SECUREAUTH
Welcome to Better Identity.

HOT TOPIC

AGENTIC AI
WIRING MCP TO YOUR IdP

Integrating AI systems into enterprise infrastructure starts with trust. This piece explains how to securely connect your MCP server to your Identity Provider using OAuth 2.1 and OpenID Connect—enabling agentic AI to authenticate, authorize, and operate within your existing identity fabric.

ARCHITECTING IDENTITY FOR AGENTIC AI: PART 2

This is **Part 2** in the series **Architecting Identity for Agentic AI**, a technical guide for software architects securing AI systems with OAuth, OIDC, and trust frameworks. Designed for architects building scalable, policy-driven identity across enterprise and cross-boundary environments.

BY LUKASZ RADOSZ
SVP of Engineering, SecureAuth

With over two decades in identity and access management, Lukasz brings deep expertise in authentication, authorization, and API security to the SecureAuth team. A champion of open standards like OAuth and OIDC, he's an advocate for modern identity architectures across machine identity, open banking, and transactional access control.

Introduction

In Part 1, we covered the fundamentals of OAuth 2.1 and OpenID Connect (OIDC) as they apply to AI agents. Now, in Part 2, we focus on integrating an **enterprise-hosted MCP server** with your organization's **Identity Provider (IdP)** for secure authentication and authorization. The Model Context Protocol (MCP) is an open standard that provides a universal way to connect AI systems with data sources, replacing fragmented, custom integrations with a single protocol. Wiring your MCP stack into your existing IdP using OAuth 2.1 and OIDC ensures that AI agents use the same trusted identity and access management frameworks as the rest of your enterprise. This article explains how to achieve that integration, covering direct OAuth flows, advanced token exchange for delegated access, and the division of responsibilities between the IdP and the MCP resource servers.

Enterprise Identity Integration Overview

In a typical enterprise environment, the **Identity Provider (IdP)** acts as the **OAuth2/OIDC authorization server**, issuing tokens after authenticating users. The MCP server and its tools, though, should function as **OAuth2 resource servers**, consuming and validating those tokens. In other words, the IdP remains the single source of truth for identity and access, while the MCP server trusts the IdP's tokens to authorize agent requests. This separation follows OAuth best practices and avoids making each MCP tool its own mini-IdP. In fact, if an MCP server tried to implement all the **/authorize** and **/token** logic itself, it would essentially be acting as an OAuth provider—something that is "extremely difficult to get right, which is precisely why most organizations rely on dedicated IdPs for this task". Instead, we delegate all authentication to the enterprise IdP and keep the MCP server focused on enforcing access to the data (resource) it fronts.

By using your existing IdP as the authorization server, you can leverage features like single sign-on, multi-factor auth, centralized user management, and compliance auditing for all AI agent interactions. OIDC builds on OAuth 2.0 to provide an identity layer (with ID tokens for user info), enabling SSO between applications. When a user or agent authenticates via the IdP, you get an access token for calling the MCP API and an ID token to identify the user. The MCP server doesn't issue its own credentials; it trusts the tokens from the IdP. This aligns with enterprise OAuth patterns—use the existing IdP as the authorization server (supporting flows like authorization code or client credentials) and have MCP servers act solely as resource servers, verifying tokens issued by the IdP. Each component has a clear role: the IdP authenticates and authorizes, while the MCP server (or its APIs) enforces access to the protected resources (databases, repositories, etc.).

Note: In the earliest draft of the Model Context Protocol, the MCP server was expected to function as a "mini OAuth server." Feedback from the identity-management community prompted a redesign, and the latest draft now introduces a clear separation between MCP and OAuth responsibilities.

Authorization Server vs. Resource Server: Separation of Concerns

It's crucial to architect the integration with a strong **separation of concerns** between the IdP (AuthZ Server) and the MCP server (Resource Server). In this setup, **all OAuth 2.1 interactions (authorization codes, token issuance, refresh, etc.) occur at the IdP** (or dedicated Secure Token Service)—not at the MCP server. The MCP server's job is to validate incoming tokens and serve data if the token is valid and sufficiently privileged. This mirrors the classic web API model: clients obtain a bearer token from a central auth server and then call protected APIs with that token. In practice, this means your MCP server trusts whatever tokens your IdP issues, and does not itself implement user logins or token issuance. The MCP server can remain stateless regarding auth (no token database needed), simply verifying token signatures or introspecting tokens on each request.

