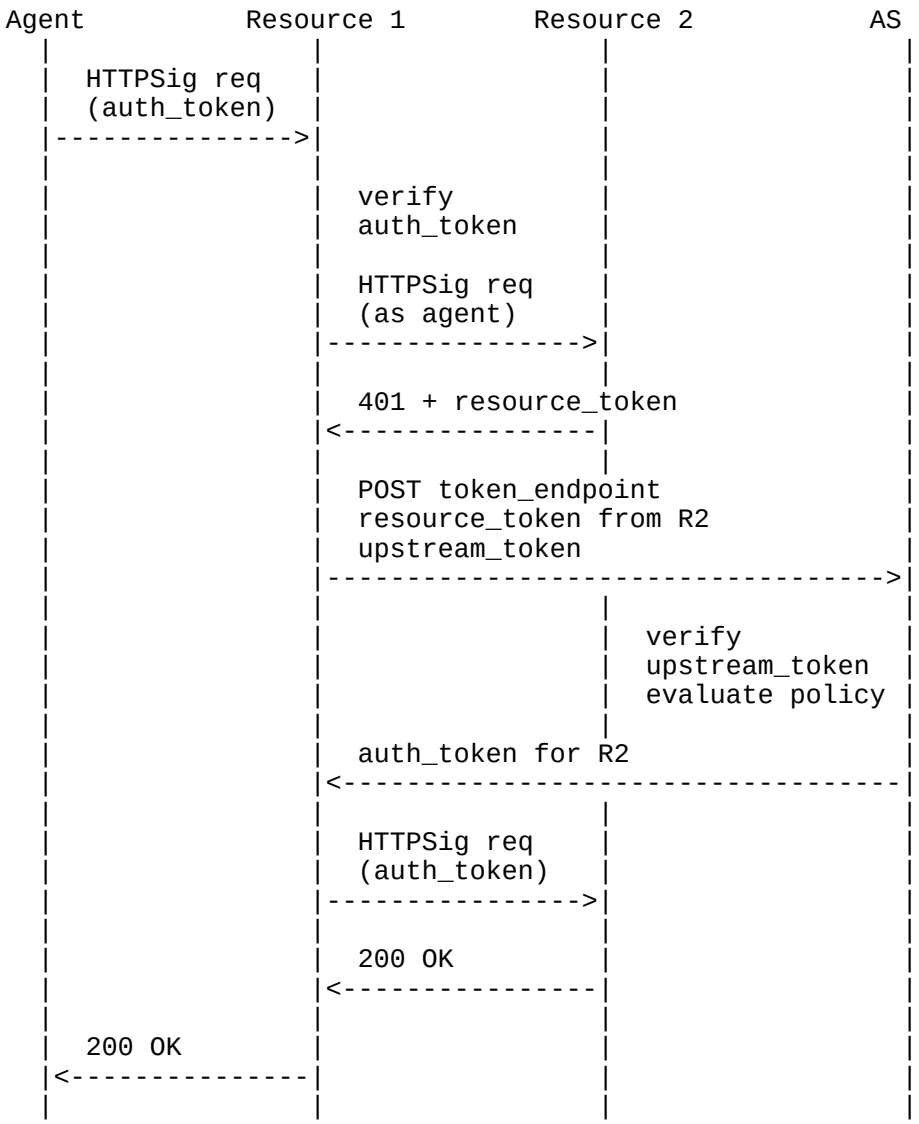
When a resource needs to access a downstream resource on behalf of the caller, it acts as an agent. Like any agent, it sends the downstream resource token to its own auth server along with the auth token it received from the original caller as the upstream_token.

The auth server evaluates its own policy based on both the upstream auth token and the downstream resource token. The resulting authorization is not necessarily a subset of the upstream scopes.

