Authorization Server as PDP for Tool-Scoped MCP Access: Audience-Bound Tokens and Deterministic PEP Enforcement

David Tubía

2026-02-20

Abstract

Agentic systems built on the Model Context Protocol (MCP) tend to inherit a familiar OAuth failure mode: authorization is verified at the MCP server boundary, but not at the individual tool boundary. In practice, once an access token is accepted for an MCP audience, a wide set of tools becomes reachable. In an agentic chain, this breaks least privilege and makes the agent a confused deputy: a probabilistic planner now holds operational permissions that the calling service never intended to grant.

This paper reframes the practical problem as follows: how do we achieve efficient, tool-scoped access control without embedding a policy decision point (PDP) into every gateway and MCP server? The deployable answer is to make the Authorization Server / IdP act as the PDP at token issuance time. Access tokens become signed "decision artifacts" that are simultaneously:

- Resource-bound: the protected resource is encoded in `aud` (canonical URL per RFC 8707).
- Tool-bound: the allowed MCP tools are encoded in `scope` as exact tool identifiers.

Gateways and MCP runtimes become lightweight policy enforcement points (PEPs): they validate JWTs (RFC 9068), enforce audience membership by canonicalizing the request host to a resource identifier, and perform an exact match between the MCP JSON-RPC tool name (`params.name`) and a scope token. We cover delegation using OAuth 2.0 Token Exchange (RFC 8693), multi-resource transactions (multi-audience `aud[]`), and multiplexing/federation scenarios where a single gateway fronts multiple MCP servers. We provide deterministic matching rules, complete allow/deny walkthroughs, and a conformance suite of test vectors.

1. Scope, assumptions, and threat model

This paper deliberately focuses on the backend client application (a confidential client) and the agent runtime. We do not assume an end-user is present or that consent UI exists. Think of a scheduled batch job, an internal service, or a platform workflow that invokes an agent to orchestrate tools.

1.1 In-scope

- Client application authenticates to an Authorization Server (AS) and obtains tokens.
- Client calls an agent for a specific intent (business operation).
- Agent calls MCP servers via tool invocations (`tools/call`), optionally using `tools/list`.
- Authorization is enforced per tool, not merely per MCP server.

1.2 Out-of-scope (but noted)

- Human-in-the-loop approvals.
- Prompt safety and content filtering (important, but orthogonal).
- Full proofs-of-possession (DPoP/mTLS) beyond brief recommendations.
- Full MCP server registry governance (we cover only what is needed for authZ).

1.3 Threats we care about

- Over-privilege / blast radius: token valid for an MCP server implies access to many tools.
- Confused deputy: agent has broader privileges than the client intended, and performs actions "for" the client.
- Tool abuse via planning error: agent chooses the wrong tool (e.g., executes `payments.transfer` instead of `list.accounts`).
- Prompt-induced tool abuse: input manipulates the agent into calling a high-risk tool it should not call.
- Tool enumeration: agent (or attacker controlling it) lists all tools and pivots.

1.4 Threat Model and Mitigations

This paper assumes modern OAuth/OIDC foundations (confidential clients, TLS everywhere, short-lived access tokens). The failure mode we are targeting is not "no auth", but "auth that is valid at the server boundary yet too broad at tool execution time". In agentic chains, that gap is where prompt-induced abuse and confused-deputy behavior live.

The mitigations below are intentionally biased toward deterministic, local enforcement at PEPs (gateways, agent runtimes, MCP servers) using tokens minted by a single PDP (the Authorization Server / IdP). The goal is a simple and verifiable contract: a signed token is a decision artifact, and the PEP does exact matching between request context (resource + tool) and token claims.

2. Background

2.1 MCP tools at a glance

MCP defines a tool model where servers expose tools that clients can discover and invoke. The protocol commonly uses JSON-RPC 2.0 messages. Two relevant methods are:

- `tools/list`: discover available tools
- `tools/call`: invoke a tool, typically using `params.name` and `params.arguments`

2.2 OAuth building blocks

We build on four OAuth/IETF primitives:

- Bearer token usage (RFC 6750): access tokens are presented to resources over TLS.
- JWT access token profile (RFC 9068): a widely used profile for JWT-encoded access tokens.
- Resource Indicators (RFC 8707): clients request tokens for a specific protected resource using a `resource` parameter; AS can constrain the token audience to that resource.
- Token Exchange (RFC 8693): a client exchanges one token for another (often used as OBO) to obtain downscoped tokens for downstream calls.

Optionally:

Threat (MCP authn/authz)	Primary PEP/PDP control	Secondary (optional hardening)
Prompt-induced tool abuse	Pre-exec policy check (PEP calling PDP): treat planner output as untrusted; validate tool args against schema and enforce per-tool/action allow policies	HITL for high-privilege or irreversible actions
Confused deputy (token passthrough / transitive privilege)	Validate OAuth2.1/OIDC on every call (iss/aud/exp/sig) + prohibit token passthrough; use OBO/token delegation	Centralize policy enforcement (gateway PDP) for consistent authz/consent/tool filtering
Token replay	Short-lived, narrowly scoped tokens; revalidate per call before tool/resource execution	`jti`/nonce tracking + sender-constrained tokens (e.g., DPoP/mTLS where feasible)
Audience misuse	Strict `aud` enforcement at the PEP before any tool execution	Signed intent binding (subject, audience, purpose, session); reject mismatches
Naming confusion / tool alias collisions	Enforce fully-qualified tool identifiers and fail-closed on ambiguity	Require explicit disambiguation and/or re-consent for ambiguous resolution
Scope escalation (delegation chains / permission creep)	Task-scoped, time-bound permissions with explicit permission boundaries; per-action authorization via centralized PDP	Bind permissions to subject/resource/purpose/duration; prevent privilege inheritance unless intent is re-validated

Table 1: Threats and mitigations for tool-scoped MCP authorization.

- Rich Authorization Requests (RFC 9396): structured permissions via `authorization_details`.
- Protected Resource Metadata (RFC 9728): resource discovery, relevant because MCP authorization guidance encourages metadata discovery and dynamic client registration.

2.2.1 Terminology: `resource` vs JWT `aud` vs Token Exchange `audience`

These three words get mixed in real implementations, and that is where subtle bugs hide:

- `resource` is an OAuth request parameter (RFC 8707). Clients use it to ask the Authorization Server for an access token intended for a specific protected resource (in our case: a specific MCP server or MCP gateway).
- `aud` is a JWT claim (and required by the JWT access token profile, RFC 9068). The Authorization Server encodes the target protected resource identifier(s) into `aud`. The MCP gateway / server enforces it by membership: its own canonical identifier must be present (string or array).
- `audience` is an optional Token Exchange (RFC 8693) request parameter. It is not the same thing as RFC 8707 `resource`, and different Authorization Servers interpret it differently (sometimes as a client identifier, sometimes as a resource hint, sometimes ignored).

For MCP, this paper uses a consistent convention:

- Use RFC 8707 `resource` when you mean "mint a token for this MCP server".
- Treat JWT `aud` as the enforcement surface at the PEP.
- Only mention Token Exchange `audience` when you are explicitly talking about the RFC 8693 parameter, and explain why your AS needs it (most designs do not).

2.3 MCP 2025-11-25: details that matter for tool-scoped auth

This paper is aligned to the MCP specification revision 2025-11-25. For backend-to-agent-to-MCP designs, four details matter most:

1) Tool invocations are explicit protocol messages. A `tools/call` request carries the tool identifier in `params.name`. That gives us a stable, parseable authorization primitive.

2) Tool identifiers have naming guidance. The MCP spec recommends tool names be treated as case-sensitive, 1-128 characters, and restricted to a tight ASCII set (letters/digits/underscore/hyphen/dot). That makes "scope token == tool name" feasible without Unicode confusable horror. Many deployments still choose to enforce a canonical form (e.g., lowercase) at the gateway as a hardening and governance measure; if you do, treat non-canonical variants as DENY and document it as part of your tool contract.

3) The authorization model is OAuth-native. MCP authorization guidance relies on protected resource metadata (RFC 9728) for discovery and on audience-bound access tokens via the OAuth `resource` indicator (RFC 8707). In practice, MCP servers (or gateways) must reject tokens not minted specifically for their canonical resource URI (typically enforced via `aud`).

4) The spec explicitly supports challenge-driven, incremental authorization. When a client lacks scope, the resource can respond with `WWW-Authenticate` describing the required scope. For machine clients, that becomes deterministic: re-run OAuth (or token exchange) to obtain a token scoped to exactly the tool set needed for the current intent.

Also: access tokens MUST NOT be sent via URL query parameters. Use the `Authorization: Bearer` header.

Concrete envelope (HTTP + JSON-RPC):

```
POST /mcp HTTP/1.1
Host: mcp-gw.example.com
Authorization: Bearer <AT_mcp>

{
  "jsonrpc":"2.0",
  "id":7,
  "method":"tools/call",
  "params":{
    "name":"payments.transfer",
    "arguments":{...}
  }
}
```

Tool-scoped enforcement ultimately hinges on a single field: `params.name`.

3. The practical problem: tool-level access control without embedding a PDP

Consider a typical backend flow:

1. The client obtains an access token for an MCP server (requested with `resource=<MCP base URL>`; minted with `aud` bound to that canonical resource).
2. The agent uses that token to connect to the MCP server.
3. The MCP server checks token validity and `aud` membership.
4. The agent can now call any tool the server exposes.

This breaks least privilege. In production:

- Tool selection is probabilistic and prompt-sensitive.
- MCP servers may expose read, write, admin, destructive tools.
- The agent runtime is a complex software system, not a trusted principal.

We want the token to say: "This caller can do exactly these tools, for this resource, for this intent."

4. Design goals and invariants

4.1 Invariant A: resource binding is canonical (`resource` -> `aud`)

A token used at an MCP gateway (or MCP server) MUST be bound to that protected resource:

- The client requests the token using the RFC 8707 `resource` indicator (canonical MCP base URL).
- The Authorization Server encodes the selected resource identifier(s) into the JWT `aud` claim.
- The gateway/server enforces it by exact match (membership when `aud` is an array).

Avoid mixing this with Token Exchange `audience` unless you are explicitly relying on the RFC 8693 parameter and you have documented how your Authorization Server maps it to `aud` (behavior is product-specific).

4.2 Invariant B: tool permissions are explicit and enforceable

A token used for MCP tool calls MUST encode which tools are allowed. Representations:

- Option 1 (simple): `scope` contains tool names exactly (space-delimited) plus any generic scopes.
- Option 2 (safer): a structured claim such as `tool_permissions`.
- Option 3 (most expressive): RAR `authorization_details` representing tool permissions with constraints.

Enforcement MUST be deterministic: exact match, no wildcards by default.

4.3 Invariant C: downscoping is monotonic

When the agent obtains an OBO token for the MCP gateway, the resulting token MUST be a subset of the caller's effective privileges for the specific intent.

4.4 Invariant D: enforcement is externalized (preferably)

Tool enforcement should not be re-implemented in every MCP server and agent. Prefer gateways (PEP) that validate JWTs and enforce tool rules.

5. Architecture patterns

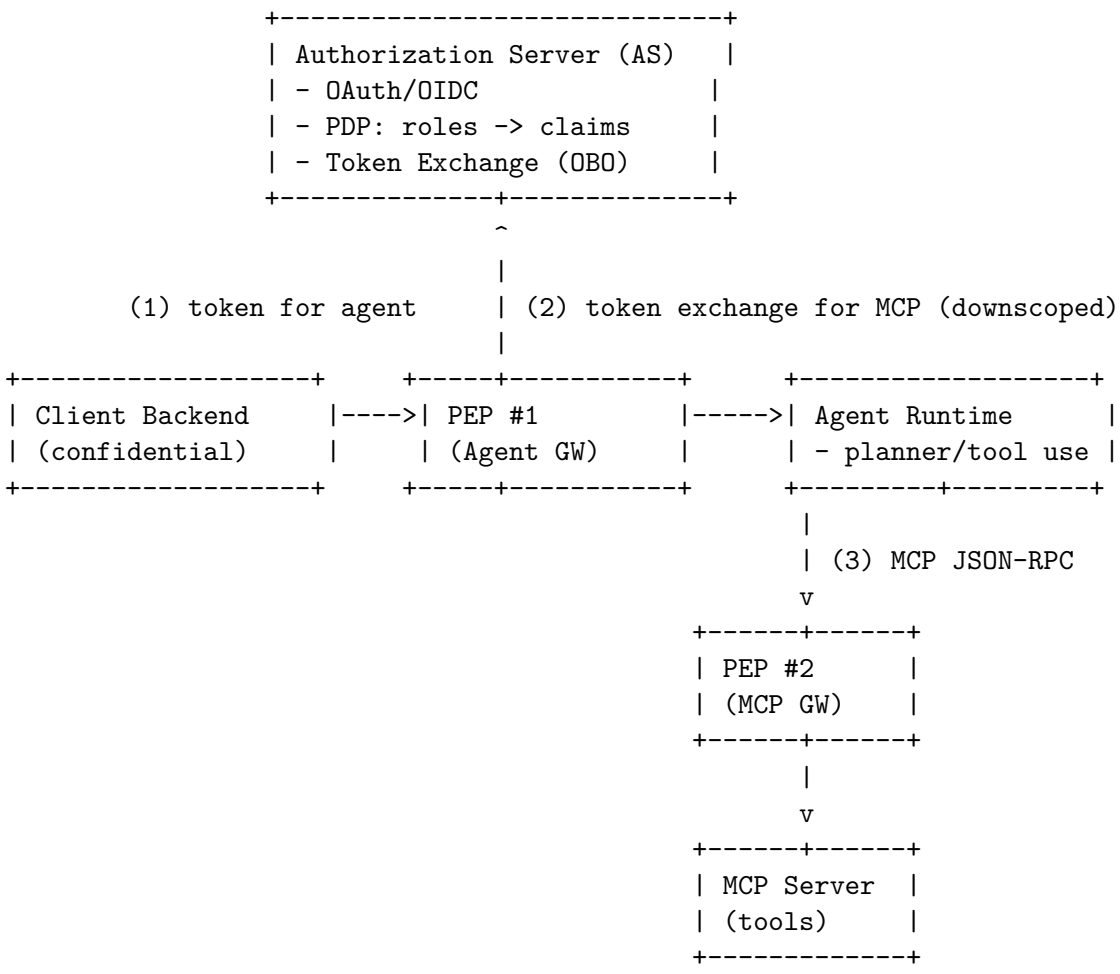
5.1 Gateway-enforced authorization (Agent Gateway + MCP Gateway)

We model two enforcement boundaries. Both are **PEPs** (Policy Enforcement Points): they perform deterministic verification on the request and the token. The **AS/IdP remains the PDP** (Policy Decision Point) by deciding which claims appear in the token at issuance time.

- **PEP #1 (Agent Gateway, recommended):** sits in front of the agent runtime. It enforces that callers hold a token minted for the **agent resource** by validating the JWT and requiring an **audience match** against the gateway's canonical RFC 8707 resource URL. This keeps authentication/authorization out of agent code (developers do not become accidental security engineers).
- **PEP #2 (MCP Gateway, recommended):** sits in front of MCP servers. It validates the JWT and performs **tool-level authorization** by matching the requested MCP tool name against claims (scope / tool_permissions) that the AS emitted.

