Python + MCP

🛠️ Dec 16: Building MCP servers with FastMCP

☁️ Dec 17: Deploying MCP servers to the cloud

👤 Dec 18: Authentication for MCP servers

🔗 Register at aka.ms/pythonmcp/series

Python + MCP

- 🛠️ Building MCP servers with FastMCP
- ☁️ Deploying MCP servers to the cloud
- 👤 Authentication for MCP servers

Microsoft

Python + MCP

👤 Authentication for MCP servers

aka.ms/pythonmcp/slides/auth

Pamela Fox

Python Cloud Advocate

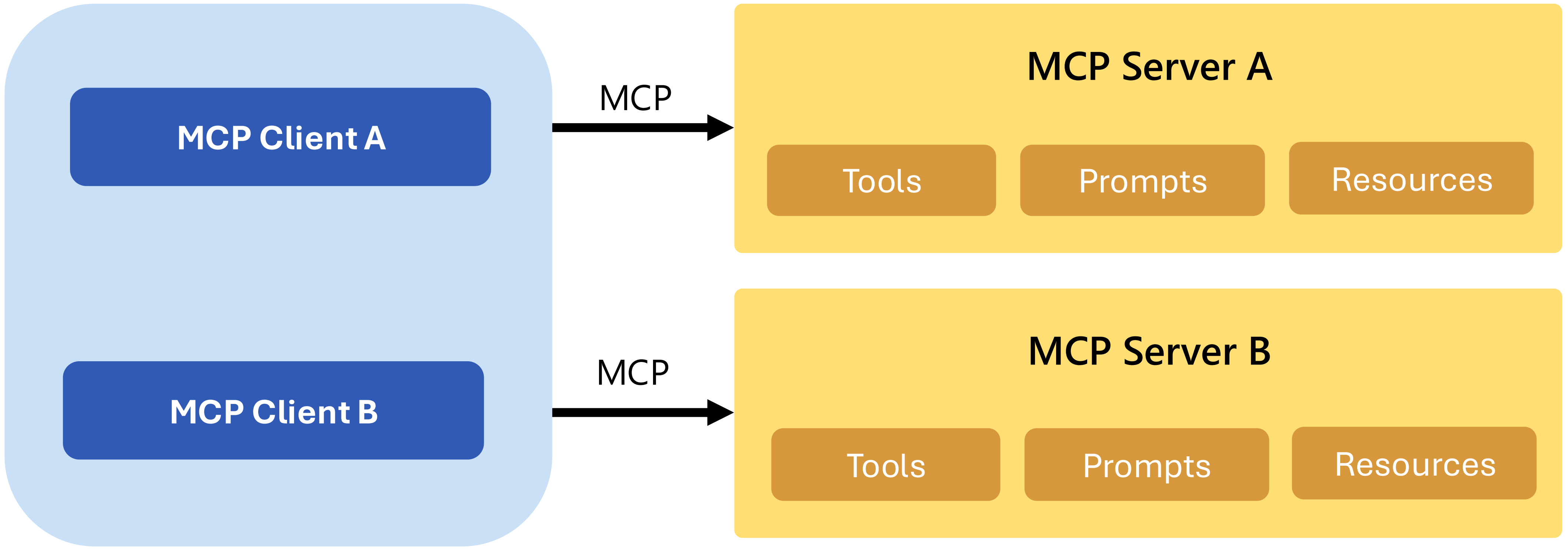
www.pamelafox.org

Today we'll cover...

- Restricting access to MCP servers
- Key-based access
- OAuth-based access
- Keycloak integration
- Entra integration

🛡️ Restricting MCP server access

Recap: MCP architecture

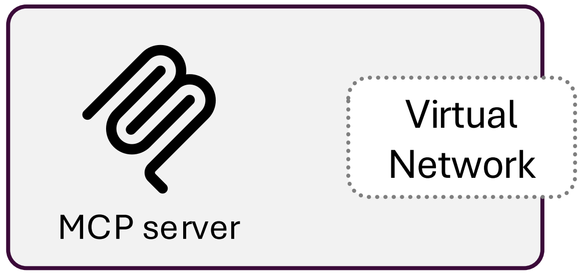


MCP clients may be inside desktop applications like VS Code/Claude Code, or from programmatic AI agents written with frameworks like Langchain.

Restricting access to MCP servers

These are the three primary approaches:

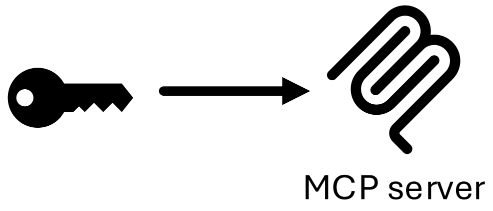
Private network



Access is allowed only within the restricted private network, or over VPN gateways into it.