One immediate benefit of this approach is consistency and security. All policies (password rules, 2FA requirements, session lifetime, etc.) are enforced by the IdP globally. If an employee leaves or roles change, their access to the MCP-connected tools is updated centrally. Furthermore, this design aligns with **Zero Trust** principles—each request to the MCP resource is individually authenticated via a token that the IdP issued and the MCP validates. There's no "backdoor" or separate credential system for the AI tools; it's all under your existing identity infrastructure. If the MCP server receives a request without a valid token (or with an expired/insufficient one), it should reject it with an HTTP 401 Unauthorized. Ideally, it can also guide the client on how to obtain a token. For example, the MCP server's response could include a **WWW-Authenticate** header pointing to the IdP's authorization URL and discovery document, like so:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="MCP",
  error="invalid_token",
  authorization_uri="https://<your-idp>/oauth/authorize",
  discovery_uri="https://<your-idp>/.well-known/oauth-authorization-server",
  token_type="Bearer"
```

This way, any MCP client (including AI agents or developer tools) knows where to initiate the OAuth flow to get a proper token for the next request. With roles separated, we can now look at the specifics of wiring OAuth flows into the MCP ecosystem.

Direct OAuth 2.1 Flows with the IdP and MCP

Direct integration of MCP with your IdP means that when an AI agent (or a user using an AI-driven app) needs to access an MCP-protected resource, it will go through a **standard OAuth 2.1 flow** against the enterprise IdP. The MCP server's endpoints will simply expect a valid bearer token in the **Authorization** header of incoming requests. Let's break down two common scenarios: an interactive user-driven flow using Authorization Code (with OIDC) and a machine-to-machine flow using Client Credentials.

User Authorization Code Flow (Interactive)

In this flow, a human user needs to grant an AI agent (or application) access to an MCP tool on their behalf. OIDC often comes into play here to provide user identity. The high-level process is:

1. Client Registration
Register the AI application or agent as an **OAuth client** in your IdP. Configure redirect URI(s) and permitted grant types (use Authorization Code with PKCE, and request **OpenID Connect** scopes if you need an ID token). Also register or configure the **MCP resource** in the IdP if required—for example, by defining custom scopes or audience identifiers that represent the MCP server's APIs in the IdP's configuration. (Some IdPs let you define an API resource with scopes that clients can request.)

2. Initiate Auth Flow
When the agent or user tries to use an MCP tool, the client application redirects the user to the IdP's **/authorize** URL (or uses an OAuth SDK) with the appropriate parameters (client ID, redirect URI, requested scopes, etc.). For instance, the client might request scopes like **mcp.read** or **mcp.tools.slack**—whatever scopes your IdP uses to represent allowed MCP operations.

3. User Login & Consent
The IdP authenticates the user (e.g., via SSO login, possibly MFA if configured). Once authenticated, the IdP shows a consent screen if the application requests access on the user's behalf (unless pre-consented by policy). The user grants access to the requested scopes for the MCP tool.

4. Token Issuance