In practice, PEP #1 and PEP #2 can be deployed as **two separate gateways** or as a **single shared gateway** with two logical policies. The key invariant is the same: *the token is minted for the resource being called, and the PEP checks that deterministically.*

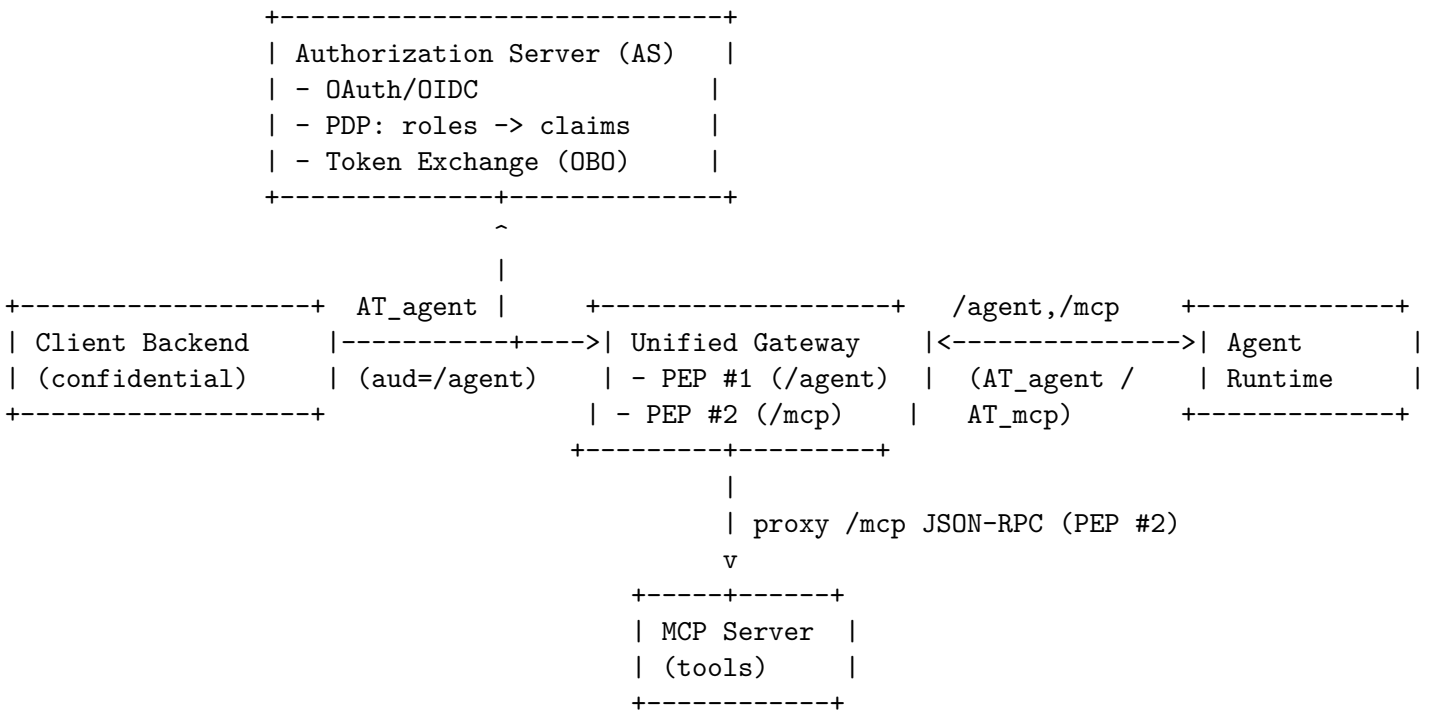
Option A: two gateways (distinct audiences)



Option B: one shared gateway (two logical PEP roles)

When you prefer a single governed data plane, you can run one gateway and publish two **resource identifiers** (RFC 8707) on the same host (path-scoped resources are fine):

- Agent API resource: `https://gw.example.com/agent`
- MCP API resource: `https://gw.example.com/mcp` (optionally per upstream: `https://gw.example.com/mc` or `https://gw.example.com/mcp/<tenant>`)



This deployment strengthens governance: the security team owns a single gateway policy plane that fronts **both** the agent API and the MCP API, while the AS/IdP remains the single source of truth for *who can get which claims*.

The key property of this option is that there is **no direct agent-to-MCP connectivity**. Both the agent runtime and MCP server(s) live in isolated networks and are reachable only from the gateway (for example using mTLS, ACLs, private routing, or service mesh identity). The flow is:

- **Client -> GW (/agent):** the client requests `AT_agent` with `aud=https://gw.example.com/agent` and an `intent_id`, then calls the agent endpoint on the gateway. PEP #1 validates the token (issuer, signature, exp/nbf, `aud`) and proxies to the agent runtime.
- **Agent -> GW (/mcp):** when the agent decides it needs a tool, it obtains/uses `AT_mcp` with `aud=https://gw.example.com/mcp` and `scope` containing the allowed tool IDs, then calls `/mcp` on the same gateway. PEP #2 enforces tool-level scope (and any tenant/namespace rules) and proxies the JSON-RPC call to the MCP server.

5.1.1 Gateway resource identifiers vs upstream routing (who validates `aud`?) A subtle but important question in shared-gateway deployments is: what is the *protected resource* for the purposes of RFC 8707 audience restriction? Is it the **gateway URL** the caller (client or agent) calls, or the **upstream agent / MCP server** that ultimately executes the request?

Pattern	What the token targets	Notes (security + ops)
P1. Gateway is the OAuth resource server (terminate at GW)	`aud = gw URL` (path-scoped per agent/MCP endpoint)	Upstreams are not publicly reachable; GW enforces `aud`/`scope`/tool claims and routes internally. Protect GW-to-upstream with mTLS/ACLs/service mesh. Simplest and usually preferred.
P2. Gateway performs token exchange (downstream token mediation)	Inbound: `aud = gw`; downstream: `aud = upstream`	GW validates inbound token, then uses RFC 8693 token exchange to mint a new, short-lived token for the upstream resource (optionally downscoped). Enables defense-in-depth. Adds latency/AS load. Requires strict allowlists and downscoping to prevent exchange abuse.
P3. Multi-audience token (`aud[]` includes GW and upstream)	`aud = [gw, upstream]`	Possible with JWT and some AS policies, but increases blast radius. RFC 8707 warns multi-audience bearer tokens require high trust between recipients; RFC 9700 recommends audience restriction to a specific RS or a small set. Only consider when upstream is not directly reachable and the trust boundary is strong.

Table 2: Gateway-to-upstream audience patterns for RFC 8707-style resource identifiers.

This matters because audience validation is mandatory: OAuth Security BCP requires access tokens to be audience-restricted to a specific resource server (or, if not feasible, a small set) and requires every resource server to verify the intended audience on every request (RFC 9700). Resource Indicators (RFC 8707) similarly assume the client requests a token for the location where it intends to use it, and warns that access tokens that are valid for multiple resources/audiences require high trust between recipients.

If you mint a token with `aud=https://gw.example.com/mcp/acme` (because the client is calling that gateway endpoint) and then you forward that *same bearer token* to an upstream MCP server that expects `aud=https://mcp-acme.example.com/mcp`, the upstream will (correctly) deny the request on audience mismatch. That is not a bug; it is the mechanism working as designed.

In practice, you have three deployable patterns. The "most correct" choice is not a philosophical one: it depends on whether you want the upstream to be a first-class OAuth resource server, and how hard you want to push defense-in-depth.

Pattern P1: Gateway terminates OAuth; upstream is private This is the classic API gateway model: the gateway is the resource server, and the upstream is an internal component. The access token is minted for the gateway's canonical URL (the URL the client actually calls), and only the gateway performs OAuth validation. The upstream is protected by network-layer controls (mTLS, ACLs, private subnets) and trusts the gateway.

Key properties:

- Tokens are usable only at the gateway URLs (path-scoped resource identifiers are fine).
- Upstreams never accept client-issued bearer tokens directly (they either never see them, or they ignore them).
- Compromise of an upstream is less valuable because it cannot be accessed directly from outside the gateway boundary.

Illustrative trace (resource indicators + tool-scoped scope):

Agent -> AS: token_exchange (target GW MCP resource)

```
POST /oauth2/token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=urn:ietf:params:oauth:grant-type:token-exchange&
subject_token=AT_agent&
subject_token_type=urn:ietf:params:oauth:token-type:access_token&
resource=https://gw.example.com/mcp/acme&
scope=mcp.call_tool inventory.get quote.read&
intent_id=ord-2026-000125
```

AS -> Agent: AT_mcp_gw (decoded JWT sketch)

```
{
  "iss":"https://as.example.com",
  "sub":"agent_runtime",
  "aud":"https://gw.example.com/mcp/acme",
  "exp":1760669100,
  "scope":"mcp.call_tool inventory.get quote.read",
  "tool_permissions":[
    {"tool":"inventory.get","actions":["invoke"]},
    {"tool":"quote.read","actions":["invoke"]}
  ],
  "act":{"sub":"agent_runtime","typ":"service"}
}
```

Agent -> GW (/mcp/acme): tools/call (token targets GW)

```
POST /mcp/acme HTTP/1.1
Host: gw.example.com
Authorization: Bearer AT_mcp_gw
Content-Type: application/json

{
  "jsonrpc":"2.0",
  "id":"p1-1",
  "method":"tools/call",
  "params":{
    "name":"inventory.get",
    "arguments":{"sku":"X-42"}
  }
}
```

At this point the GW enforces `aud` (exact match against the canonical GW URL) and tool scope (exact match against `inventory.get`). The GW then proxies the request to the upstream MCP server over a private channel (mTLS, ACLs). The upstream does *not* need to validate the client token because, in this pattern, it is not an OAuth resource server.

Pattern P2: Gateway performs token exchange for the upstream resource If you want the upstream to also validate OAuth tokens (defense-in-depth, stronger tenant boundaries, or independent upstream ownership), the GW can act as a confidential OAuth client and mint an *upstream-specific* token via RFC 8693 token exchange.

Operationally, the GW:

1. validates the inbound token for its own resource identifier,
2. performs token exchange using the inbound token as `subject_token`,
3. requests a new token with `resource=<upstream URL>` and a strict subset of scopes,
4. forwards to the upstream with the exchanged token.

This is explicitly covered by RFC 8693: a resource server can assume the role of client and exchange an incoming access token for a token usable at a backend service.

Illustrative trace:

GW -> AS: token_exchange (mint upstream token)

```
POST /oauth2/token HTTP/1.1
Host: as.example.com
Authorization: Basic Z3c6Li4u  % gw client creds
Content-Type: application/x-www-form-urlencoded

grant_type=urn:ietf:params:oauth:grant-type:token-exchange&
subject_token=AT_mcp_gw&
subject_token_type=urn:ietf:params:oauth:token-type:access_token&
resource=https://mcp-acme.internal.example.net/mcp&
scope=mcp.call_tool inventory.get&
intent_id=ord-2026-000125
```

AS -> GW: AT_mcp_upstream (decoded JWT sketch)

```
{
  "iss":"https://as.example.com",
  "sub":"agent_runtime",
  "aud":"https://mcp-acme.internal.example.net/mcp",
  "exp":1760668860,
  "scope":"mcp.call_tool inventory.get",
  "act":{"sub":"gw.example.com","typ":"pepgateway"}
}
```

GW -> Upstream MCP: tools/call (token targets upstream)

```
POST /mcp HTTP/1.1
Host: mcp-acme.internal.example.net
Authorization: Bearer AT_mcp_upstream
Content-Type: application/json
```

```
{
  "jsonrpc":"2.0",
  "id":"p2-1",
  "method":"tools/call",
  "params":{
    "name":"inventory.get",
    "arguments":{"sku":"X-42"}
  }
}
```

Security notes for P2:

- Treat token exchange as a privileged operation. Enforce allowlists (which clients can exchange, which targets are allowed) and strict downscoping. Otherwise token exchange becomes a lateral movement primitive.
- Cache exchanged tokens aggressively (short TTL, per (subject, intent, target, scope) tuple) to control latency and AS load.
- Prefer sender-constrained tokens (mTLS/DPoP) for the GW-to-upstream token to reduce replay risk.

Pattern P3: Mint a multi-audience access token (`aud[]` contains GW and upstream) JWT permits `aud` to be an array, and RFC 9068 carries JWT semantics into access tokens. Some deployments mint a single token that is valid both at the GW URL and at the upstream URL:

```
"aud": ["https://gw.example.com/mcp/acme", "https://mcp-acme.example.com/mcp"]
```

This makes it possible for both the GW and the upstream to accept the same token. It can be practical, but it expands the blast radius of token leakage: any holder can potentially use the token at *either* audience. RFC 8707 explicitly warns that multi-audience access tokens require a high level of trust between the audiences, and RFC 9700 recommends audience restriction to a specific resource server (or a small set) as a core mitigation against token phishing and leakage.

If you adopt P3, treat it as an optimization for controlled environments, and combine it with strong mitigations:

- Upstreams must not be publicly reachable (otherwise you enable bypass of the gateway).
- Use sender-constrained tokens where possible.
- Keep TTLs short and bind tokens to a transaction window (`intent_id`, `jti`, etc).
- Use resource-qualified tool permissions (see Section 6.3) to avoid confused-deputy behavior when `aud[]` covers multiple protected resources.

Minimum enforcement at PEP #1 (Agent Gateway)

1. Verify JWT signature and standard claims (`iss`, `exp`/`nbf`, `iat`) and reject unsigned/invalid tokens.
2. Compute the expected resource identifier for the incoming request and require an **exact match** with one of the token audiences:
   - Separate gateways: expected `aud` = `https://agent-gw.example.com`
   - Shared gateway: expected `aud` = `https://gw.example.com/agent`
3. (Optional but practical) Require a coarse scope like `agent.invoke` to prevent accidental use of unrelated tokens.

No parsing of MCP tool payload is required at PEP #1. Tool-level authorization is enforced at PEP #2.

5.2 Deny-path diagram (tool mismatch)

This is the exact failure mode we want to make boring.

```
JWT (signed) says:   tool_permissions = {list.accounts}
MCP payload says:    tools/call name = payments.transfer

              +--------------------+
              |  MCP Gateway (PEP) |
              +--------------------+
                | verify JWT sig
                | check aud
                | extract tool name
                v
           +-------------------------+
           | requested_tool in set ? |
           +-------------------------+
               | YES -> forward
               |
               | NO
               v
            DENY (403 / JSON-RPC error)
```

5.3 Sequence view (ALLOW and DENY)

ALLOW:

```
(0) Client -> AS    : get AT_agent (resource=agent, intent=accounts.read)
(1) Client -> Agent : call /agent (Authorization: Bearer AT_agent)
(2) Agent -> AS     : token exchange (subject_token=AT_agent,
                      resource=mcp-gw, scope=list.accounts)
(3) AS -> Agent     : AT_mcp (aud=mcp-gw, tool_permissions=[list.accounts])
(4) Agent -> MCP GW : tools/call name=list.accounts (Authorization: Bearer AT_mcp)
(5) MCP GW          : validate + authorize => ALLOW
(6) MCP GW -> MCP   : forward
```

DENY (tool mismatch):

```
(4) Agent -> MCP GW : tools/call name=payments.transfer (Authorization: Bearer AT_mcp)
(5) MCP GW          : validate ok, authorize fails => DENY
(6) MCP GW -> Agent : error (insufficient_tool_scope)
```

6. The authorization contract

Assume JWT access tokens. We define a minimal contract; you can add more claims as needed.

