

AI Integration into SDLC with MCP

Integrating AI into the Software Development Life Cycle (SDLC) often involves enabling AI agents (like coding assistants in Cursor IDE or chatbots) to interact with development tools (e.g. GitLab repositories, Jira trackers) on behalf of users. The **Model Context Protocol (MCP)** is an emerging standard that makes this possible by connecting AI applications to external systems in a standardized way modelcontextprotocol.io. For example, an MCP server can expose a tool's API (GitLab, Jira, etc.) to an AI assistant, allowing the assistant to perform tasks like retrieving issues or committing code. This can greatly enhance developer productivity as part of a digital transformation plan. However, **managing authentication and access control in a multi-user environment** is a critical challenge in this setup.

The Challenge: Multi-User Access and Permissions

In a real-world enterprise, different users have different rights and permissions in tools like GitLab or Jira. The AI integration must respect these permissions. For instance, if a junior developer's assistant tries to merge code via GitLab or update a Jira issue, it should only do so if that developer has the rights. The core challenge is **how to ensure the AI agent uses each user's own credentials and permissions when accessing these tools**, especially when using a shared MCP server or service.

By default, many MCP servers were designed for single-user scenarios. Early MCP implementations assumed a local, single-user context – often using environment variables or static tokens for access medium.com. This works for prototypes (or when each user runs their own MCP server locally), but it **breaks down in multi-user production environments** medium.com. Using one set of credentials for all users is *“not multi-user friendly... not scalable – every user ends up impersonating the same bot identity”* medium.com. Clearly, that approach violates security and audit requirements, since actions wouldn't be attributed to the correct user.

Native MCP Server Capabilities (First Hop vs Second Hop)

It's important to distinguish two stages of authentication in the MCP model medium.com:

1. **First Hop (Client → MCP Server):** Authenticating the user when they connect to the MCP server (or agent). This decides *whether* a user is allowed to use the MCP server's capabilities at all medium.com. For example, can the user access the “GitLab tool” provided by the AI agent?

2. **Second Hop (MCP Server → External Tool):** Authenticating the MCP server's requests to the downstream tool (GitLab, Jira, etc.) on behalf of the user medium.com. This ensures the actions the agent performs use that user's privileges.

The **MCP specification currently covers only the first hop** – it defines how an MCP server can act as an OAuth2.1 resource server to accept tokens from clients (so you can restrict who can connect to it) medium.com. Recent updates to the spec allow using OAuth 2.1 for this, meaning an MCP server can be protected behind an authorization server (like an enterprise SSO/IdP) that issues access tokens to clients. In practice, this means you *can* enable SSO/OAuth login for the MCP server itself (e.g. requiring an access token for the AI agent to talk to the MCP server) medium.com. This addresses user authentication **to the agent** (first hop).

However, native MCP servers do not inherently handle the second hop (downstream tool auth) for multiple users medium.com. Most existing MCP server implementations assume a single set of credentials for the downstream API – often configured via environment variables or config files medium.com. They don't provide built-in support for switching credentials based on which user is invoking the AI. In other words, if you just deploy a stock MCP server for (say) GitLab, it likely uses one API token internally, treating all requests the same. This is clearly insufficient when different users have different access rights.

Using SSO and OAuth for User Authentication (First Hop)

Single Sign-On (SSO) can play a key role in the first hop. You'll want to integrate the MCP client or AI agent with your SSO/identity provider so that only authenticated users can use the AI's tools. For example, you might require users to log in with corporate SSO (OAuth 2.1) to obtain a token, and the MCP server will validate that token before allowing any tool usage medium.com. The MCP spec's authorization flow supports this by treating the MCP server as a protected resource; an **API gateway or auth proxy** in front of the MCP server can validate incoming JWT access tokens issued by your IdP medium.com. This ensures that **each request to the MCP server is tied to an authenticated user identity**.

By using SSO, you not only control access, but also get an audit trail of *who* is using the AI assistant. This addresses part of the problem (preventing unauthorized use), but we still need to handle the *credentials for downstream APIs*. SSO might simplify that if your tools (GitLab, Jira, etc.) are federated with the same IdP – but typically, you'll still need specific tokens for each service.

MCP Proxy for Multi-User Tool Access (Second Hop)

To manage per-user access to external tools, the common approach is to introduce an **MCP Proxy or Orchestrator** between the AI client and the MCP servers. The proxy's job is to handle

the “**two-hop**” **authentication problem** by injecting the appropriate credentials on the second hop [medium.commedium.com](#). In essence:

- The proxy/service implements OAuth flows or token management for each external tool **on behalf of each user**. For example, the first time a user wants the AI to access Jira, they would go through an OAuth authorization code flow to grant access to their Jira account. The proxy captures the OAuth tokens and stores them securely (e.g. encrypted in a database or vault) keyed by user and service [medium.com](#). This establishes a user-specific session for that tool.
- On subsequent AI requests, the proxy checks which user is making the request (from the first hop token or session context) and retrieves the corresponding stored credentials for the target service. It then **injects the correct user’s token into the MCP server’s request** to the external API [medium.com](#). For instance, if user Alice’s AI agent calls the “GitLab MCP server” to create an issue, the proxy would attach Alice’s OAuth token for GitLab in that request. If user Bob is invoking it, the proxy uses Bob’s token.
- The MCP server (the tool integration) doesn’t need to know about multiple users at all – it simply sees a request with a valid token and uses it. By the time the request reaches the MCP server, *“it already has all the valid user tokens in the request”* [medium.com](#), so it can act as if it were that user calling the API.