The IdP then issues an **authorization code** (since we use Auth Code flow) and redirects back to the client's callback URI with that code. The client app exchanges the code at the IdP's **/token** endpoint (providing its client credentials or using PKCE if it's a public client) to receive tokens. The IdP responds with an **access token** (and typically a **refresh token** for longer-lived access, and an **ID token** if OIDC scope was requested). The access token is the credential that will be used to call the MCP server. It could be a JWT or an opaque token—either way, it is issued by the IdP and represents the user's granted rights.

Direct OAuth 2.1 Flows with the IdP and MCP (cont.)

5. Call MCP Server
Armed with the access token, the client (or the AI agent on behalf of the user) invokes the MCP server's API (e.g., a specific tool endpoint) with the header **Authorization: Bearer <access_token>**. Now the MCP server receives the request along with the token.

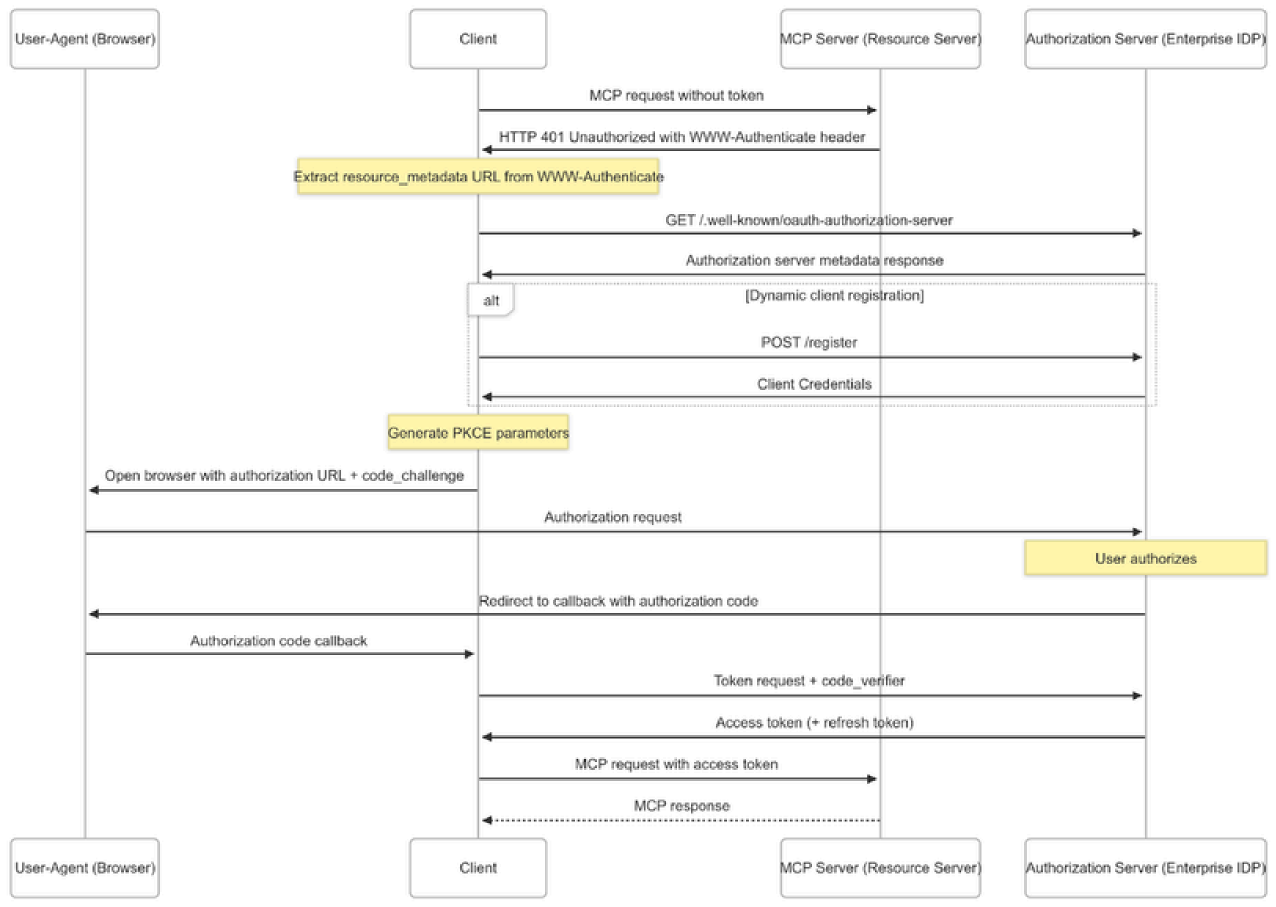
6. Token Validation (Resource Server)
The MCP server validates the token. If it's a JWT, the MCP checks the signature using the IdP's public keys (usually retrieved from the IdP's OIDC **/.well-known** JWKS endpoint), verifies the token is not expired, and verifies the token's **issuer** matches the IdP and the **audience** claim corresponds to the MCP resource. It also checks that the token has the necessary scope or claims for the requested operation (for example, if the user is trying to access an inventory list through the MCP, the token might need a scope like **inventory.list**). If the token is opaque, the MCP server (or an API/AI gateway in front of it) may perform introspection by calling the IdP's introspection endpoint to get token details. Either way, the MCP trusts the IdP as the authority on token validity.

7. Authorize Request
If the token checks out, the MCP server allows the requested operation and returns the data or performs the action. At this point, the user (or agent) gets access to the data through the MCP. If the token is missing or invalid, the MCP returns 401 as described earlier, prompting authentication.

8. (Optional) Use ID Token
If an OIDC **ID token** was issued alongside the access token, the client/agent can use it to know the user's identity (subject, email, etc.) or even pass some user context to the AI model. The MCP server typically doesn't use the ID token (it cares only about the access token for authorization), but the presence of OIDC means the system as a whole knows who the user is. This can be useful for logging or for the AI agent to personalize responses, etc., without the MCP server needing its own user database.

In the sequence on the following page, the heavy lifting of user authentication is done by the **IdP**, and the MCP server only sees the final access token presented with the request. This direct OAuth flow ensures the MCP server fits neatly into the enterprise security architecture. As a software architect, you should ensure that the MCP server is configured with the IdP's metadata (issuer URL, JWKS keys, accepted audience values, etc.) so that it **knows how to validate tokens**. Most enterprise OIDC IdPs (Azure AD, Okta, Auth0, Ping, etc.) publish a discovery document (**.well-known/openid-configuration**) which the MCP server or its gateway can use to fetch keys and token info dynamically. Additionally, if your MCP server supports it, you might configure accepted **scope names or roles** that map to actions on the MCP tools. For instance, a token with scope **mcp:slack:read** might allow an AI agent to read messages via an MCP Slack integration tool.

Direct OAuth 2.1 Flows with the IdP and MCP (cont.)


User-Agent (Browser)
Client