6.1 Token used to call the agent (AT_agent)

- `aud`: canonical **agent gateway** URL (resource indicator, RFC 8707). If you run a shared gateway, use a path-scoped resource such as `https://gw.example.com/agent`.
- `client_id` (or `azp`): identifies the backend client.
- Optional: `intent` claim (string), `intent_id` (UUID).

6.2 Token used by the agent to call MCP tools (AT_mcp)

- `aud`: canonical MCP gateway URL (resource indicator).
- Tool permissions:
  - `scope` includes allowed tool names (space-delimited, exact), and/or
  - `tool_permissions` as a structured claim.

Structured claim example (preferred for safety):

```
"tool_permissions": [
  { "tool": "list.accounts", "actions": ["invoke"] },
  { "tool": "accounts.get", "actions": ["invoke"] }
]
```

Example decoded JWT payload for AT_mcp (signature omitted):

```
{
  "iss": "https://as.example.com",
  "sub": "client_backend_app",
  "aud": "https://mcp-gw.example.com/mcp",
  "iat": 1760668800,
  "exp": 1760669100,
  "scope": "list.accounts accounts.get",
  "tool_permissions": [
    {"tool":"list.accounts","actions":["invoke"]},
    {"tool":"accounts.get","actions":["invoke"]}
  ],
  "intent": "accounts.read",
  "intent_id": "e1b2f3c4-5d6e-7a8b-9c0d-1e2f3a4b5c6d",
  "azp": "client_backend_app",
  "act": { "sub": "agent_runtime", "typ": "service" }
}
```

6.3 Multi-resource transactions: multi-audience (`aud[]`) access tokens (valid, optional)

A single agentic "business action" often spans multiple MCP protected resources. Example: read accounts from an "accounts MCP" (MCP-A) and then execute a payment via a "payments MCP" (MCP-B). The naive implementation requests (or exchanges for) one token per MCP server. That is portable, but it creates token churn and complexity in the agent runtime.

In JWT, the `aud` (audience) claim MAY be either a single string or an array of strings. A resource server validates `aud` by checking whether its own identifier is present (membership), not by assuming `aud` is always a single value. This is defined by JWT (RFC 7519) and carried into the JWT access token profile (RFC 9068).

In OAuth terms, MCP 2025-11-25 relies on Resource Indicators (RFC 8707): the client asks the Authorization Server (AS) for a token bound to a protected resource by sending a `resource` parameter. RFC 8707 allows multiple `resource` parameters in a single request, but it does not mandate that the AS must issue a single token that covers all resources. What comes back is an AS policy decision:

- the AS may issue one token per resource (most portable),
- or it may issue a single JWT access token with `aud` as an array covering multiple canonical resource identifiers,
- or it may reject multi-resource requests.

This section treats `aud[]` tokens as a valid, optional optimization that can be justified in well-controlled environments, while still preserving least privilege by binding permissions to (resource, tool).

6.3.1 Why `aud[]` is worth considering in controlled environments In a hardened, backend-to-backend environment (confidential clients, strict network boundaries, short-lived tokens), `aud[]` can reduce operational friction without weakening the policy model:

- Less token churn: fewer token endpoint round-trips during a transaction.
- Lower AS load: fewer exchanges during bursty agent workloads.
- Simpler agent state: less per-resource token caching logic, fewer race conditions when a plan requires multiple downstream hops.
- Cleaner "step-up": the resource can challenge for extra scope once, and the resulting token can cover the set of MCP servers needed for the current intent window.

The goal is not to create a "fat" token. The goal is to mint a short-lived, transaction-scoped token that is still tightly constrained by tool permissions.

6.3.2 Standards view: what is "valid" vs what is "portable"

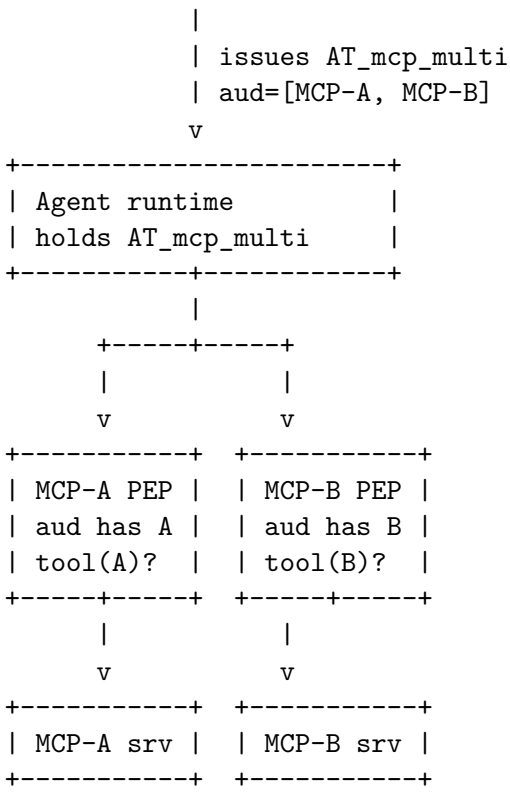
- Valid (JWT): `aud` can be an array. Any JWT consumer must handle that.
- Valid (RFC 9068): JWT access tokens follow JWT semantics, including `aud` as string-or-array.
- Optional (RFC 8707): multiple `resource` parameters MAY appear, but AS behavior is not fixed. If you need portability across IdPs, assume you will need per-resource tokens unless you control the AS and the resource servers.

Practical interpretation: `aud[]` is a deployable pattern when you control the AS (or at least its policy behavior) and you are willing to treat it as an optimization that can be disabled per environment.

6.3.3 Reference architecture for a multi-resource transaction token The contract is intentionally strict: multi-resource `aud[]` tokens MUST also carry resource-qualified tool permissions, otherwise a permissive tool list becomes a confused-deputy enabler.

```
(OBO / Token Exchange)

+------------------------+
| Authorization Server   |
| (PDP: roles -> tools)  |
+-----------+------------+
```



6.3.4 Example: multi-resource OBO (Token Exchange) request Below is an illustrative RFC 8693 Token Exchange request used as OBO. The agent exchanges `AT_agent` for a short-lived `AT_mcp_multi` and asks the AS for two protected resources in one request by repeating `resource=`.

Important: this is not guaranteed to work on every AS. It is a policy-driven feature. If unsupported, fall back to per-resource exchanges (Section 8.10).

```
POST /token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic <agent-client-credentials>

grant_type=urn:ietf:params:oauth:grant-type:token-exchange&
subject_token_type=urn:ietf:params:oauth:token-type:access_token&
subject_token=<AT_agent>&
requested_token_type=urn:ietf:params:oauth:token-type:access_token&
resource=https://mcp-a.example.com/mcp&
resource=https://mcp-b.example.com/mcp&
scope=list.accounts payments.transfer
```

If issued as a JWT access token, the decoded payload can look like:

```
{
  "iss":"https://as.example.com",
  "sub":"client_backend_app",
  "aud":[
    "https://mcp-a.example.com/mcp",
```

```
    "https://mcp-b.example.com/mcp"
  ],
  "iat":1760668800,
  "exp":1760669100,
  "intent":"balance-and-transfer",
  "intent_id":"8c9d7e6f-1111-2222-3333-444455556666",
  "tool_permissions":[
    {"rs":"https://mcp-a.example.com/mcp","tool":"list.accounts","actions":["invoke"]},
    {"rs":"https://mcp-b.example.com/mcp","tool":"payments.transfer","actions":["invoke"]}
  ],
  "act":{ "sub":"agent_runtime", "typ":"service" }
}
```

6.3.5 Example: two MCP calls with the same `AT_mcp_multi` (ALLOW) Call 1 to MCP-A:

```
{
  "jsonrpc":"2.0",
  "id":1101,
  "method":"tools/call",
  "params":{
    "name":"list.accounts",
    "arguments":{"limit":10}
  }
}
```

MCP-A PEP evaluation:

```
resource:       https://mcp-a.example.com/mcp
aud contains:   YES
requested:      list.accounts
permitted(rs):  {list.accounts}
result:         ALLOW
```

Call 2 to MCP-B:

```
{
  "jsonrpc":"2.0",
  "id":1102,
  "method":"tools/call",
  "params":{
    "name":"payments.transfer",
    "arguments":{
      "from_account":"ES00-1234",
      "to_account":"ES99-9876",
      "amount":"250.00",
      "currency":"EUR"
    }
  }
}
```

MCP-B PEP evaluation:

```
resource:       https://mcp-b.example.com/mcp
aud contains:   YES
requested:      payments.transfer
permitted(rs):  {payments.transfer}
result:         ALLOW
```

6.3.6 Example: same token, wrong tool on the wrong resource (DENY) A common agent failure mode is mixing "tool intent" with "resource target". Even with `aud[]`, the token must not allow cross-resource tool leakage.

Agent sends `payments.transfer` to MCP-A by mistake:

```
resource:       https://mcp-a.example.com/mcp
aud contains:   YES
requested:      payments.transfer
permitted(rs):  {list.accounts}
result:         DENY (insufficient_tool_scope)
```

This is the key property: multi-audience does not imply "tools are global".

6.3.7 DENY-by-default guardrail: reject flat tool lists when `aud` has multiple resources If you adopt `aud[]`, treat a token as malformed (or deny) if it contains multiple audiences but does not bind tools to resources. Otherwise, a caller could smuggle a permissive tool list that accidentally authorizes side effects on the wrong resource.

Recommended rule:

```
if aud is array with size > 1 AND tool_permissions are not resource-qualified:
  DENY (invalid_token / invalid_scope_contract)
```

This is strict, but it prevents a class of confused-deputy mistakes.

6.3.8 Alias audiences: one logical resource, multiple canonical identifiers Sometimes you want multiple `aud` values not because you have multiple resources, but because the same gateway has more than one stable identifier (internal DNS and external DNS, or a legacy base path and a new base path).

Example:

```
"aud":[
  "https://mcp-gw.internal.example.com/mcp",
  "https://mcp-gw.example.com/mcp"
]
```

This is usually lower risk than spanning unrelated resources, but it still requires strict canonicalization in the PEP:

- normalize scheme/host casing,
- normalize trailing slash rules,
- map inbound Host + path to one canonical resource identifier,
- then validate membership against `aud`.

6.3.9 Alternatives when you need richer constraints

6.4 One gateway, many MCP servers: binding tools to the selected upstream

The "aud is the resource, tools are in scope" contract is cleanest when a token targets exactly one MCP protected resource. In practice, teams often front multiple MCP servers behind a single gateway endpoint to simplify client configuration and to keep network and auth controls centralized.

Two deployment variants exist:

Variant 1: Gateway is the only protected resource (`aud` = gateway) In this model the gateway is the OAuth protected resource. The gateway validates the token (`aud` == gateway resource identifier) and then enforces tool access purely by `scope` membership. The tool-to-upstream mapping is a gateway configuration concern, not a token concern.

This works best when the gateway exposes a unified tool namespace where each tool name is globally unique, for example by prefixing tools with a target identifier:

- `bank.list_accounts`
- `bank.payments_transfer`
- `crm.search_customers`

The PEP check stays simple: requested tool name must match a scope token exactly.

Variant 2: Multiple upstream protected resources behind one gateway (multi-resource semantics) In this model the gateway still terminates traffic, but you want the IdP policy to remain upstream-aware (for example, to separate "bank MCP" from "crm MCP" as independent resources). You can keep tokens short-lived and still avoid running a PDP in the gateway by binding each tool permission to an upstream resource identifier using an additional claim.

Recommended claim shape:

```
"mcp_toolset": [
  {"rs":"https://mcp-bank.example.com/mcp","tools":["accounts.list","payments.transfer"]},
  {"rs":"https://mcp-crm.example.com/mcp","tools":["customers.search"]}
]
```

Enforcement rule at the gateway:

1) Determine which upstream resource (`rs_selected`) the tool invocation will be routed to (based on tool prefix, a session map, or routing metadata).
2) Validate `aud` membership for the gateway resource (or allow `aud[]` membership if you explicitly model upstream resources in `aud`).
3) Validate tool permission by selecting the matching `mcp_toolset` entry for `rs_selected` and checking exact match of the tool name.

This gives you a strict binding between tool permissions and upstream MCP servers without a runtime policy lookup.

Mandatory guardrail when multiplexing If a token is usable against more than one logical resource, treat it as malformed unless tool permissions are resource-qualified (tool list must be tied to an `rs`). This prevents "flat tool list" confused-deputy failures where a permissive set of tools accidentally authorizes side effects on the wrong upstream.

If you need per-resource constraints beyond "tool allow-list" (amount limits, account ranges, region,

tenant, time windows), prefer structured authorization:

- Rich Authorization Requests (RFC 9396) `authorization_details`, or
- per-resource tokens via Token Exchange (RFC 8693) (most portable), or
- gateway multiplexing (Section 12.4): one canonical audience, per-upstream policy inside the gateway.

`aud[]` remains useful as a pragmatic optimization when the environment is controlled and the guardrails above are enforced.

7. Enforcement: exact tool match between token and MCP payload

7.1 Extracting the tool name from MCP requests

For streamable HTTP MCP, tool calls look like JSON-RPC `tools/call` with:

- `method = "tools/call"`
- `params.name = "<tool-name>"`

Example request:

```
POST /mcp HTTP/1.1
Host: mcp-gw.example.com
Authorization: Bearer <AT_mcp>
Content-Type: application/json

{
  "jsonrpc":"2.0",
  "id":101,
  "method":"tools/call",
  "params":{
    "name":"list.accounts",
    "arguments":{"limit":10}
  }
}
```

7.2 PEP decision algorithm (reference)

```
INPUT: jwt, mcp_request

1) Verify JWT signature (kid -> JWKS), iss, exp/nbf
2) Verify audience and resource binding:
   - Compute the canonical protected resource identifier for *this* request.
     Recommended: scheme=https, lowercase host, normalize default ports, and normalize the MCP
   - aud_set = {jwt.aud} if jwt.aud is a string else set(jwt.aud)
   - require canonical_resource_id in aud_set (membership check; aud MAY be an array)
3) Parse MCP request:
   - require jsonrpc == "2.0"
   - require method in {"tools/list","tools/call"}
4) If method == "tools/call":
   - requested_tool = params.name
   - allowed = extract permissions from tool_permissions OR scope-derived set
```

```
   - if tool_permissions are resource-qualified (e.g., each entry has `rs`):
       - allowed = { t.tool | t.rs == canonical_mcp_gateway_url }
   - if requested_tool in allowed => ALLOW
     else => DENY (insufficient_tool_scope)
5) If method == "tools/list":
   - allow but filter the tool list to allowed_tools (ALLOW+FILTER), or deny
```

8. Worked examples (deterministic allow/deny)

Each example shows the decoded JWT payload and the MCP JSON-RPC request. Under each, a compact table highlights the decision-critical fields in **bold**.