Discussed in the 12/17 livestream.

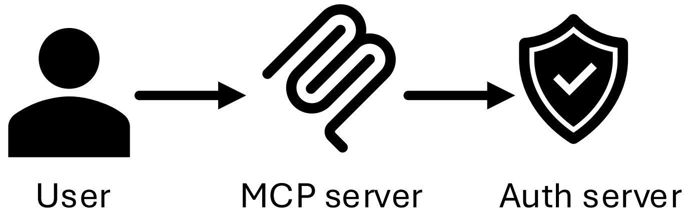
Key-based access



Access is granted with keys that are registered with the MCP server.

Discussing today!

OAuth-based access



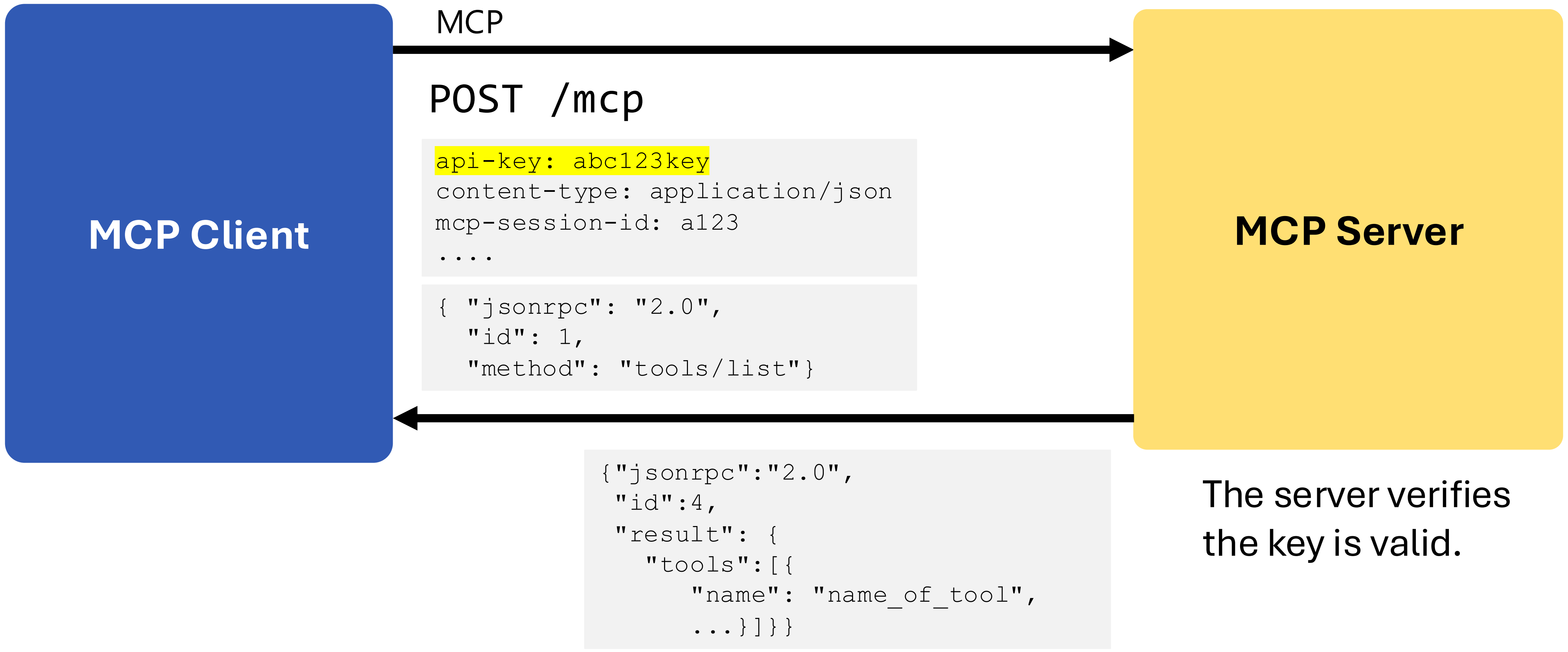
Access is granted based on OAuth2 flow between user, MCP client, authentication provider, and MCP server.

Discussing today!

🔐 Key-based access

Key-based access flow

A key is often specified in headers or URL query parameters:



```
POST /mcp
```

```
api-key: abc123key
content-type: application/json
mcp-session-id: a123
....
```

```
{ "jsonrpc": "2.0",
  "id": 1,
  "method": "tools/list"}
```

```
{"jsonrpc":"2.0",
 "id":4,
 "result": {
   "tools":[{
      "name": "name_of_tool",
      ...}]}}
```

The server verifies the key is valid.