MCP Server (Resource Server)
Authorization Server (Enterprise IDP)
MCP request without token
HTTP 401 Unauthorized with WWW-Authenticate header
Extract resource_metadata URL from WWW-Authenticate
GET /.well-known/oauth-authorization-server
Authorization server metadata response
alt
[Dynamic client registration]
POST /register
Client Credentials
Generate PKCE parameters
Open browser with authorization URL + code_challenge
Authorization request
User authorizes
Redirect to callback with authorization code
Authorization code callback
Token request + code_verifier
Access token (+ refresh token)
MCP request with access token
MCP response

Service-to-Service Client Credentials (Non-Interactive)

Not all agentic AI scenarios involve an interactive user in real time. Sometimes, an **AI agent service** or an MCP tool might need to authenticate itself to another service for background tasks (e.g., a nightly data synchronization, or an AI agent that runs with a service identity to maintain an index). For such cases, the **Client Credentials** flow is appropriate. In this flow, there is no end-user; the client (which could be the MCP server itself in some sub-component, or another service) obtains an access token representing its own identity or a predefined machine identity.

Key points of the client credentials integration:

- **Client App Registration:** Register a machine client in the IdP for the agent or service that needs to call the MCP tool. For example, create a client application in the IdP and configure it with Client Credentials grant enabled. Typically, this client will have a client ID and a client secret (or a certificate for auth) issued by the IdP.
- **Scope and Audience:** In the IdP, grant this client application the necessary scope or audience to access the MCP resource. For instance, you might have a scope `mcp.backup` that allows reading a database via the MCP server. Assign that scope to the client or configure a policy that the client is allowed to request it.

Direct OAuth 2.1 Flows with the IdP and MCP (cont.)

- **Token Retrieval:** The client service calls the IdP's **/token** endpoint directly with its client ID/secret (using HTTP basic auth or a JWT assertion, depending on IdP setup), requesting the configured scope. This is a server-to-server call. The IdP authenticates the client and issues an access token (no ID token since no user). The token will typically have the client itself as the **sub** (subject) and will be scoped for the MCP resource.
- **Call MCP with Token:** The service then calls the MCP server's API with the bearer token like before. The MCP server validates it similarly (checks signature, issuer, audience = MCP, and that the token's scopes or roles permit the requested operation).
- **Authorization:** The MCP can enforce that only certain operations are allowed by a client credentials token (since it might be all-powerful or limited by design). If valid, it serves the request.

This flow allows internal services or headless AI processes to use MCP tools without a user context, using the enterprise IdP for authentication. The result is still fully tracked and managed by the IdP (for example, you can revoke the client credentials or monitor usage). By using OAuth 2.1 here, you also get modern security features (private key JWT or mutual TLS client auth if configured, mandatory PKCE isn't relevant in client cred but other 2.1 improvements like token binding could apply).