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

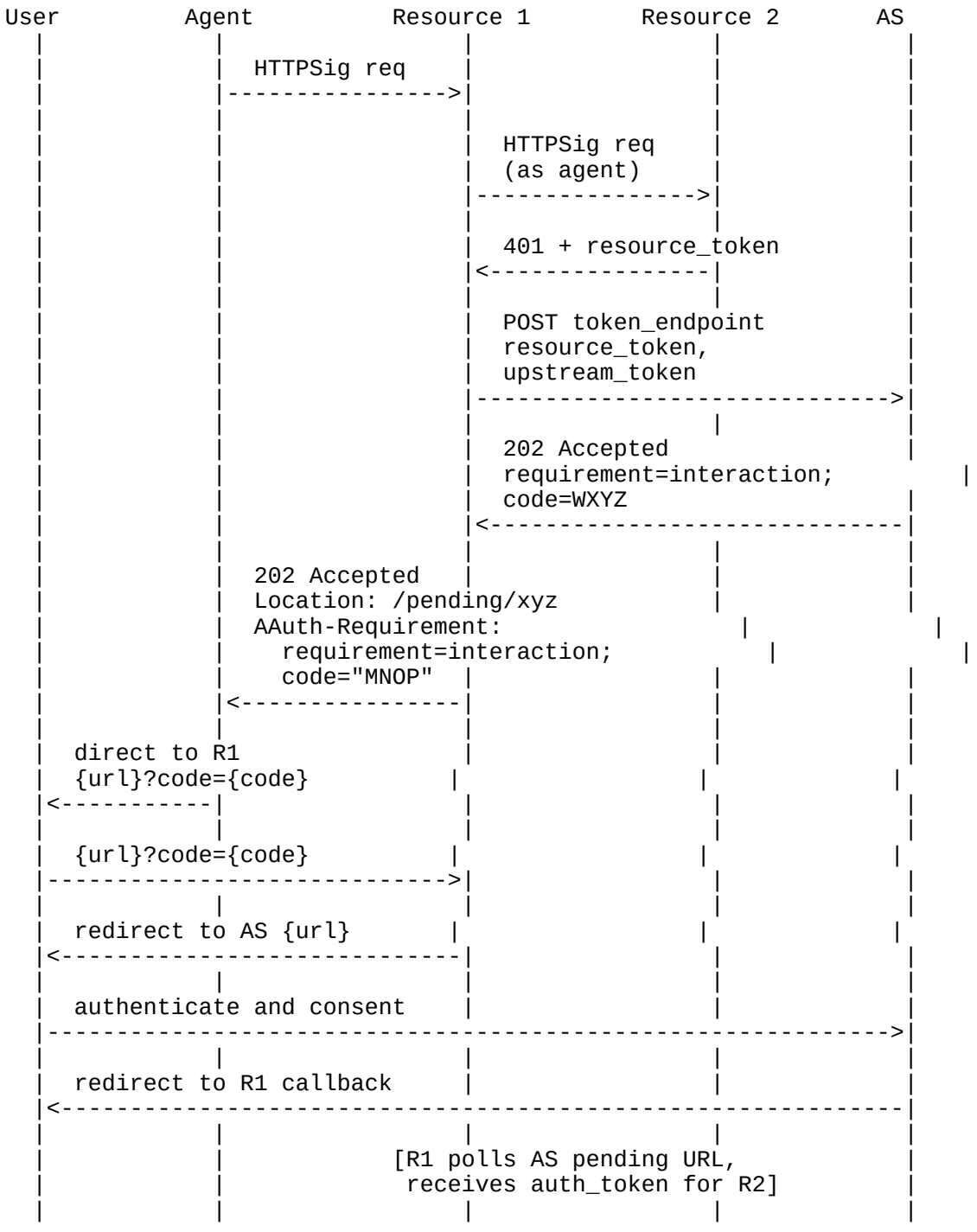
B.6.1. Same Auth Server

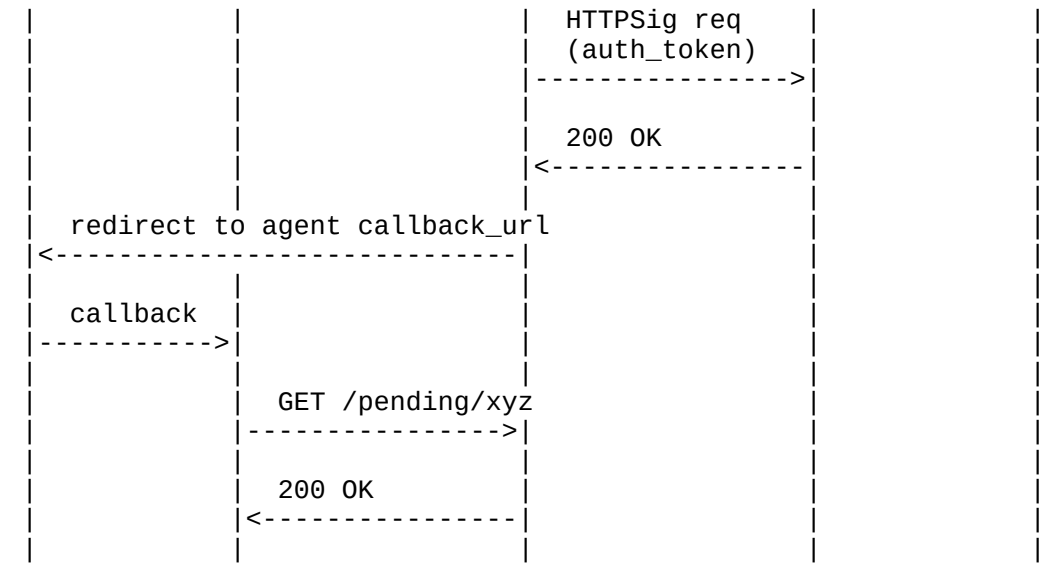
When both resources share the same auth server:



B.6.2. Interaction Chaining

When the auth server 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.


User
Agent
Resource 1
Resource 2
AS
HTTPSig req
HTTPSig req
(as agent)
401 + resource_token
POST token_endpoint
resource_token,
upstream_token
202 Accepted
requirement=interaction;
code=WXYZ
202 Accepted
Location: /pending/xyz
AAuth-Requirement:
requirement=interaction;
code="MNOP"
direct to R1
{url}?code={code}
{url}?code={code}
redirect to AS {url}
authenticate and consent
redirect to R1 callback
[R1 polls AS pending URL,
receives auth_token for R2]



When a resource acting as an agent receives a 202 Accepted response with AAuth-Requirement: requirement=interaction from its auth server, 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.

When call chaining crosses auth server domains, the agent's auth server (or the intermediary resource's auth server) federates with the downstream resource's auth server. See Appendix B.7.

B.7. Cross-Domain Trust

When an agent's auth server (AS1) receives a resource token whose aud identifies a different auth server (AS2), AS1 federates with AS2 to obtain the auth token. This enables agents and resources that operate within different trust domains to work together — the federation effort is at the auth server level, not at every agent and resource.