8.1 Example A (ALLOW): correct token and correct tool call

Sequence:

```
client backend -> agent runtime -> MCP gateway : tools/call list.accounts (AT_mcp)
MCP gateway -> MCP server : forward
MCP server -> MCP gateway : result
MCP gateway -> agent runtime -> client backend : result
```

Decoded JWT payload (illustrative; signature omitted):


```
{
  "iss":"https://as.example.com",
  "sub":"client_backend_app",
  "aud":"https://mcp-gw.example.com/mcp",
  "exp":1760669100,
  "iat":1760668800,
  "jti":"8ddc2a5b-5e0f-4c2f-88f3-5d1a9b8f12a1",
  "intent":"accounts.read",
  "tool_permissions":[
    {"tool":"list.accounts","actions":["invoke"]}
  ]
}
```


Decision inputs (token highlights):

Claim	Value
iss	`https://as.example.com`
sub	`client_backend_app`
aud	`https://mcp-gw.example.com/mcp`
exp	`1760669100`
intent	`accounts.read`
tool_permissions	`list.accounts` (`invoke`)

MCP request:


```
{
  "jsonrpc":"2.0",
```

```
  "id":1,
  "method":"tools/call",
  "params":{
    "name":"list.accounts",
    "arguments":{"limit":10}
  }
}
```

Decision inputs (request highlights):

Field	Value
method	`tools/call`
params.name	`list.accounts`
params.arguments.limit	`10`

Decision:

```
aud matches:  YES
requested:    list.accounts
permitted:    {list.accounts}
result:       ALLOW
```

8.2 Example B (DENY): token permits list.accounts, agent tries payments.transfer

Decoded JWT payload (same policy as Example A):

```
{
  "iss":"https://as.example.com",
  "sub":"client_backend_app",
  "aud":"https://mcp-gw.example.com/mcp",
  "exp":1760669100,
  "intent":"accounts.read",
  "tool_permissions":[
    {"tool":"list.accounts","actions":["invoke"]}
  ]
}
```

Decision inputs (token highlights):

Claim	Value
aud	`https://mcp-gw.example.com/mcp`
tool_permissions	`list.accounts` (`invoke`)

MCP request (agent attempts an unauthorized side effect):

```
{
  "jsonrpc":"2.0",
  "id":2,
```

```
  "method":"tools/call",
  "params":{
    "name":"payments.transfer",
    "arguments":{
      "from_account":"ES00-1234",
      "to_account":"ES99-9876",
      "amount":"250.00",
      "currency":"EUR"
    }
  }
}
```

Decision inputs (request highlights):

Field	Value
method	`tools/call`
params.name	`payments.transfer`

PEP decision:

```
aud matches:  YES
requested:    payments.transfer
permitted:    {list.accounts}
result:       DENY (insufficient_tool_scope)
```

Example JSON-RPC error (illustrative):

```
{
  "jsonrpc":"2.0",
  "id":2,
  "error":{
    "code":-32603,
    "message":"unauthorized tool call",
    "data":{
      "reason":"insufficient_tool_scope",
      "requested_tool":"payments.transfer",
      "permitted_tools":["list.accounts"],
      "aud":"https://mcp-gw.example.com/mcp",
      "intent":"accounts.read"
    }
  }
}
```

8.3 Example C (DENY): token permits list.accounts, payload uses payments.payment

Decoded JWT payload:

```
{
  "iss":"https://as.example.com",
  "sub":"client_backend_app",
```

```
  "aud":"https://mcp-gw.example.com/mcp",
  "exp":1760669100,
  "tool_permissions":[
    {"tool":"list.accounts","actions":["invoke"]}
  ]
}
```


MCP request:


```
{
  "jsonrpc":"2.0",
  "id":3,
  "method":"tools/call",
  "params":{
    "name":"payments.payment",
    "arguments":{
      "amount":"12.34"
    }
  }
}
```


Decision inputs (highlights):

Field	Value
aud	`https://mcp-gw.example.com/mcp`
params.name	`payments.payment`
tool_permissions	`list.accounts` (`invoke`)

Decision:

```
aud matches: YES
requested:   payments.payment
permitted:   {list.accounts}
result:      DENY
```

8.4 Example D (DENY): scope-string mismatch (JWT says list.accounts, payload says payments.transfer)

Some deployments encode tool permissions directly in the standard OAuth `scope` claim (RFC 6750 / RFC 9068 style), using tool names as scopes.

Decoded JWT payload (illustrative):


```
{
  "iss":"https://as.example.com",
  "sub":"client_backend_app",
  "aud":"https://mcp-gw.example.com/mcp",
  "exp":1760669100,
  "scope":"list.accounts",
  "intent":"accounts.read",
```

```
  "client_id":"backend-billing-service"
}
```

MCP request:

```
{
  "jsonrpc":"2.0",
  "id":5,
  "method":"tools/call",
  "params":{
    "name":"payments.transfer",
    "arguments":{
      "from_account":"ES00-1234",
      "to_account":"ES99-9876",
      "amount":"250.00",
      "currency":"EUR"
    }
  }
}
```

Decision inputs (highlights):

Field	Value
aud	`https://mcp-gw.example.com/mcp`
scope	`list.accounts`
params.name	`payments.transfer`

Decision:

```
aud matches:   YES
scope tokens:  {list.accounts}
requested:     payments.transfer
result:        DENY
```

Example response (HTTP 403 + RFC 6750 style challenge, plus JSON-RPC error body):

```
HTTP/1.1 403 Forbidden
WWW-Authenticate: Bearer error="insufficient_scope",
  scope="payments.transfer",
  resource="https://mcp-gw.example.com/mcp"
Content-Type: application/json

{
  "jsonrpc":"2.0",
  "id":5,
  "error":{
    "code":-32603,
    "message":"unauthorized tool call",
    "data":{
      "reason":"insufficient_scope",
```

```
      "requested_tool":"payments.transfer",
      "permitted_scopes":["list.accounts"]
    }
  }
}
```

ASCII view of the check (exact match, no substring tricks):

```
JWT scope:  "list.accounts"
Payload:    tools/call name="payments.transfer"

PEP rule: requested_tool IN split(scope, " ")
```

8.5 Example E (ALLOW+FILTER): tools/list filtering to reduce enumeration

If you allow `tools/list`, do not leak the full catalog.

Decoded JWT payload (illustrative):


```
{
  "iss":"https://as.example.com",
  "sub":"client_backend_app",
  "aud":"https://mcp-gw.example.com/mcp",
  "exp":1760669100,
  "tool_permissions":[
    {"tool":"list.accounts","actions":["invoke","list"]}
  ]
}
```


Request:


```
{"jsonrpc":"2.0","id":4,"method":"tools/list","params":{}}
```


Decision inputs (highlights):

Field	Value
method	`tools/list`
aud	`https://mcp-gw.example.com/mcp`
tool_permissions	`list.accounts` (`list`)

Filtered response (only tools allowed by policy):


```
{
  "jsonrpc":"2.0",
  "id":4,
  "result":{
    "tools":[
      {
        "name":"list.accounts",
        "description":"List accounts",
        "inputSchema":{"type":"object"}
```

```
      }
    ]
  }
}
```

8.6 Example F (DENY): scope-string pitfalls and the fix

Avoid checks like:

```
if jwt.scope contains requested_tool => allow
```

This is unsafe (substring risk). Fix by using either:

- a structured claim (`tool_permissions`) and exact matching, or
- strict tokenization (`scope.split(" ")`) and exact matching.

8.7 Example G (ALLOW): one token, two MCP servers via `aud[]` and resource-bound tool permissions

Assume the agent must call **two different MCP protected resources** within a single transaction:

- MCP-A (read-only): `https://mcp-a.example.com/mcp`
- MCP-B (side effects): `https://mcp-b.example.com/mcp`

The Authorization Server issues a single `AT_mcp_multi` with:

- `aud` as an array containing both canonical resource identifiers, and
- `tool_permissions` entries explicitly bound to a resource (`rs`) and tool.

Decoded JWT payload (illustrative):

```
{
  "iss":"https://as.example.com",
  "sub":"agent_runtime",
  "aud":[
    "https://mcp-a.example.com/mcp",
    "https://mcp-b.example.com/mcp"
  ],
  "exp":1760669100,
  "tool_permissions":[
    {"rs":"https://mcp-a.example.com/mcp","tool":"list.accounts","actions":["invoke"]},
    {"rs":"https://mcp-b.example.com/mcp","tool":"payments.transfer","actions":["invoke"]}
  ]
}
```

Decision inputs (token highlights):

Claim	Value
aud	`[MCP-A, MCP-B]`
tool_permissions	`(MCP-A, list.accounts), (MCP-B, payments.transfer)`

Request 1 (MCP-A):

{"jsonrpc":"2.0","id":7,"method":"tools/call","params":{"name":"list.accounts","arguments":{"li

Request 2 (MCP-B):

{"jsonrpc":"2.0","id":8,"method":"tools/call","params":{"name":"payments.transfer","arguments"

Sequence (two calls, same token):

```
agent -> MCP-A gateway : tools/call list.accounts (AT_mcp_multi)
MCP-A gateway : check aud contains MCP-A AND tool allowed for rs=MCP-A
agent -> MCP-B gateway : tools/call payments.transfer (AT_mcp_multi)
MCP-B gateway : check aud contains MCP-B AND tool allowed for rs=MCP-B
```

Decision:

```
call 1: ALLOW
call 2: ALLOW
```

8.8 Example H (DENY): `aud[]` includes MCP-B, but the tool is not permitted for MCP-B

Token permits only `list.accounts` on MCP-A, but includes both audiences:


```
{
  "iss":"https://as.example.com",
  "sub":"agent_runtime",
  "aud":[
    "https://mcp-a.example.com/mcp",
    "https://mcp-b.example.com/mcp"
  ],
  "exp":1760669100,
  "tool_permissions":[
    {"rs":"https://mcp-a.example.com/mcp","tool":"list.accounts","actions":["invoke"]}
  ]
}
```


Decision inputs (token highlights):

Claim	Value
aud	`[MCP-A, MCP-B]`
tool_permissions	`(MCP-A, list.accounts)`

Agent attempts to call a side-effect tool on MCP-B:


```
{
  "jsonrpc":"2.0",
  "id":8,
  "method":"tools/call",
  "params":{
    "name":"payments.transfer",
    "arguments":{
      "from_account":"ES00-1234",
```

```
      "to_account":"ES99-9876",
      "amount":"250.00"
    }
  }
}
```

Decision inputs (request highlights):

Field	Value
params.name	`payments.transfer`
target resource	`https://mcp-b.example.com/mcp`

Decision:

```
aud contains MCP-B:      YES
tools bound to MCP-B:    {}
requested tool:          payments.transfer
result:                  DENY (insufficient_tool_scope)
```

8.9 Example I (DENY): agent reuses `AT_mcp_multi` against an unlisted MCP server (audience mismatch)

Token permits two audiences only:

```
{
  "iss":"https://as.example.com",
  "sub":"agent_runtime",
  "aud":[
    "https://mcp-a.example.com/mcp",
    "https://mcp-b.example.com/mcp"
  ],
  "exp":1760669100
}
```

Agent calls a third MCP server not listed in `aud[]`:

```
POST /mcp HTTP/1.1
Host: mcp-c.example.com
Authorization: Bearer <AT_mcp_multi>

{ "jsonrpc":"2.0", "id":9, "method":"tools/call", "params":{ "name":"list.accounts" } }
```

Decision inputs (highlights):

Field	Value
expected resource	`https://mcp-c.example.com/mcp`
aud	`[MCP-A, MCP-B]`

Decision:

```
aud contains expected resource: NO
result:                         DENY (invalid_token / invalid_audience)
```

9. OBO downscoping with Token Exchange (RFC 8693)

The agent should not reuse the client's token directly against the MCP gateway. Instead, the agent performs a **standards-based on-behalf-of (OBO)** flow using **OAuth 2.0 Token Exchange** (RFC 8693):

- `AT_agent` represents the upstream **subject** (the backend client and its intent).
- The agent runtime is the **actor** performing the exchange (identified at least by OAuth client authentication, optionally also via an `actor_token`).
- The issued token `AT_mcp` is (a) **audience-bound** to the MCP resource via RFC 8707 `resource`, and (b) **downscoped** to the minimal tool set for the intent.

RFC 8693 models delegation explicitly. When the issued token is a JWT, the Authorization Server can represent delegation using an `act` claim (current actor) and/or `may_act` (authorized actors). That makes auditing deterministic: a downstream gateway can distinguish "who the token is about" (`sub`) from "who is acting right now" (`act`).

Illustrative token exchange request (agent authenticates as a confidential client):

```
POST /token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic <agent-client-credentials>

grant_type=urn:ietf:params:oauth:grant-type:token-exchange&
subject_token_type=urn:ietf:params:oauth:token-type:access_token&
subject_token=<AT_agent>&
resource=https://mcp-gw.example.com/mcp&
scope=list.accounts
```

Optional: make the actor explicit (some deployments prefer this for clearer delegation chains):

```
...&
actor_token_type=urn:ietf:params:oauth:token-type:access_token&
actor_token=<AT_agent_runtime_identity>
```

Illustrative issued token shape (decoded JWT payload; signature omitted):

```
{
  "iss": "https://as.example.com",
  "sub": "client_backend_app",
  "aud": "https://mcp-gw.example.com/mcp",
  "exp": 1760669100,
  "scope": "list.accounts",
  "intent": "accounts.read",
  "azp": "client_backend_app",
  "act": { "sub": "agent_runtime", "typ": "service" }
}
```

PDP rules the AS should enforce (non-negotiable):

- The agent client is allowed to use token exchange.
- The requested `resource` is an approved downstream resource for that agent.
- Requested tools are a subset of what the subject token implies for the intent and client role (**monotonic downscoping**).
- Delegation is recorded in the issued token (e.g., `act` and correlation identifiers like `jti` / `intent_id`). ### 9.1 Token Exchange in one page: the parameters that matter for agents

OAuth 2.0 Token Exchange (RFC 8693) is often described as "OBO", but it is more precise to call it **delegation + downscoping**:

- **Delegation:** an actor (the agent runtime) requests a new token to act on behalf of a subject (the backend client, or a user in other scenarios).
- **Downscoping:** the new token is intentionally "smaller" (resource/audience-bound, shorter-lived, fewer scopes/tools) than the input context.

The request is a normal OAuth token request with a specific `grant_type`:

`grant_type=urn:ietf:params:oauth:grant-type:token-exchange`

Core parameters you will almost always use in an agent deployment:

- `subject_token` and `subject_token_type`: the token that represents the party on behalf of whom the exchange is performed (in this paper: `AT_agent`).
- `resource` (RFC 8707): the canonical identifier of the MCP protected resource you want the issued token to be valid for.
- `scope`: the tool permissions being requested (or a coarse intent scope that maps to a structured claim, depending on your design).