This proxy pattern cleanly **decouples the auth logic from the MCP servers** themselves. Each MCP tool server can remain stateless and single-purpose, while the proxy/orchestrator handles session management and credential injection [medium.commedium.com](#). It also centralizes security: tokens can be encrypted and rotated in one place, and the proxy can refresh tokens when needed [medium.commedium.com](#). In effect, *“the API gateway (or proxy) validates the user, injects the right tokens, and the MCP host runs securely as if the user themselves made the call.”* [medium.com](#)

Do Native MCP Servers Support This?

Out of the box, **most MCP servers do not support multi-user credential management**. As noted, they assume one credential context [medium.commedium.com](#). The MCP specification didn’t originally define how to pass user-specific tokens for downstream calls [medium.com](#). Therefore, you would need to extend or wrap the servers to handle it. Some options include:

- **Running separate MCP server instances per user:** For example, spin up an isolated MCP tool server for each user session with that user’s credentials. This can be orchestrated by an agent platform, but it’s complex to scale manually.
- **Using libraries/extensions for multi-user support:** The community has created tools like **MCPresso** (an OAuth2.1 library for MCP servers) that *“gives your MCP server multi-user authentication (OAuth 2.1 + PKCE) and automatically injects the user context*

into every handler”, avoiding manual token handling [reddit.com](#). Such libraries essentially build the “proxy” logic into the server itself.

- **Hosted MCP orchestration services:** There are emerging platforms (e.g. Arcade.dev, mcp.run, etc.) that offer managed solutions to handle multi-user auth and aggregate tools behind a unified interface [reddit.com](#) [reddit.com](#). These can simplify setup by providing built-in user auth and a vault for API tokens.

Despite these evolving solutions, the fundamental approach is the same: **an extra layer is required to manage user identities and credentials**. Native MCP servers alone are not enough for multi-user scenarios – you will either incorporate an auth proxy or use an enhanced server implementation.

Recommended Approach: SSO + Proxy for Secure Multi-User AI Integration

To answer the question directly: *Should you rely on native MCP server capabilities or use an MCP proxy for managing user access?* The evidence suggests you should **use both SSO and an MCP proxy solution** to cover all bases:

- **Secure the First Hop with SSO/OAuth:** Leverage your SSO (OAuth 2.1 identity provider) so that when users access the AI assistant (e.g. in Cursor IDE with MCP), they authenticate and obtain an access token. Configure the MCP server (or an API gateway in front of it) to require and validate this token [medium.com](#). This ensures only authorized users can reach the MCP endpoints, and it ties requests to a user identity for auditing [modelcontextprotocol.io](#).
- **Handle Second Hop via Proxy/Orchestrator:** Implement an MCP proxy or authentication layer that manages connections to GitLab, Jira, and other tools on behalf of users. Through OAuth flows and token storage, it will supply the appropriate credentials to the MCP server calls [medium.com](#) [medium.com](#). This way, the AI agent always acts **with the requesting user’s privileges** – no more, no less – in those external systems. You could build this yourself (as a microservice using OAuth libraries) or use existing frameworks that address the “two-hop” auth problem.

By combining these, you fulfill enterprise security requirements. Each user’s actions through the AI are authenticated (you know who they are) and authorized (using that user’s own access rights). The AI integration will not bypass permission controls because it never uses a generic admin token – it uses the user’s token or a delegated token with only their scope [medium.com](#) [dev.to](#).

In summary, **native MCP servers alone don't natively handle multi-user permission differences**. You will need to introduce an authentication layer (for SSO) and likely an MCP proxy/orchestrator to manage user-specific tool access. This approach is confirmed by community best practices and recent literature: *"Most [MCP servers] assume a single-user context... not multi-user friendly... What we want is secure OAuth-based connections, so AI agents can act on behalf of the end user"*[medium.commedium.com](https://medium.com/medium.com). Therefore, for a robust digital transformation plan, plan to integrate both **SSO** (for identity/authentication) and an **MCP proxy** (for per-user authorization to downstream tools) rather than relying on the MCP server's default behavior alone.

Key Takeaways

- **MCP (Model Context Protocol)** enables AI assistants to use external tools in the SDLC (like GitLab, Jira), but was initially built with single-user assumptions.
- **Different users' access rights** must be respected – an AI agent should only perform actions permitted to that specific user.
- **Native MCP servers** support OAuth-based auth for client→server (first hop) but do *not* inherently manage multi-user credentials for tool APIs (second hop)[medium.com](https://medium.com/medium.com).
- **Use SSO/OAuth for the first hop:** Protect the MCP server or AI service with your SSO to authenticate users and issue tokens[medium.com](https://medium.com/medium.com).
- **Use an MCP proxy (or similar auth layer) for the second hop:** Handle OAuth flows to GitLab/Jira per user, store tokens, and inject them into requests so the AI acts under each user's own identity[medium.commedium.com](https://medium.com/medium.com).
- **Combining both** gives a secure, scalable setup: *"the API gateway validates the user, the proxy injects the right tokens, and the MCP host runs securely as if the user themselves made the call."*[medium.com](https://medium.com/medium.com)

By following this approach, your AI integration into the SDLC will be aligned with enterprise security practices, ensuring that automation and intelligence do not come at the cost of compliance or security. The **digital transformation plan** should thus include setting up the necessary authentication infrastructure (SSO integration, token management service) alongside any MCP servers and custom commands in the IDE. This will allow different users to safely access AI-driven tools according to their roles and permissions, fulfilling the customer's requirements for AI in the SDLC.

Sources:

- Model Context Protocol documentation and community discussions on multi-user authentication [medium.commedium.commedium.com](#)
- Insights from industry articles on implementing OAuth and proxy patterns for MCP in production [medium.commedium.commedium.com](#)
- Reddit Q&A from the MCP community about multi-user setups and available solutions