Specifying a key for MCP server in VS Code

The Tavily MCP server supports key-based access:

```
{"servers": {
  "tavily-mcp": {
    "url": "https://mcp.tavily.com/mcp/",
    "type": "http",
    "headers": {
      "Authorization": "Bearer ${input:tavily-key}"
      }
    }}},
 "inputs": [{
   "type": "promptString",
   "id": "tavily-key",
   "description": "Tavily MCP API Key",
   "password": true
  }]
}
```

VS Code lets you designate keys as "password" inputs to reduce risk of exposure.

https://docs.tavily.com/documentation/mcp#remote-mcp-server

Specifying a key for MCP server in an AI agent

AI agent frameworks provide ways to customize the URL and headers.

agent-framework:

```
MCPStreamableHTTPTool(
  name="Tavily MCP",
  url="https://mcp.tavily.com/mcp/",
  headers={"Authorization": f"Bearer {tavily_key}"}
)
```

🔗 aka.ms/python-mcp-demos: agents/agentframework_tavily.py

langchain:

```
MultiServerMCPClient({
  "tavily": {
    "url": "https://mcp.tavily.com/mcp/",
    "transport": "streamable_http",
    "headers": {"Authorization": f"Bearer {tavily_key}"}}})
```

🔗 aka.ms/python-mcp-demos: agents/langchainv1_tavily.py

Deploying key-based access in Azure

Azure Functions

Azure Functions offers a basic key-based access option.
Most useful for internal tools with limited users.

Azure API Management

APIM offers an API key management system and developer portal.
Scalable and production ready.

...or build your own key management system.

Deploying Azure Function with key access

1. Open this GitHub repository:

   https://github.com/Azure-Samples/mcp-sdk-functions-hosting-python

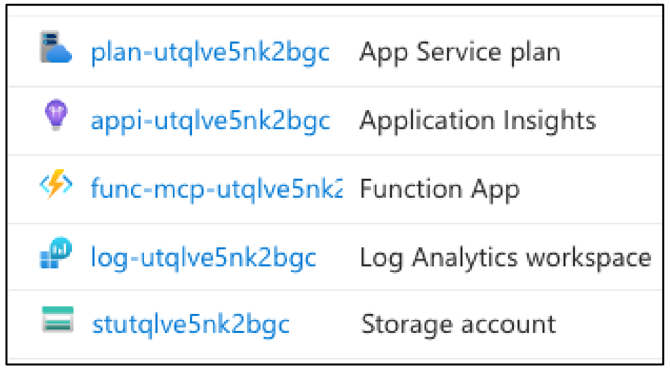
2. Change "**`DefaultAuthorizationLevel`**" to "**`function`**" in host.json

3. Deploy with Azure Developer CLI:

```
>> azd auth login
```

```
>> azd env set ANONYMOUS_SERVER_AUTH true
```

```
>> azd up
```