Optional parameters that become valuable in real systems:

- `actor_token` and `actor_token_type`: explicitly represent the agent runtime identity as the acting party (some AS products can infer the actor from client authentication, but explicit actor tokens make the chain harder to fake).
- `requested_token_type`: pin the output to an access token (or `urn:ietf:params:oauth:token-type:jwt` when you want a JWT output in a predictable format).
- `audience` (RFC 8693): optional parameter, not equivalent to RFC 8707 `resource`. Some products treat it as a target client selector or map it to `aud`; others ignore it. For MCP, prefer `resource` whenever possible; only use `audience` if your AS requires it and you have a documented mapping to `aud`.

A minimal request for a single MCP resource and one tool:

```
POST /token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic <agent-client-credentials>

grant_type=urn:ietf:params:oauth:grant-type:token-exchange&
subject_token_type=urn:ietf:params:oauth:token-type:access_token&
subject_token=<AT_agent>&
resource=https://mcp-a.example.com/mcp&
scope=list.accounts
```

A typical response (RFC 8693) returns the new token in `access_token` and includes

`issued_token_type`:

```
{
  "access_token": "<AT_mcp>",
  "issued_token_type": "urn:ietf:params:oauth:token-type:access_token",
  "token_type": "Bearer",
  "expires_in": 300,
  "scope": "list.accounts"
}
```

9.2 Two identities, one chain: "subject" vs "actor" and why it matters

If you skip explicit delegation semantics, incident response becomes astrology.

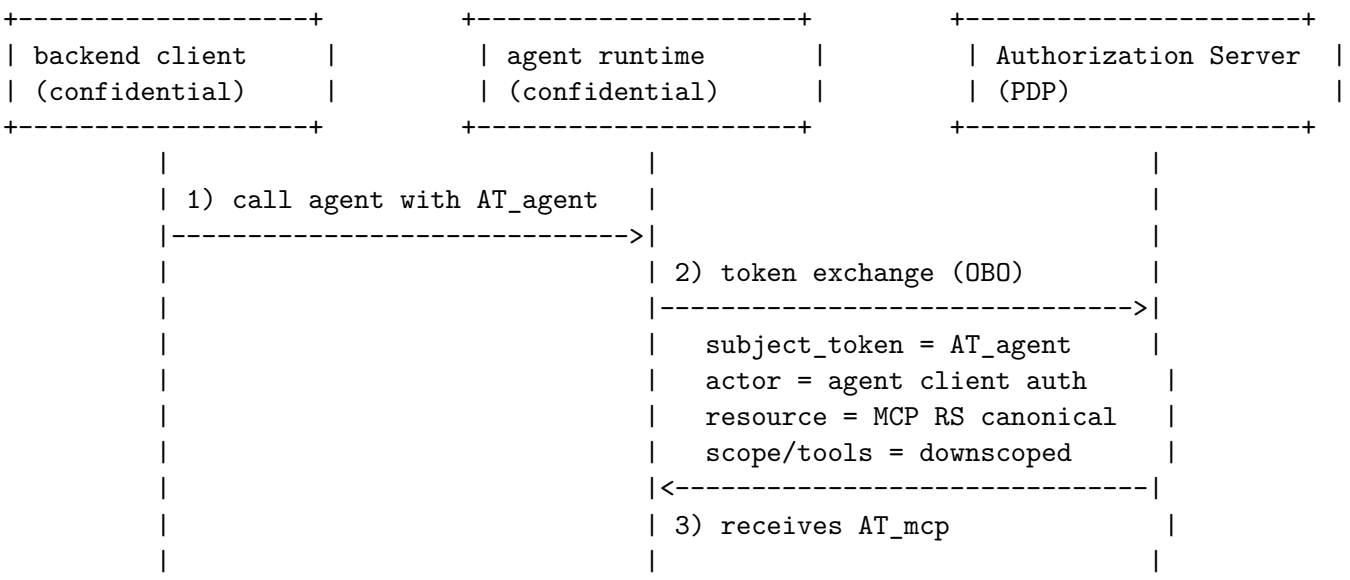
Token Exchange gives you a formal model:

- **Subject:** the identity the token is "about" (`sub` in the issued JWT).
- **Actor:** the identity currently using delegated authority (`act` in the issued JWT, when the AS emits it).

This lets a gateway enforce policy using deterministic claims, and lets you audit:

- "client_backend_app caused the transfer" (subject)
- "agent_runtime executed the call" (actor)

ASCII trace of the OBO chain:



In many deployments, the AS can represent the actor even when there is no `actor_token`, by using the authenticated client identity (the agent) as the actor. If you want stronger non-repudiation, you can require an `actor_token` and bind it to the agent runtime instance identity (service identity, workload identity, or mTLS-attested credential).

9.3 What the AS must enforce as PDP for agent OBO

Token Exchange is only as safe as the PDP logic the Authorization Server applies.

Minimum PDP checks for an MCP tool authorization design:

1) **Actor eligibility:** the agent client is allowed to use the token exchange grant at all.
2) **Delegation boundary:** the agent is allowed to exchange the given subject token (e.g., `sub` and `azp` must match an approved upstream client).
3) **Resource boundary:** each requested `resource` is in the agent's allowlist of downstream protected resources (RFC 8707 identifiers).
4) **Tool boundary:** requested tools/scopes must be a subset of what the upstream context allows (monotonic downscoping).
5) **Lifetime boundary:** issued tokens should be short-lived and scoped to the transaction (minutes, not hours).
6) **Auditability:** the issued token must preserve delegation metadata (at least `act`, plus correlation like `jti` / `intent_id`).

When the agent attempts to "upgrade" permissions, the AS should reject the token exchange request early (e.g., `invalid_scope` or `invalid_target` depending on what is violated) rather than relying on the MCP gateway to catch it later.

Example: the agent tries to request a write tool that was not granted upstream.

```
POST /token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic <agent-client-credentials>

grant_type=urn:ietf:params:oauth:grant-type:token-exchange&
subject_token_type=urn:ietf:params:oauth:token-type:access_token&
subject_token=<AT_agent>&
resource=https://mcp-a.example.com/mcp&
scope=list.accounts payments.transfer
```

Expected AS outcome (illustrative):

```
{
  "error": "invalid_scope",
  "error_description": "Requested scope not permitted for subject token or client role"
}
```

9.4 Multi-resource OBO: repeated RFC 8707 `resource` and why `aud[]` shows up

RFC 8707 explicitly allows multiple occurrences of the `resource` parameter, but also warns that tokens with multiple audiences are higher trust / higher blast radius and are not always supported by AS products. In controlled environments, though, this is a pragmatic optimization:

- a single business transaction may require two MCP resources
- repeatedly calling `/token` adds latency, load, and operational complexity
- a short-lived multi-audience token can keep the transaction single-pass

A multi-resource token exchange request (same transaction, two MCP servers):

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

```
Authorization: Basic <agent-client-credentials>

grant_type=urn:ietf:params:oauth:grant-type:token-exchange&
subject_token_type=urn:ietf:params:oauth:token-type:access_token&
subject_token=<AT_agent>&
resource=https://mcp-a.example.com/mcp&
resource=https://mcp-b.example.com/mcp&
scope=list.accounts payments.transfer
```

If the AS chooses to mint a single JWT access token, it may encode the audience as an array:

```
{
  "iss": "https://as.example.com",
  "sub": "client_backend_app",
  "aud": [
    "https://mcp-a.example.com/mcp",
    "https://mcp-b.example.com/mcp"
  ],
  "exp": 1760669100,
  "scope": "list.accounts payments.transfer",
  "act": { "sub": "agent_runtime" }
}
```

This paper treats `aud[]` as **valid but optional**: it is a performance and ergonomics lever for deployments that control both the AS and the downstream resources (or that explicitly accept the trust assumptions).

The enforcement requirement does not change:

- each PEP checks that its own canonical resource identifier is present in `aud`
- and that the requested tool is permitted for that resource (resource-qualified permissions are strongly recommended)

9.5 Token contents: JWT access token profile (RFC 9068) and delegation claims

If you issue JWT access tokens, align with the JWT access token profile (RFC 9068) so resource servers can validate tokens consistently and avoid accepting ID Tokens by mistake.

Keep it simple:

- Use the `typ` header value `at+jwt` (or `application/at+jwt`) and have the resource server reject anything else.
- Include the required claims (`iss`, `sub`, `aud`, `exp`). Treat `aud` as string-or-array and validate by membership.
- Include `scope` (tool identifiers) and, if you need deterministic delegation/audit, include an `act` claim (RFC 8693) and correlation claims like `jti`.
- Use either `client_id` or `azp` (depending on your ecosystem) to preserve which OAuth client obtained the token.

Decoded JWT header (illustrative):

```
{
  "typ": "at+jwt",
```

```
  "alg": "RS256",
  "kid": "RjEwOwOA"
}
```

A more complete issued token sketch for MCP (payload only; illustrative):

```
{
  "iss": "https://as.example.com",
  "sub": "client_backend_app",
  "client_id": "agent_runtime_client",
  "aud": "https://mcp-gw.example.com/mcp",
  "exp": 1760669100,
  "iat": 1760668800,
  "jti": "8ddc2a5b-5e0f-4c2f-88f3-5d1a9b8f12a1",
  "scope": "list.accounts",
  "intent": "accounts.read",
  "act": {
    "sub": "agent_runtime",
    "typ": "service"
  },
  "tool_permissions": [
    {"rs":"https://mcp-gw.example.com/mcp","tool":"list.accounts","actions":["invoke"]}
  ]
}
```

Important: do not treat `scope` as a free-form string. Tokenize by spaces and do exact comparisons. If you include both `scope` and a structured permission claim (like `tool_permissions`), pick one as the source of truth and make the other redundant (or omit it).

9.6 Hardening checklist for token exchange in agentic systems

When you deploy Token Exchange for agents, assume a hostile environment and design for containment:

- Short TTLs for `AT_mcp` (minutes) and cache only within a bounded transaction/session.
- Reject wildcard scopes for tools unless you have a separate, risk-accepted admin path.
- Consider sender-constrained tokens (mTLS or DPoP) for high-impact tools.
- Require exact `resource` canonicalization and reject non-canonical forms.
- Emit correlation claims (`jti`, `intent_id`) and log them at gateways and MCP servers.
- Decide if you want to allow multi-resource (`resource` repeated -> `aud[]`) and document the trust assumptions explicitly.

9.7 Example 7 (DENY): scope escalation attempt in OBO/token exchange

The safe failure mode is to reject privilege escalation at the Authorization Server (PDP) during token issuance, not after the tool call hits the gateway.

Scenario: the agent tries to exchange a subject token into a downstream MCP token that includes a write/destructive tool (`payments.refund`) that is not allowed by the subject context and policy.

Sequence (token issuance denied early):

```
backend client     -> agent runtime   : call agent (AT_agent, intent=accounts.read)
agent runtime      -> Authorization Server : token exchange
Authorization Server -> agent runtime : DENY (invalid_scope)
```

Token Exchange request (form parameters; illustrative):

```
grant_type=urn:ietf:params:oauth:grant-type:token-exchange
subject_token_type=urn:ietf:params:oauth:token-type:access_token
subject_token=<AT_agent>
resource=https://mcp-gw.example.com/mcp
scope=list.accounts payments.refund
```

Decision inputs (request highlights):

Field	Value	Why it matters
grant_type	`urn:ietf:params:oauth:grant-type:token-exchange`	Selects token exchange flow (RFC 8693)
subject_token	`<AT_agent>`	Upper bound of what the agent can request
resource	`https://mcp-gw.example.com/mcp`	Canonical target MCP resource (RFC 8707)
scope	`list.accounts payments.refund`	Requested tool set (contains an escalation)

Expected AS response (illustrative):

```
{
  "error": "invalid_scope",
  "error_description": "requested permissions exceed subject token and policy",
  "reason": "downscope_violation",
  "policy_version": "2026-02-17.1"
}
```

Decision outputs (response highlights):

Field	Value	Why it matters
error	`invalid_scope`	Clear OAuth failure surface
reason	`downscope_violation`	Deterministic denial reason for audit/debug
policy_version	`2026-02-17.1`	Makes decisions reproducible across rollouts

9.8 Example 8: Multi-tenant and namespace enforcement

Token includes tenant and tenant-scoped tool:

Token includes tenant and tenant-scoped tool

```
{
  "tenant_id":"acme",
  "tool_permissions":[{"tool":"acme.inventory.get","actions":["invoke"]}],
  "aud":"https://mcp-gw.example.com"
}
```

Request for `globex.inventory.get` is denied with `tenant_mismatch`:

MCP request (cross-tenant tool)

```
POST /mcp HTTP/1.1
Host: mcp-gw.example.com
Authorization: Bearer AT_mcp
Content-Type: application/json

{
  "jsonrpc":"2.0",
  "id":"req-9.8-1",
  "method":"tools/call",
  "params":{
    "name":"globex.inventory.get",
    "arguments":{"sku":"X-42"}
  }
}
```

MCP gateway response

```
{
  "error":"access_denied",
  "reason":"tenant_mismatch",
  "token_tenant":"acme",
  "requested_tool":"globex.inventory.get"
}
```

10. Candidate Authorization Servers / IdPs

Keycloak (primary open-source candidate). Keycloak supports token exchange and is a pragmatic baseline for building monotonic downscoping policies with explicit audience discipline.

Microsoft Entra ID. Entra documents the OBO flow and is widely used for middle-tier API delegation. In practice you must be strict about assertion audience and route-level audience pinning when exchanging tokens for downstream resources.

Okta. Okta documents OBO-style delegation for microservice chains via custom authorization servers and service-app exchange controls. The operational emphasis is on clearly scoped policies and exchange allowlists.

Tool-level checks at the gateway are only as good as the policy that minted the token.

10.1 Maintain a canonical tool catalog

Minimum fields:

- tool_id (canonical string, e.g., `list.accounts`)
- risk tier (read, write, admin, destructive)
- owning team/system
- which MCP gateway/namespace exposes it

10.2 Role-to-tool matrix (example)

```
ROLE                        ALLOWED TOOLS
--------------------------  ----------------------------------------
mcp.accounts.reader         list.accounts  accounts.get  accounts.search
mcp.payments.initiator      payments.create  payments.quote  payments.status
mcp.payments.admin          payments.refund  payments.chargeback  payments.cancel
```

10.3 Token exchange governance (OBO)

You need an explicit policy for which clients can exchange and what they can request. Recommended invariant:

```
tools(AT_mcp)  <=  tools_allowed_for_intent(AT_agent, client_roles)
```

10.4 Keycloak as a pragmatic candidate

Keycloak is widely used as an open-source IdP/AS and appears in MCP auth example documentation. A practical setup:

- Define realm roles like `mcp.accounts.reader`.
- Assign roles to confidential clients (service accounts).
- Map roles -> tool claims using protocol mappers:
  - either map to `scope` entries (space-delimited tool ids), or
  - map to `tool_permissions` claim.