Tip: In many cases, you might front your MCP server with an **API Gateway or service mesh** that handles the OAuth token validation for you. The gateway can require a valid JWT from the IdP, perform verification and scope checks, and then forward the request to the MCP server with some context (for example, injecting an identity header or using mTLS). This is an optional architecture choice, but it can simplify the MCP server implementation and provide a centralized point for things like rate limiting and logging. Regardless of whether a gateway is used, the core idea remains: the IdP is the authority for issuing tokens, and tokens are validated where the requests hit.

Token Exchange and Delegated Authorization (On-Behalf-Of)

Direct OAuth flows cover the scenario where a client obtains a token and uses it to call a single MCP server. But what happens in more complex agentic workflows, where one service needs to call another on behalf of the same user? For instance, imagine an AI agent that first queries a database via one MCP tool, then needs to call a different MCP tool (say, a CRM system) using the data it found—all under the original user's authority. We want to avoid asking the user to log in again for the second service, and we definitely should not be sharing the original access token with a service it wasn't intended for. This is where the **OAuth 2.0 Token Exchange extension (RFC 8693)** comes into play, enabling delegated authorization or so-called **on-behalf-of (OBO)** patterns.

Token Exchange (RFC 8693) is a standard grant type that allows one token to be exchanged for another. In essence, it lets a client (or service) swap an existing token for a new token that might have a different audience, scopes, or other attributes. According to RFC 8693, you supply a **subject token** (the original token representing the user or entity on whose behalf you act) and can optionally supply an **actor token** (a token representing the calling service itself, if we need to distinguish the calling service's identity). The Authorization Server (your IdP acting as a Security Token Service) will validate the incoming token(s) and issue a new token as requested, if policy allows. This new token can impersonate the original user or indicate delegation by nesting the original identity inside it.

In practice, the **on-behalf-of flow** works like this:

1. Initial Token
The user or client obtains an access token (Token A) from the IdP in the normal way (e.g., via auth code flow). This token is intended for Service A (audience = Service A, which could be an MCP tool).

2. Call to Service A
The client calls Service A (e.g., an MCP server Tool A) with Token A. Service A validates it and processes the request.

3. Need to Call Service B
To complete its operation, Service A now needs data from Service B (another protected resource, maybe another MCP tool or external API) on behalf of the same user. Service A itself is acting as a client to Service B, and it has the user's context from Token A.

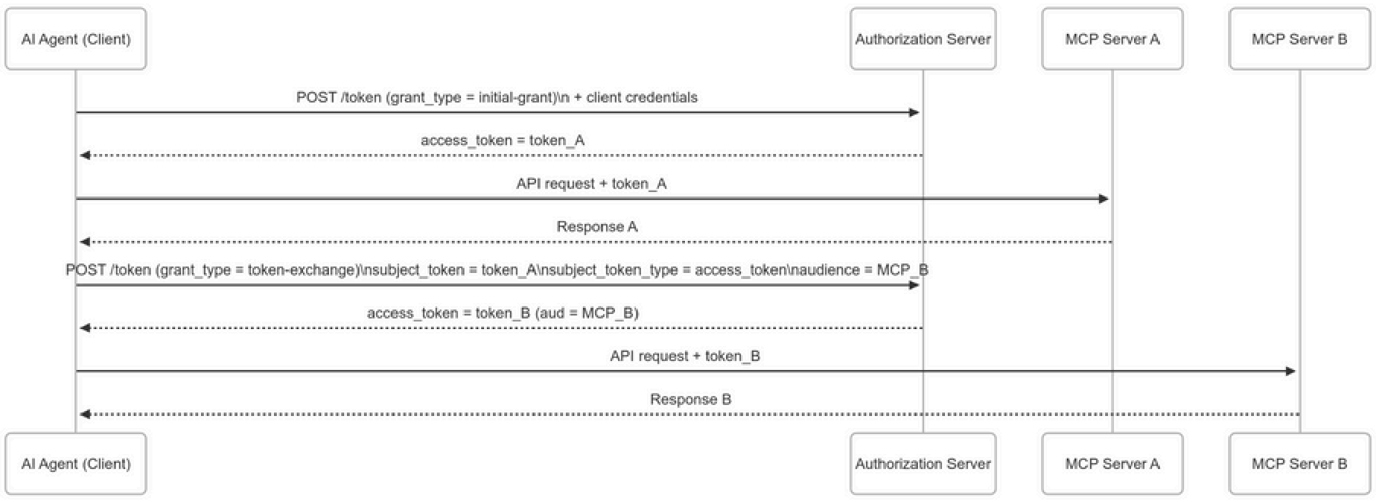
Token Exchange and Delegated Authorization (cont.)

4. Token Exchange Request
Service A sends a request to the IdP's **token endpoint**, using the **token exchange grant** (often identified by **grant_type=urn:ietf:params:oauth:grant-type:token-exchange**). It includes the original **Token A as a subject_token**, and it identifies the target resource or audience for the new token (Service B). Service A also authenticates itself to the IdP (for example, using its own client credentials or a JWT as the actor token to prove it is allowed to act on behalf of the user). The request basically says: "Exchange this user token for a new token that I (Service A) can use to call Service B."

5. IdP Validation and New Token