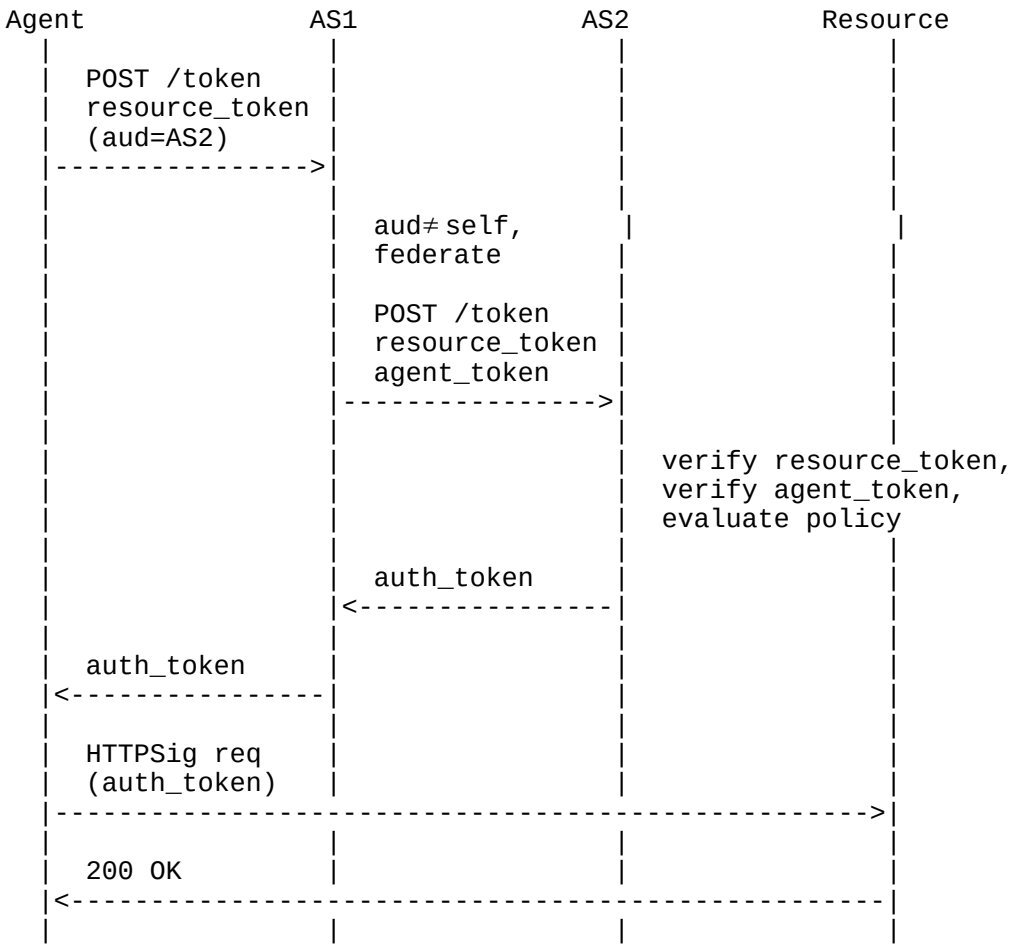
Cross-domain federation is required when the agent's AS and the resource's AS differ. Within a single AS domain, no federation is needed.

B.7.1. AS-to-AS Federation

AS1 calls AS2's token endpoint with the resource token and the agent's agent token. AS2 uses the agent token to verify agent identity, confirm the agent's key matches the agent_jkt in the resource token, and include agent claims in the auth token it issues.

AS2 returns the auth token to AS1. AS1 passes the auth token through to the agent unchanged — AS1 MUST NOT re-sign or modify the auth token. The auth token's iss is AS2 and aud is the resource. The agent forwards it to the resource, which verifies the auth token against AS2's JWKS (its own auth server). The agent's response verification Section 16.1 is limited to checking that aud, cnf, and agent match its own values — the agent does not need to verify the auth token's signature.

AS2 MAY also return an assessment, a Markdown string describing AS2's evaluation of the request for AS1 to consider in its authorization decision. *TODO:* Define recommended sections. Any step may return a 202 deferred response — the standard AAuth deferred response protocol applies to AS-to-AS calls.



B.7.2. Organization Visibility

Organizations benefit from the trust model Section 4: internal agents and resources share a single AS with zero federation overhead. Federation is only incurred at the boundary, when an internal agent accesses an external resource or vice versa.

B.7.3. Token Endpoint Parameters for Federation

When an auth server calls another auth server’s token endpoint for federation, the following additional parameter is used:

* agent_token (OPTIONAL): The agent's agent token. Sent by AS1 to AS2 so that AS2 can verify the agent's identity and signing key, and include agent claims in the issued auth token. AS2 verifies that the cnf claim in the agent token matches the agent_jkt in the resource token.

The upstream_token parameter (used in call chaining) may also be present in federated calls to provide authorization chain provenance.

B.7.4. Relationship to AAuth Mission Protocol

AAuth Mission extends cross-domain federation with mission-scoped authorization, centralized audit, and MA countersignatures. When a mission is active, the agent's auth server acts as the Mission Authority and adds mission context to federation calls. See the AAuth Mission specification for details.

Author's Address

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