Restrict token exchange:

- only allow it for the agent runtime client
- only to approved resources
- only to approved tool scopes

10.5 Keycloak and MCP 2025-11-25: current state (26.4+ / 26.5.x) and coping patterns

Keycloak is a pragmatic open-source baseline for OAuth/OIDC deployments and, starting in the 26.4+ line, it explicitly positions itself as an authorization server for MCP by publishing OAuth 2.0 server metadata at the required well-known location and providing MCP setup guidance.

However, MCP 2025-11-25 raises the bar around *resource-bound* tokens and registration. The current state is best described as:

- **Token Exchange (RFC 8693):** officially supported in Keycloak 26.2+. This gives you a solid OBO foundation for minting downscoped downstream tokens.
- **MCP authorization server metadata (RFC 8414):** supported and documented, enabling standards-based discovery of Keycloak's OAuth endpoints for MCP clients.
- **Resource Indicators (RFC 8707):** still not implemented in Keycloak as of 26.4/26.5.x, which means the clean "client sends `resource=...` and Keycloak maps it into `aud`" flow is

not available out of the box. Keycloak's MCP guide therefore marks MCP 2025-11-25 as partially supported.

- **Client ID Metadata Document (MCP 2025-11-25 SHOULD):** not supported yet; use pre-registration or Dynamic Client Registration (RFC 7591) depending on your deployment constraints.

10.5.1 Coping pattern: keep the MCP narrative RFC 8707-clean, but implement with Keycloak-compatible knobs The safest way to avoid design drift is to keep your *conceptual* model consistent (RFC 8707 `resource` -> token `aud`), even if the IdP cannot process `resource` natively yet.

A practical coping approach with Keycloak is:

1) **Treat each MCP server/gateway as a protected resource with a canonical identifier** (the URL you want to appear in `aud`).
2) **Model resource binding using Keycloak client scopes + Audience mappers**:
   - Create optional client scopes that represent MCP resource categories or bundles (for example: `mcp:tools`, `mcp:resources`, `mcp:prompts`).
   - Attach an Audience mapper to those scopes so the resulting access token contains the MCP server URL in `aud`.
   - Require the client to request at least one of these MCP scopes when asking for a token to that MCP server.
3) **Enforce consistency in the gateway**:
   - Canonicalize the incoming host/path to a resource id.
   - Validate that `aud` contains that id.
   - Do not accept "generic" tokens that happen to have broad audiences.

This gives you deterministic behavior even when the AS does not accept `resource` directly.

10.5.2 Bonus: sender-constrained tokens are becoming realistic Keycloak 26.4 highlights full DPoP support. If your MCP clients and gateways can handle it, sender-constrained access tokens meaningfully reduce replay value for high-impact tools (still not a substitute for tool-level authorization).

The overall recommendation stays the same: use Keycloak for what it is strong at (OIDC, token exchange, policy-managed issuance) and put protocol-aware enforcement at the gateway (PEP), with an optional external PDP for complex enterprise policy.

10.6 Governance: deterministic authorization needs a catalog

Tool-scoped authorization becomes operationally simple only if you treat **resources** and **tools** as first-class governed objects, not ad-hoc strings in code.

A practical governance model that keeps decisions deterministic:

1) **Register MCP servers/gateways as protected resources in the IdP/AS**
   - Pick a canonical resource identifier (usually the base URL of the MCP server or gateway route, e.g., `https://mcp-gw.example.com/mcp`).
   - Treat that identifier as the value that must appear in the access token `aud`.

2) **Maintain a tool catalog (per resource)**
   - Fully-qualified tool identifier (stable string used in `params.name`).
   - JSON Schema for arguments (what the PEP validates before execution).
   - Risk tier (read/write/irreversible) and any step-up requirements.
   - Optional: action model (`invoke`, `list`, `admin`) so the token can be action-scoped, not just tool-scoped.
3) **Model roles/policies as bundles of (resource, tool, action)**
   - Roles are assigned to OAuth clients (backend apps, agent runtimes) and optionally to workloads (SPIFFE identities, K8s SAs, etc.).
   - Token issuance becomes a deterministic function of: (client identity, subject context, requested intent, requested resource, requested tools).
4) **Keep enforcement simple at the edge**
   - Gateways/PEPs do not "decide"; they verify + match:
     - JWT validity + `aud` membership
     - exact tool id match
     - optional action match
     - optional intent/session binding checks

This setup gives you the "single source of truth" property: the IdP/AS owns the permission model, and every PEP enforces the same contract.

A minimal onboarding example (new tool):

- MCP-B team ships a new tool: `payments.refund` (irreversible).
- Security registers `payments.refund` in the tool catalog for resource `https://mcp-b.example.com/mcp`, marks it high risk, and requires HITL.
- A role `mcp.payments.refunder` is created/updated to include `(MCP-B, payments.refund, invoke)`.
- Only the break-glass workflow client is allowed to request that role (and the AS enforces it during token exchange).

Result: the gateway logic stays unchanged; only catalog + AS policy evolves.

11. Gateways as PEPs: why this is usually the best option

11.1 Benefits

- Consistency across languages and teams.
- Better observability and auditing at the choke point.
- Faster rollout of new policies.
- Less developer foot-gun potential.

11.2 Trade-offs (ASCII table)

```
Trade-offs summary (compressed to avoid line overflow):

PEP in gateway (recommended)
  Pros:  Central policy, consistent enforcement, strong audit point
  Cons:  Extra component, must be highly available, policy rollout discipline

PEP in each MCP server
```

```
  Pros:  Full semantic context, no extra hop
  Cons:  Code duplication, drift risk, inconsistent logging/metrics

Scope string (tool names)
  Pros:  Simple, widely compatible with OAuth tooling
  Cons:  Expressiveness limits, scope explosion risk, parsing bugs if not careful

Structured claim (tool_permissions)
  Pros:  Exact match, avoids scope parsing ambiguity, easy to extend per tool
  Cons:  Custom claim, governance required, interop depends on clients

authorization_details (RAR)
  Pros:  Structured and extensible constraints (amount limits, accounts, etc.)
  Cons:  More complex, uneven ecosystem support

JWT local validation
  Pros:  Low latency, no introspection dependency
  Cons:  Revocation lag, key rotation and cache management

Introspection per call
  Pros:  Immediate revocation, dynamic policy
  Cons:  Latency, AS dependency, potential SPOF
```

12. Solo.io Agent Gateway (agentgateway) as a practical implementation

Solo.io publishes extensive agentgateway material that maps well to this paper:

- MCP auth spec compliance with OAuth providers and protected resource metadata exposure (agentgateway can act as a resource server validating JWTs).
- Tool access control using fine-grained RBAC policies; documentation and release notes describe policy-driven authorization, including a Cedar policy engine in agentgateway and CEL-based RBAC examples in tool access guides.
- Protocol-aware routing for stateful JSON-RPC sessions, including per-session authorization considerations.
- MCP multiplexing and tool federation (multiple MCP servers behind a single endpoint, with unified tool listing).

12.1 Why agentgateway is relevant to tool-scoped authorization

A generic reverse proxy can validate a JWT, but it usually cannot authorize "this JSON-RPC message is a tools/call for tool X". agentgateway is MCP-aware and can reason about JSON-RPC bodies, tool names, and session behavior.

It also fits naturally as the **Agent Gateway (PEP #1)** in front of an agent API: the same proxy can enforce JWT validity and RFC 8707 audience pinning for agent endpoints, while still performing tool-level authorization for MCP routes (Section 5.1, Option B).

The mismatch scenario becomes enforceable without modifying MCP server code:

- token permits `list.accounts`
- payload requests `payments.transfer`

- gateway denies based on tool name and claims

12.2 MCP auth and dynamic client registration

MCP authorization guidance strongly recommends metadata discovery and dynamic client registration because clients do not know the set of MCP servers in advance. agentgateway documentation provides a complete walkthrough that uses Keycloak as an IdP, where the MCP Inspector dynamically registers as a client, performs OAuth, obtains a JWT, and then uses that JWT to access MCP tools via the gateway.

Simplified view:

```
(1) Client discovers metadata  ->
(2) Client registers           ->
(3) OAuth code flow            ->
(4) JWT to MCP
```

12.2.1 agentgateway config patterns that map directly to this paper

agentgateway exposes two MCP-specific policies that are almost tailor-made for "PEP outside the app code":

1) `mcpAuthentication`: makes agentgateway behave like an OAuth-protected resource in front of one or more MCP servers. It validates JWTs and can expose protected resource metadata.
2) `mcpAuthorization`: evaluates CEL rules in the context of MCP method invocations and tool names. Critically, if a tool is not allowed, agentgateway automatically filters it from list responses.

A minimal (illustrative) standalone config looks like:

```yaml
# Authentication: treat agentgateway as the resource server in front of MCP.
mcpAuthentication:
  issuer: https://idp.example.com/realms/mcp
  jwksUrl: https://idp.example.com/realms/mcp/protocol/openid-connect/certs
  provider:
    keycloak: {}
  resourceMetadata:
    resource: https://mcp-gw.example.com/mcp
    scopesSupported:
      - list.accounts
      - payments.transfer
    bearerMethodsSupported:
      - header
```

Note: agentgateway examples show additional bearer methods, but MCP authorization guidance explicitly forbids sending access tokens via URL query parameters. For MCP deployments, prefer the `Authorization: Bearer` header.

Authorization rules (CEL). The docs show rules keyed off `mcp.tool.name` and JWT claims:

```yaml
mcpAuthorization:
  rules:
```

```
    # Anyone with a valid token can call list.accounts
    - 'mcp.tool.name == "list.accounts"'
    # Only a specific caller can call payments.transfer (example pattern)
    - 'jwt.sub == "payments-service" && mcp.tool.name == "payments.transfer"'
```

This is not yet "dynamic scope membership", but it demonstrates the two core mechanics we need:

- Gate on tool name (`mcp.tool.name`).
- Gate on claims (`jwt.*`).
- Let the gateway do both enforcement and list filtering.

For scalable deployments, you would replace the second rule with a membership check against a claim that encodes the allowed tool set (for example a structured `tool_permissions` claim, or a tokenized `scope` claim that your gateway parses safely).

12.2.2 Delegating authorization to an external helper (ext_authz) for complex PEP logic

For many organizations, the most realistic way to keep developers out of the authorization business is:

- keep **PEP** at the gateway (close to the request path, protocol-aware, fast fail)
- delegate **PDP** to a separate service that you own and can version independently

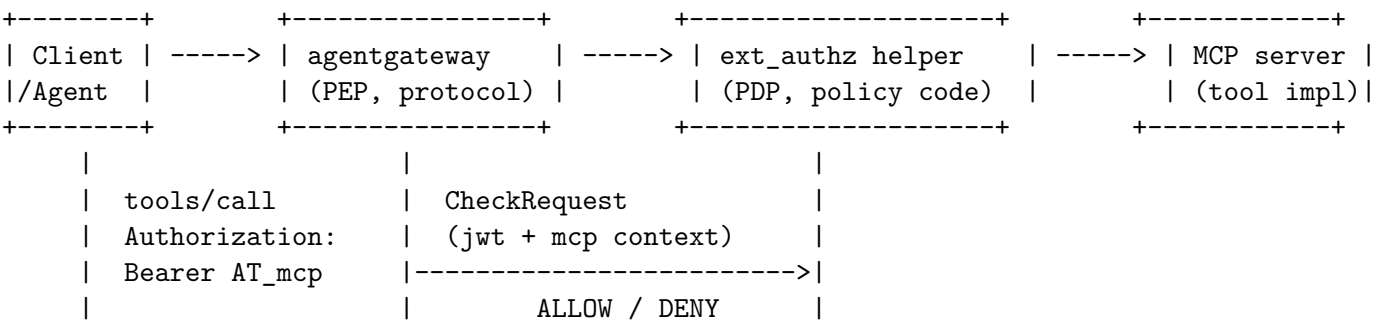
Solo.io AgentGateway supports this pattern via **external authorization (ext_authz)**. Conceptually, this is the Envoy external authorization filter: the gateway sends an authorization CheckRequest to an external gRPC/HTTP service; the service returns ALLOW/DENY; the gateway enforces (returning 403 or forwarding upstream).

From a product and governance perspective, this is an attractive separation of concerns: security teams can own and version the external policy service, while platform teams keep the gateway configuration thin. For MCP specifically, a first-party deterministic MCP authz module (built-in, but still policy-driven) would reduce deployment friction even further by standardizing resource canonicalization, tool matching, and error mapping.

This is useful when CEL or built-in RBAC is not enough, for example:

- strict canonicalization rules for RFC 8707 resources (`aud` vs route resource id)
- tool-to-claim matching that requires parsing JSON-RPC payloads and mapping tools across federated targets
- enterprise policies that live in a central engine (OPA, Cedar-as-a-service, custom policy code)
- decision caching, risk scoring, or step-up auth requirements for "dangerous" tools

ASCII flow (gateway as PEP, helper as PDP):



```
|                    |<-------------------------|
|          403 or forward to MCP upstream       |
```

A minimal ext_authz attachment (illustrative) from agentgateway docs:

```yaml
extAuthz:
  host: localhost:9000
  protocol:
    grpc:
      # Optional: send selected metadata to the helper
      # (CEL expression; "jwt" refers to parsed claims)
      metadata:
        dev.agentgateway.jwt: '{"claims": jwt}'
```

The helper itself can be a small service that implements a deterministic check:

1) Parse and validate JWT locally (signature, iss, exp, aud contains this gateway's resource id).
2) Parse MCP JSON-RPC:
   - verify `method == "tools/call"`
   - extract `params.name` as the tool id.
3) Authorize:
   - exact match: tool id must exist in scope (tokenized) or in a structured claim
   - optional: verify tool is valid for the selected upstream/target resource when multiplexing.
4) Return:
   - ALLOW (2xx) and optionally attach headers/metadata for upstream logging
   - DENY (403) with an error body aligned with your threat model (avoid leaking tool names if needed).

This makes the paper's "claims <-> payload" enforcement implementable without embedding authorization logic inside every agent and tool server.

12.2.2.1 Concrete example: what the helper sees and how it denies a tool mismatch

A simplified (pseudo) CheckRequest the helper receives (conceptual, not full proto):

```
CheckRequest
  attributes:
    request:
      http:
        method: "POST"
        path: "/mcp"
        headers:
          authorization: "Bearer <AT_mcp_multi>"
          content-type: "application/json"
        body: "{...jsonrpc payload...}"
    metadata_context:
      filter_metadata:
        dev.agentgateway.jwt:
          claims:
            iss: "https://as.example.com"
            sub: "client_backend_app"
            aud: ["https://mcp-a.example.com/mcp","https://mcp-b.example.com/mcp"]
```

```
scope: "list.accounts payments.transfer"
tool_permissions:
  - rs: "https://mcp-a.example.com/mcp"
    tool: "list.accounts"
  - rs: "https://mcp-b.example.com/mcp"
    tool: "payments.transfer"
```

The helper extracts:

- `resource_id` from the gateway route / upstream selection (the "local" resource)
- `tool_id` from MCP JSON-RPC (`method == tools/call`, `params.name`)

Example MCP request that should be denied at MCP-A (wrong tool for that resource):

```
{
  "jsonrpc":"2.0",
  "id":4401,
  "method":"tools/call",
  "params":{
    "name":"payments.transfer",
    "arguments":{"amount":"10.00","currency":"EUR"}
  }
}
```

Deterministic helper decision:

```
resource_id = https://mcp-a.example.com/mcp
aud contains resource_id?   YES
tool_id = payments.transfer
is tool permitted for rs=A? NO (only list.accounts is permitted)
=> DENY (403)
```

A minimal deny response pattern (HTTP ext_authz style) is simply a non-2xx status code. For gRPC ext_authz, the helper returns a DENIED response and can optionally include a body for debugging.