The IdP (STS) verifies that Token A is valid and that Service A is authorized to perform this exchange (there may be pre-configured trust that Service A can act on user's behalf for calls to Service B). If all checks pass, the IdP issues **Token B**—an access token intended for Service B. This token will carry the user's identity (so Service B knows who the end-user is), and typically will also carry an indication that Service A is the one acting. For example, in a JWT, the **sub** claim might be the user, and there might be an **act** (actor) claim or an **azp** showing Service A. The scopes of Token B are limited to what Service A is allowed to do on Service B. The new token will allow API1 to make requests to API2, and it retains the user context so the API2 service knows on whose behalf the request is made.

6. Call to Service B
Now Service A calls Service B's API, presenting **Token B** in the Authorization header. Service B validates Token B (signature, audience = Service B, etc., just like any resource server would with the IdP's token) and sees that the user (subject) is authorized via Service A's delegation. Service B then performs the requested action and returns the result to Service A.

7. End-to-End Outcome
The user's request initiated at Service A is fulfilled, with Service A orchestrating a call to Service B without ever needing the user to directly provide a separate credential to Service B. The user's identity and permissions flowed securely through the system using tokens, and each service only accepted tokens meant for itself.


AI Agent (Client)
Authorization Server
MCP Server A
MCP Server B
POST /token (grant_type = initial-grant)\n + client credentials
access_token = token_A
API request + token_A
Response A
POST /token (grant_type = token-exchange)\nsubject_token = token_A\nsubject_token_type = access_token\naudience = MCP_B
access_token = token_B (aud = MCP_B)
API request + token_B
Response B
AI Agent (Client)
Authorization Server
MCP Server A
MCP Server B

Example of an OAuth 2.0 on-behalf-of token exchange flow

Token Exchange and Delegated Authorization (cont.)

The benefit of this pattern is that it **preserves the chain of trust and least privilege**. Service A didn't just forward Token A to Service B (which might violate audience restrictions or expose more privilege than necessary); instead, it got a fresh Token B that is precisely scoped for Service B. The IdP can keep audit logs linking Token B back to the original user and Service A (for compliance, you can often trace "which service acted on behalf of which user" in logs). In some IdPs, this is called the On-Behalf-Of grant, and it is a form of delegated access. OAuth 2.0 Token Exchange enables delegation and impersonation, allowing a client to act on behalf of a user or another client. Changing the **subject** of a token (the user identity) in this way is effectively impersonation (the new token impersonates the user toward Service B), whereas carrying the original user while still recognizing the acting party is a delegation. RFC 8693 supports both styles depending on how the IdP implements the claims in the new token.

From an implementation standpoint, to use token exchange in your MCP integration, your IdP must support RFC 8693 or a similar OBO mechanism. Some popular enterprise IdPs do, though sometimes it requires enabling an extension or special configuration. You would register the **delegating service (Service A)** in the IdP and give it permission to request token exchange for certain targets. In our context, Service A could be one MCP tool and Service B another, or Service A could even be the core MCP server and Service B an external API the MCP server needs to call internally. The general principle stays the same. Make sure to design the scopes and trust relationships carefully. Service A should only be able to get tokens for Service B if that's intended in your architecture.