Key point: this denial happens before the request reaches the MCP server, so you get consistent enforcement even if tool servers are implemented by different teams.

12.2.3 External auth helper as a governance pressure valve

External authorization is not only a technical mechanism; it is an operational control:

- you can ship policy changes without redeploying agents or MCP servers
- you can centralize audit decisions (why denied, which claim missing, which resource mismatched)
- you can add guardrails like rate limits or anomaly detection for specific tools
- you can run a single PDP implementation across HTTP, MCP, and other protocols if the gateway can provide enough context ### 12.3 Tool access control and discovery reduction

agentgateway tool access guides describe applying RBAC rules so that access to MCP tools is authorized based on JWT claims and tool context. Practically, you want:

- DENY unauthorized `tools/call`

- ALLOW but FILTER `tools/list` so only permitted tools are shown

This aligns with least privilege: agents only see the tools they are authorized to use, reducing confusion and limiting attack surface.

12.4 MCP multiplexing: one endpoint, many MCP servers

agentgateway supports federating multiple MCP servers (targets) behind a single MCP connection. The gateway can aggregate tools from multiple servers and present them as one unified tool server, typically prefixing tool names with target identifiers.

ASCII multiplex sketch:

```
Client connects to one endpoint: http(s)://agentgateway.example.com/mcp

+----------------------+      +---------------------+
| agentgateway         |----->| MCP server A (time) |
| (single connection)  |----->| MCP server B (fetch)|
|                      |----->| MCP server C (bank) |
+----------------------+      +---------------------+

Unified tool view (prefixed):
  time_get_current_time
  fetch_fetch
  bank_list_accounts
  bank_payments_transfer
```

This matters for authorization because a single PEP can enforce consistent policy across multiple upstream tool servers.

12.5 Putting it together: agentgateway as PEP #2

At minimum, configure:

- JWT validation (issuer, JWKS)
- audience restriction (resource-centric `aud`)
- tool authorization policy

Policy intent:

```
ALLOW if:
  aud contains "https://mcp-gw.example.com/mcp"
  AND method == "tools/call"
  AND requested_tool in jwt.tool_permissions[].tool
Else DENY
```

If your policy engine is expression-based and you store permissions in scope, implement exact tokenization, not substring checks.

12.6 Strong points summary

```
Agent Gateway (agentgateway) strengths as a PEP for MCP:
```

```
- Spec-aware MCP auth: implements MCP authentication/authorization patterns
- JWT validation at the edge: local verification against issuer/JWKS
- Authorization Server Proxy mode: proxy metadata/registration + IdP quirks
- Tool-level authorization: CEL rules using mcp.tool.name + jwt claims
- Automatic tools/list filtering: hides tools the caller cannot invoke
- Protocol-aware routing: can inspect JSON-RPC bodies to enforce policies
- Multiplexing/federation: one endpoint for multiple MCP servers; tools prefixed per target
- Session-aware policy: can vary discovery and access per session/client
```

13. Hands-on reproduction: MCP Inspector as a test harness

The MCP Inspector is commonly used to connect to MCP servers through gateways. A practical test recipe:

1. Connect through your MCP gateway using streamable HTTP.
2. Provide a bearer token that only permits `list.accounts`.
3. Verify:
   - `tools/list` returns only `list.accounts` (if filtering is enabled).
   - `tools/call` for `list.accounts` succeeds.
   - `tools/call` for `payments.transfer` is denied with a tool-level authorization error.

This gives you a regression harness for policy changes and token issuance logic.

14. Conformance suite (selected vectors)

Assumptions:

- `aud` membership match (string or array; resource must be present)
- exact tool name match
- wildcards disabled
- canonical tool naming (lower-case recommended)

```
Test vectors (compact, width-safe):

T01  ALLOW
  aud ok:       yes
  token tools:  {list.accounts}
  requested:    list.accounts
  reason:       tool in token

T02  ALLOW (with filtering)
  aud ok:       yes
  token tools:  {list.accounts}
  requested:    tools/list
  reason:       list is allowed, but response is filtered to permitted tools

T03  DENY
  aud ok:       yes
  token tools:  {list.accounts}
  requested:    payments.transfer
  reason:       mismatch (tool not permitted)
```

```
T04  DENY
  aud ok:       yes
  token tools:  {list.accounts}
  requested:    payments.payment
  reason:       mismatch (tool not permitted)

T05  DENY
  aud ok:       yes
  token tools:  {list.accounts}
  requested:    payments.transfer
  reason:       prompt-induced abuse blocked

T06  DENY
  aud ok:       NO
  token tools:  {list.accounts}
  requested:    list.accounts
  reason:       audience mismatch

T07  DENY (if strict tool IDs)
  aud ok:       yes
  token tools:  {list.accounts}
  requested:    LIST.ACCOUNTS
  reason:       case mismatch

T08  DENY
  aud ok:       yes
  token tools:  {list.accounts}
  requested:    list.accounts.v2
  reason:       version mismatch

T09  ALLOW
  aud ok:       yes
  token tools:  {list.accounts, accounts.get}
  requested:    accounts.get
  reason:       exact match

T10  DENY
  aud ok:       yes
  token tools:  {accounts.get}
  requested:    accounts.delete
  reason:       scope missing

T11  DENY
  aud ok:       yes
  token tools:  {list.accounts}
  requested:    <missing name>
  reason:       malformed MCP request
```

```
T12  DENY
  aud ok:       yes
  token tools:  {list.accounts}
  requested:    tools/call without JWT
  reason:       authentication failure

T13  ALLOW (multi-aud)
  aud ok:       yes (aud contains target resource)
  token tools:  {(A,list.accounts),(B,payments.transfer)}
  requested:    list.accounts @ MCP-A
  reason:       permitted pair for resource

T14  DENY (multi-aud)
  aud ok:       yes (aud contains B)
  token tools:  {(A,list.accounts)}
  requested:    payments.transfer @ MCP-B
  reason:       tool not permitted for that resource

T15  DENY (multi-aud)
  aud ok:       NO (aud does not contain C)
  token tools:  {(A,list.accounts),(B,payments.transfer)}
  requested:    list.accounts @ MCP-C
  reason:       audience mismatch

T16  DENY (multi-aud)
  aud ok:       yes (aud contains B)
  token tools:  {(B,payments.transfer)}
  requested:    payments.payment @ MCP-B
  reason:       mismatch (tool not permitted)

T17  ALLOW (multi-aud, alias audiences for one gateway)
  aud ok:       yes (aud contains an alias of the same gateway)
  token tools:  {(GW,list.accounts)}
  requested:    list.accounts @ MCP-GW (alias host)
  reason:       alias mapped to canonical resource id before membership check

T18  ALLOW (multi-aud, trailing slash normalized)
  aud ok:       yes (aud contains /mcp/ but canonical is /mcp)
  token tools:  {(A,list.accounts)}
  requested:    list.accounts @ MCP-A
  reason:       canonicalization removes trailing slash before aud check

T19  DENY (multi-aud, wrong resource for a permitted tool)
  aud ok:       yes (aud contains A)
  token tools:  {(B,payments.transfer)}
  requested:    payments.transfer @ MCP-A
  reason:       tool permitted only for resource B
```

```
T20  DENY (multi-aud, flat tool list rejected)
  aud ok:       yes (aud contains A and B)
  token tools:  {list.accounts,payments.transfer} (no rs binding)
  requested:    payments.transfer @ MCP-A
  reason:       invalid scope contract for multi-aud tokens

T21  DENY (multi-aud, non-canonical rs value)
  aud ok:       yes
  token tools:  {(rs not canonical -> mismatch)}
  requested:    list.accounts @ MCP-A
  reason:       rs mismatch (require canonical rs in token)

T22  ALLOW (multi-aud, three resources)
  aud ok:       yes (aud contains C)
  token tools:  {(A,list.accounts),(B,payments.transfer),(C,fx.quote)}
  requested:    fx.quote @ MCP-C
  reason:       permitted pair for resource C

T23  DENY (multi-aud, structured claim beats scope string)
  aud ok:       yes
  token tools:  tool_permissions={(A,list.accounts)} but scope has transfer
  requested:    payments.transfer @ MCP-A
  reason:       tool_permissions is source of truth

T24  DENY (multi-aud, missing invoke action)
  aud ok:       yes
  token tools:  {(A,list.accounts)} but actions do not include invoke
  requested:    list.accounts @ MCP-A
  reason:       action not authorized

T25  ALLOW (multi-aud, tools/list filtered per resource)
  aud ok:       yes
  token tools:  {(A,list.accounts),(B,payments.transfer)}
  requested:    tools/list @ MCP-B
  reason:       allow+filter => only payments.transfer returned

T26  DENY (multi-aud, mixed-case tool name)
  aud ok:       yes
  token tools:  {(A,list.accounts)}
  requested:    LIST.ACCOUNTS @ MCP-A
  reason:       strict tool ids (case sensitive)
```

`ALLOW*` means: allow the call but filter the response to the permitted tool set.

15. Security considerations and hardening

Tool-scoped authorization is necessary but not sufficient. In agentic systems, the policy model must assume that inputs can influence tool selection and that planners can fail. The goal is to make the system safe under those failure modes.

15.1 Token handling and cryptographic hygiene

- Prefer short-lived access tokens for tool execution (minutes, not hours). Cache only within a bounded transaction window.
- For high-impact tools, consider sender-constrained tokens (mTLS or DPoP) to reduce replay value.
- Follow JWT Best Current Practices (RFC 8725): enforce `iss`, `aud`, `exp`, validate `alg`, pin accepted algorithms, and handle `kid` rotation safely.
- Use `jti` with replay detection where appropriate, especially for write tools.

15.2 Prevent token passthrough and preserve delegation semantics

A recurring anti-pattern is token passthrough: forwarding the caller token to downstream resources. Instead:

- The agent should obtain a downstream token explicitly minted for the MCP resource using Token Exchange (RFC 8693) or a first-party client flow.
- Preserve delegation context via an `act` claim (subject vs actor) or equivalent metadata so audits can distinguish "who caused" vs "who executed".

15.3 Tool naming: canonicalization and confusable defense

Tool IDs are an authorization surface. Treat tool names as identifiers, not free text:

- Enforce a strict ASCII charset and length bounds.
- Use a canonical form (for example lowercase) and reject non-canonical variants.
- Normalize Unicode where it can appear (NFKC) and run confusable checks if you accept non-ASCII anywhere.

15.4 Reduce discovery and blast radius

- Prefer ALLOW+FILTER for `tools/list`: only return the tools the token permits.
- Consider tool risk tiers and require step-up controls for destructive tools (break-glass roles, approvals, or human confirmation).
- Implement rate limiting and anomaly detection per tool, not just per client.

15.5 Mapping to OWASP Agentic Top 10 (2026)

The OWASP Top 10 for Agentic Applications (ASI) provides a useful taxonomy to sanity-check MCP deployments. The token-centric approach in this paper directly mitigates several ASI entries by constraining the blast radius of tool execution:

- ASI01 Agent Goal Hijack: strict tool allow-lists prevent goal manipulation from turning into arbitrary actions.
- ASI02 Tool Misuse and Exploitation: tool-level authorization blocks unsafe tool selection even when planning fails.

Role	Tool IDs
pricing_reader	inventory.get, quote.read
order_operator	orders.create, orders.cancel, inventory.get
refund_operator	payments.refund, quote.read
tenant_supply_acme	acme.inventory.get
tenant_supply_globex	globex.inventory.get
billing_exporter_v2	billing.export.v2
break_glass_ops	payments.refund, orders.cancel, billing.export.v2 (short TTL, approval)

Table 16: Sample role to tool mapping.

- ASI03 Identity and Privilege Abuse: roles in the IdP map to bounded (resource, tool) permissions; OBO downscoping prevents privilege escalation.
- ASI06 Memory and Context Poisoning: short-lived, intent-scoped tokens limit the duration of poisoned contexts.
- ASI08 Cascading Failures: monotonic downscoping and per-hop audience binding reduce chain amplification.

Treat this mapping as a checklist, not a guarantee. Authorization is the "permission layer"; you still need input validation, output handling, and operational kill switches.

15.6 Practical MCP server hardening reminders

Independent of token design, MCP servers and gateways should:

- Require authentication for any remote server access.
- Validate token signature, audience, and expiry on every tool call.
- Centralize policy enforcement in a gateway or runtime owned by security (avoid developer-by-developer implementations).
- Use safe error handling: do not leak tokens, stack traces, or tool internals in error responses.

Appendix C: Test Vectors (24 cases)

The vectors below are intended to be executable against a tool-scoped MCP gateway + policy engine (PEP/PDP split). They focus on canonicalization, audience binding, replay properties, tenant namespace enforcement, and exchange monotonicity.

Case ID	Token aud	Token perms/scope summary	MCP method + name	Canonical(name)	Expected	Deny reason
TV-01	mcp-gw	tp: inventory.get, quote.read	call_tool inventory.get	inventory.get	Allow	-
TV-02	mcp-gw	tp: inventory.get, quote.read	call_tool payments.refund	payments.refund	Deny	insufficient_tool_scope
TV-03	agent	tp: inventory.get	call_tool inventory.get	inventory.get	Deny	invalid_audience
TV-04	mcp-gw	tp: inventory.get	call_tool Inventory.Get	inventory.get	Deny	non_canonical_tool_name
TV-05	mcp-gw	tp: inventory.get	call_tool inventry.get	reject	Deny	invalid_tool_name_charset
TV-06	mcp-gw	tp: inventory.get; exp past	call_tool inventory.get	inventory.get	Deny	token_expired
TV-07	mcp-gw	tp: inventory.get; nbf future	call_tool inventory.get	inventory.get	Deny	token_not_yet_valid
TV-08	mcp-gw	issuer untrusted	call_tool inventory.get	inventory.get	Deny	invalid_issuer
TV-09	mcp-gw	signature invalid	call_tool inventory.get	inventory.get	Deny	invalid_token_signature
TV-10	mcp-gw	no tp; scope includes inventory.get	call_tool inventory.get	inventory.get	Allow	-
TV-11	mcp-gw	tp: inventory.get; no scope	call_tool inventory.get	inventory.get	Allow	-
TV-12	mcp-gw	tp+scope quote.read only	call_tool inventory.get	inventory.get	Deny	insufficient_tool_scope
TV-13	mcp-gw	tenant acme; tp acme.inventory.get	call_tool acme.inventory.get	acme.inventory.get	Allow	-
TV-14	mcp-gw	tenant acme; tp acme.inventory.get	call_tool globex.inventory.get	globex.inventory.get	Deny	tenant_mismatch
TV-15	mcp-gw	tp inventory.get	call_tool inventory.get(space)	inventory.get	Deny	non_canonical_tool_name
TV-16	mcp-gw	tp inventory.get	call_tool inventory/get	reject	Deny	invalid_tool_name_charset
TV-17	mcp-gw	tp billing.legacy_export	call_tool billing.legacy_export	billing.legacy_expor	Deny	tool_deprecated
TV-18	mcp-gw	tp valid; policy_version too old	call_tool inventory.get	inventory.get	Deny	policy_version_mismatch
TV-19	exchange	subject tp inventory.get; request adds refund	token_exchange	n/a	Deny	downscope_violation
TV-20	exchange	subject tp inventory.get; request subset	token_exchange	n/a	Allow	-
TV-21	mcp-gw	tp quote.read; short TTL	call_tool quote.read	quote.read	Allow	-
TV-22	mcp-gw	tp quote.read; long TTL over policy	call_tool quote.read	quote.read	Deny	ttl_exceeds_policy
TV-23	exchange	subject tp inventory.get; request resources mcp-a,mcp-b; request subset	token_exchange	n/a	Allow	-
TV-24	mcp-a,b	aud[]: mcp-a,mcp-b; tp: inventory.get	call_tool inventory.get	inventory.get	Allow	-

Table 17: Test vectors (24 cases).