Preparing for Trust Frameworks and Federation

By wiring your MCP server into your enterprise IdP, you've set a strong foundation for **secure and efficient agentic AI integration.** Every request an AI agent makes to your MCP-hosted tools is authenticated and authorized using the same centralized identity system that the rest of your enterprise relies on. This not only reduces the risk of mismanaging credentials but also streamlines user experience (leveraging SSO) and administration (consistent access revocation and auditing).

Looking forward, this integration prepares the ground for more advanced identity architectures such as **trust frameworks** and **federated identity**. In the future, you may want your AI agents to access tools or data across organizational boundaries or cloud providers. Federated identity (via protocols like OpenID Connect Federation 1.0) would allow your MCP server to accept tokens from a partner's IdP, or vice versa, based on established trust relationships.

Preparing for Trust Frameworks and Federation (cont.)

Having your MCP stack already speaking OAuth/OIDC with your IdP makes it much easier to plug into such federations—the MCP server can still behave as a resource server, but now it might trust multiple issuers (your enterprise IdP, plus perhaps a partner's IdP or a higher-level federation authority). Similarly, trust frameworks can be layered on top of OAuth/OIDC to enforce consistent security standards across all parties (for example, requiring certain token claims or assurance levels). Because you've kept your MCP implementation aligned with standard OAuth 2.1 and OIDC practices, it will be compatible with these broader ecosystem extensions.

In summary, integrating MCP with your IdP involves configuring OAuth 2.1/OIDC flows such that AI agents obtain and use **IdP-issued tokens** for all MCP requests. We covered how to do this for direct user-driven and client-driven scenarios, and how to handle delegated calls with token exchange. With this setup, your agentic AI system speaks the same language of trust as the rest of your enterprise services. **Part 4 and Part 5 of this series** will build on this by exploring multi-IdP federation and trust frameworks, taking your MCP deployment to the next level of scale and collaboration. For now, you can be confident that your AI agents are securely authenticated and authorized, plugged into your existing identity infrastructure—a critical step toward a robust, enterprise-grade agentic AI ecosystem.

About SecureAuth

More security shouldn't mean more obstacles. Since 2005, SecureAuth has helped leading companies simplify identity and access management for customers and employees—creating experiences that are as welcoming as they are secure.

SecureAuth is redefining authentication for the modern enterprise. Today's evolving threat landscape demands innovative, adaptive security solutions. As the first-to-market provider of continuous facial authentication, we go beyond the initial authentication to deliver ongoing security throughout the entire session. Our mature AI-driven risk engine delivers dynamic—and often invisible—authentication, making you more effective than ever at eliminating threats while ensuring frictionless, secure access for employees and customers.

Welcome to Better Identity.

SECUREAUTH