Appendix D: End-to-end flows (ALLOW/DENY)

This appendix provides end-to-end request traces with explicit hop boundaries. The intent is operational: you should be able to paste these into a test harness and observe the same allow/deny decisions at the AS (PDP) and at the MCP gateway (PEP).

D.0 DENY at Agent Gateway: audience mismatch (token confusion)

This failure mode is common in multi-hop systems: a caller obtains a valid token, but for the wrong resource.

Step 1: backend client accidentally requests a token for the MCP gateway audience, but then uses it to call the agent.

Client -> AS: client_credentials (WRONG audience)

```
POST /oauth2/token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials&
client_id=backend_app&
client_secret=...&
scope=agent.invoke inventory.get quote.read&
resource=https://mcp-gw.example.com
```

AS -> Client: AT_wrong (aud=mcp-gw)

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

{
  "access_token":"AT_wrong",
  "token_type":"Bearer",
  "expires_in":300,
  "scope":"agent.invoke inventory.get quote.read"
}
```

Step 2: client calls the agent gateway with AT_wrong. The Agent Gateway validates the signature, but denies on audience mismatch.

Client -> Agent GW: invoke (wrong token audience)

```
POST /v1/agent/invoke HTTP/1.1
Host: agent-gw.example.com
Authorization: Bearer AT_wrong
Content-Type: application/json

{
  "intent_id":"ord-2026-000124",
  "input":"Check inventory for SKU X-42 and produce a quote."
}
```

Agent GW -> Client: invalid_audience

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer error="invalid_token", error_description="audience
    mismatch", resource="https://agent-gw.example.com"
Content-Type: application/json

{
  "error":"invalid_token",
  "reason":"invalid_audience",
  "expected_aud":"https://agent-gw.example.com",
  "received_aud":["https://mcp-gw.example.com"]
}
```

D.1 ALLOW: client to agent to token exchange to MCP tool

Step 1: backend client obtains an agent token (AT_agent).

Client -> AS: client_credentials (agent audience)

```
POST /oauth2/token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials&
client_id=backend_app&
client_secret=...&
scope=agent.invoke inventory.get quote.read&
resource=https://agent-gw.example.com
```

AS -> Client: AT_agent

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

{
  "access_token":"AT_agent",
  "token_type":"Bearer",
  "expires_in":300,
  "scope":"agent.invoke inventory.get quote.read"
}
```

Step 2: client calls the **agent gateway (PEP #1)**, which validates `AT_agent` and forwards to the agent runtime.

Client -> Agent GW: invoke

```
POST /v1/agent/invoke HTTP/1.1
Host: agent-gw.example.com
Authorization: Bearer AT_agent
Content-Type: application/json

{
```

```
  "intent_id":"ord-2026-000123",
  "input":"Check inventory for SKU X-42 and produce a quote."
}
```

Step 3: agent performs OBO token exchange to obtain an MCP token (AT_mcp) scoped to the gateway and tools.

Agent -> AS: token_exchange (OBO downscope)

```
POST /oauth2/token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=urn:ietf:params:oauth:grant-type:token-exchange&
subject_token=AT_agent&
subject_token_type=urn:ietf:params:oauth:token-type:access_token&
resource=https://mcp-gw.example.com&
scope=mcp.call_tool inventory.get quote.read&
intent_id=ord-2026-000123
```

AS -> Agent: AT_mcp (downscoped)

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

{
  "access_token":"AT_mcp",
  "token_type":"Bearer",
  "expires_in":60,
  "scope":"mcp.call_tool inventory.get quote.read",
  "issued_token_type":"urn:ietf:params:oauth:token-type:access_token"
}
```

Step 4: agent calls MCP gateway, which enforces tool-level authorization.

Agent -> MCP gateway: tools/call

```
POST /mcp HTTP/1.1
Host: mcp-gw.example.com
Authorization: Bearer AT_mcp
Content-Type: application/json

{
  "jsonrpc":"2.0",
  "id":"d1-1",
  "method":"tools/call",
  "params":{
    "name":"inventory.get",
    "arguments":{"sku":"X-42"}
  }
}
```

MCP gateway -> Agent: allowed

```
{
  "jsonrpc":"2.0",
  "id":"d1-1",
  "result":{
    "sku":"X-42",
    "available":17,
    "warehouse":"MAD-01"
  }
}
```

D.2 DENY at AS: scope escalation in token exchange

Agent requests more permissions than exist in the subject token and policy.

Agent -> AS: token_exchange (attempted scope escalation)

```
POST /oauth2/token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=urn:ietf:params:oauth:grant-type:token-exchange&
subject_token=AT_agent&
resource=https://mcp-gw.example.com&
scope=mcp.call_tool inventory.get payments.refund
```

AS -> Agent: invalid_scope

```
HTTP/1.1 400 Bad Request
Content-Type: application/json

{
  "error":"invalid_scope",
  "error_description":"requested permissions exceed subject token and policy",
  "reason":"downscope_violation",
  "policy_version":"2026-02-17.1"
}
```

D.3 DENY at MCP gateway: tenant namespace mismatch

Token is valid for tenant `acme`, but the request targets a `globex` namespaced tool.

Agent -> MCP gateway: cross-tenant tools/call

```
POST /mcp HTTP/1.1
Host: mcp-gw.example.com
Authorization: Bearer AT_mcp
Content-Type: application/json

{
  "jsonrpc":"2.0",
  "id":"d3-1",
```

```
  "method":"tools/call",
  "params":{
    "name":"globex.inventory.get",
    "arguments":{"sku":"X-42"}
  }
}
```

MCP gateway -> Agent: tenant_mismatch

```
{
  "error":"access_denied",
  "reason":"tenant_mismatch",
  "token_tenant":"acme",
  "requested_tool":"globex.inventory.get"
}
```

D.4 DENY at MCP gateway: non-canonical tool name (confusable / variant)

Request uses a non-canonical tool spelling. Canonicalization rejects before authorization.

Agent -> MCP gateway: tools/call (non-canonical)

```
POST /mcp HTTP/1.1
Host: mcp-gw.example.com
Authorization: Bearer AT_mcp
Content-Type: application/json

{
  "jsonrpc":"2.0",
  "id":"d4-1",
  "method":"tools/call",
  "params":{
    "name":"Inventory.Get",
    "arguments":{"sku":"X-42"}
  }
}
```

MCP gateway -> Agent: non_canonical_tool_name

```
{
  "error":"access_denied",
  "reason":"non_canonical_tool_name",
  "canonical_name":"inventory.get",
  "requested_name":"Inventory.Get"
}
```

Appendix E: Full transaction traces (client -> IdP -> agent -> MCP gateway -> MCP server)

This appendix adds concrete traces that align with the "IdP as PDP" framing. They are intentionally verbose so they can be used as regression tests.

E.1 ALLOW: multi-resource transaction with `aud[]` and resource-qualified tool binding

Step 0: backend client calls the agent with an upstream token `AT_agent` (not shown; identical to Appendix D).

Step 1: agent exchanges `AT_agent` for a short-lived multi-resource MCP token, requesting two resources.

Agent -> AS: token_exchange requesting two MCP resources

```
POST /oauth2/token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=urn:ietf:params:oauth:grant-type:token-exchange&
subject_token=AT_agent&
subject_token_type=urn:ietf:params:oauth:token-type:access_token&
resource=https://mcp-a.example.com/mcp&
resource=https://mcp-b.example.com/mcp&
scope=list.accounts payments.transfer&
intent_id=txn-2026-02-19-0001
```

AS -> Agent: AT_mcp_multi (JWT payload sketch)

```
{
  "aud":[
    "https://mcp-a.example.com/mcp",
    "https://mcp-b.example.com/mcp"
  ],
  "scope":"list.accounts payments.transfer",
  "mcp_toolset":[
    {"rs":"https://mcp-a.example.com/mcp","tools":["list.accounts"]},
    {"rs":"https://mcp-b.example.com/mcp","tools":["payments.transfer"]}
  ],
  "exp":1760669100,
  "act":{"sub":"agent_runtime"}
}
```

Step 2: agent calls MCP-A with `list.accounts` and is allowed.

Agent -> MCP-A: tools/call (allowed)

```
POST /mcp HTTP/1.1
Host: mcp-a.example.com
Authorization: Bearer AT_mcp_multi
Content-Type: application/json

{
  "jsonrpc":"2.0",
  "id":"e1-1",
  "method":"tools/call",
  "params":{"name":"list.accounts","arguments":{"limit":10}}
}
```

Step 3: agent calls MCP-B with `payments.transfer` and is allowed.

Agent -> MCP-B: tools/call (allowed)

```
POST /mcp HTTP/1.1
Host: mcp-b.example.com
Authorization: Bearer AT_mcp_multi
Content-Type: application/json

{
  "jsonrpc":"2.0",
  "id":"e1-2",
  "method":"tools/call",

    "params":{"name":"payments.transfer","arguments":{"amount":"10.00","currency":"EUR"}}
}
```

E.2 DENY: same token, wrong tool for the selected resource

The agent mistakenly sends `payments.transfer` to MCP-A. Audience membership succeeds (because MCP-A is in `aud[]`), but the tool binding check fails:

MCP-A PEP: denied (tool not permitted for MCP-A)

```
{
  "error":"access_denied",
  "reason":"insufficient_tool_scope",
  "aud":"https://mcp-a.example.com/mcp",
  "requested_tool":"payments.transfer",
  "permitted_tools":["list.accounts"]
}
```

E.3 DENY: audience mismatch against a canonicalized resource identifier

If the token `aud` contains `https://mcp-gw.example.com/mcp` but the request is sent to `https://mcp-gw.example.com/mcp/` (trailing slash), the PEP must canonicalize the request resource ID before performing the membership check. If canonicalization is not applied consistently, you will see false denies or, worse, false allows through aliasing.

Recommended: define one canonical form for each MCP protected resource and enforce it in both the IdP and the gateway.

References

Additional security guidance:

- OWASP Top 10 for Agentic Applications (2026): https://genai.owasp.org/resource/owasp-top-10-for-agentic-applications-for-2026/
- A Practical Guide for Secure MCP Server Development (v1.0): (local copy referenced in this repository)

Standards and specifications:

- Model Context Protocol (MCP) Specification (2025-11-25): https://modelcontextprotocol.io/specification/20 11-25
- RFC 6749: The OAuth 2.0 Authorization Framework: https://www.rfc-editor.org/rfc/rfc6749
- RFC 6750: The OAuth 2.0 Authorization Framework: Bearer Token Usage: https://www.rfc-editor.org/rfc/rfc6750
- RFC 7519: JSON Web Token (JWT): https://www.rfc-editor.org/rfc/rfc7519
- RFC 8707: Resource Indicators for OAuth 2.0: https://www.rfc-editor.org/rfc/rfc8707
- RFC 8693: OAuth 2.0 Token Exchange: https://www.rfc-editor.org/rfc/rfc8693
- RFC 9068: JWT Profile for OAuth 2.0 Access Tokens: https://www.rfc-editor.org/rfc/rfc9068
- RFC 9396: OAuth 2.0 Rich Authorization Requests: https://www.rfc-editor.org/rfc/rfc9396
- RFC 9728: OAuth 2.0 Protected Resource Metadata: https://www.rfc-editor.org/rfc/rfc9728
- RFC 8414: OAuth 2.0 Authorization Server Metadata: https://www.rfc-editor.org/rfc/rfc8414
- RFC 8725: JSON Web Token Best Current Practices: https://www.rfc-editor.org/rfc/rfc8725

Gateway patterns (agentgateway / Solo / Envoy external auth):

- agentgateway overview and docs: https://agentgateway.dev/docs/kubernetes/latest/
- agentgateway: MCP connectivity: https://agentgateway.dev/docs/mcp/connect/
- agentgateway: MCP authentication: https://agentgateway.dev/docs/mcp/mcp-authn/
- agentgateway: MCP authorization (mcpAuthorization): https://agentgateway.dev/docs/mcp/mcp-authz/
- agentgateway: External authorization (extAuthz): https://agentgateway.dev/docs/configuration/security/ex authz/
- Envoy: External authorization filter (ext_authz): https://www.envoyproxy.io/docs/envoy/latest/configurati
- Solo.io docs: Secure access to MCP servers: https://docs.solo.io/agentgateway/2.1.x/mcp/mcp-access/
- Solo.io docs: Control access to tools: https://docs.solo.io/agentgateway/2.1.x/mcp/tool-access/
- Solo.io docs: BYO external auth service (Envoy ext_authz proto): https://docs.solo.io/gateway/2.0.x/securit ext-auth-service/
- Solo.io MCP Academy lab (multiplex + auth policy): https://www.solo.io/mcp-academy/multiplex-mcp-servers-auth-kgateway-agentgateway
- Solo.io blog: Overhaul of Agent Gateway supporting A2A, MCP, and Kubernetes Gateway API: https://www.solo.io/blog/updated-a2a-and-mcp-gateway
- Solo.io blog: Why do we need a new gateway for AI agents: https://www.solo.io/blog/why-do-we-need-a-new-gateway-for-ai-agents

Keycloak (open-source IdP/AS candidate and Token Exchange support):

- Keycloak guide: Integrating with Model Context Protocol (MCP): https://www.keycloak.org/guides/securing apps/mcp-authz-server
- Keycloak blog: Standard Token Exchange is officially supported in Keycloak 26.2 (RFC 8693): https://www.keycloak.org/2025/05/standard-token-exchange-kc-26-2
- Keycloak blog: Keycloak 26.4.0 released (DPoP, MCP AS metadata; RFC 8707 not implemented yet): https://www.keycloak.org/2025/09/keycloak-2640-released
- Keycloak release page (26.5.3): https://github.com/keycloak/keycloak/releases/tag/26.5.3