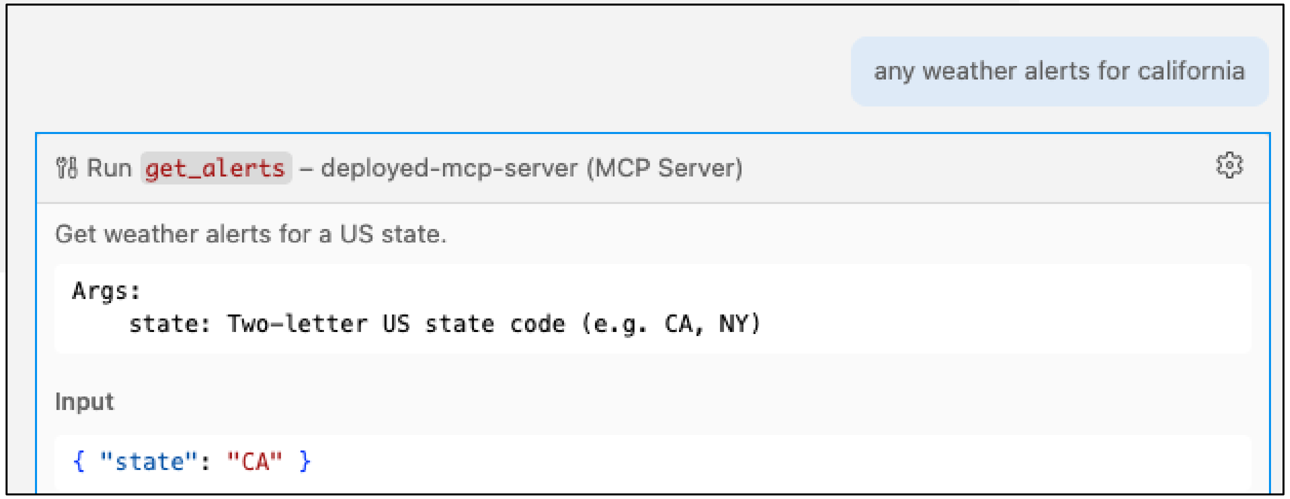


plan-utqlve5nk2bgc	App Service plan
appi-utqlve5nk2bgc	Application Insights
func-mcp-utqlve5nk2	Function App
log-utqlve5nk2bgc	Log Analytics workspace
stutqlve5nk2bgc	Storage account

Demo: Using deployed function from VS Code

.vscode/mcp.json:

```json
{"servers": {
  "deployed-mcp-server": {
    "url": "https://your-function-subdomain.azurewebsites.net/mcp",
    "type": "http",
    "headers": {
      "x-functions-key": "${input:functionapp-key}"
  }}},
 "inputs": [{
    "type": "promptString",
    "id": "functionapp-key",
    "description": "Server key",
    "password": true
}]}
```


any weather alerts for california

Run get_alerts – deployed-mcp-server (MCP Server)

Get weather alerts for a US state.

```
Args:
    state: Two-letter US state code (e.g. CA, NY)
```

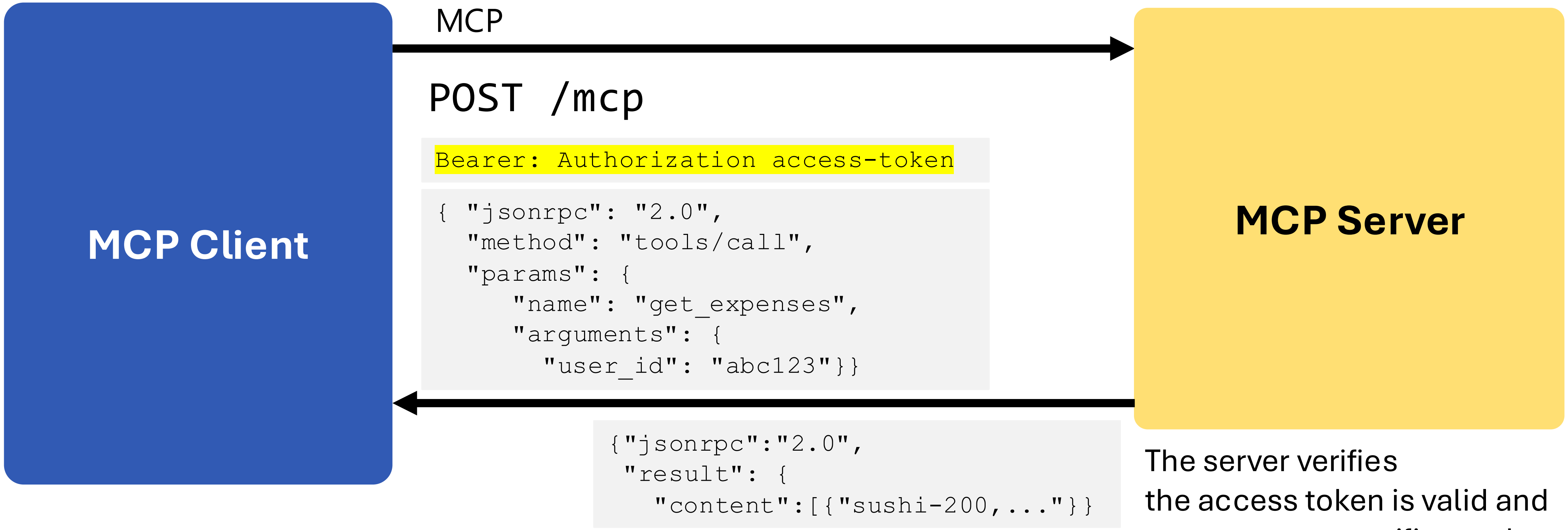
Input

```
{ "state": "CA" }
```

👤 OAuth-based access

OAuth-based access flow

MCP client can make requests to MCP servers on behalf of users:



```
POST /mcp
```

```
Bearer: Authorization access-token
```

```
{ "jsonrpc": "2.0",
  "method": "tools/call",
  "params": {
     "name": "get_expenses",
     "arguments": {
       "user_id": "abc123"}}
```

```
{"jsonrpc":"2.0",
 "result": {
   "content":[{"sushi-200,..."}}
```

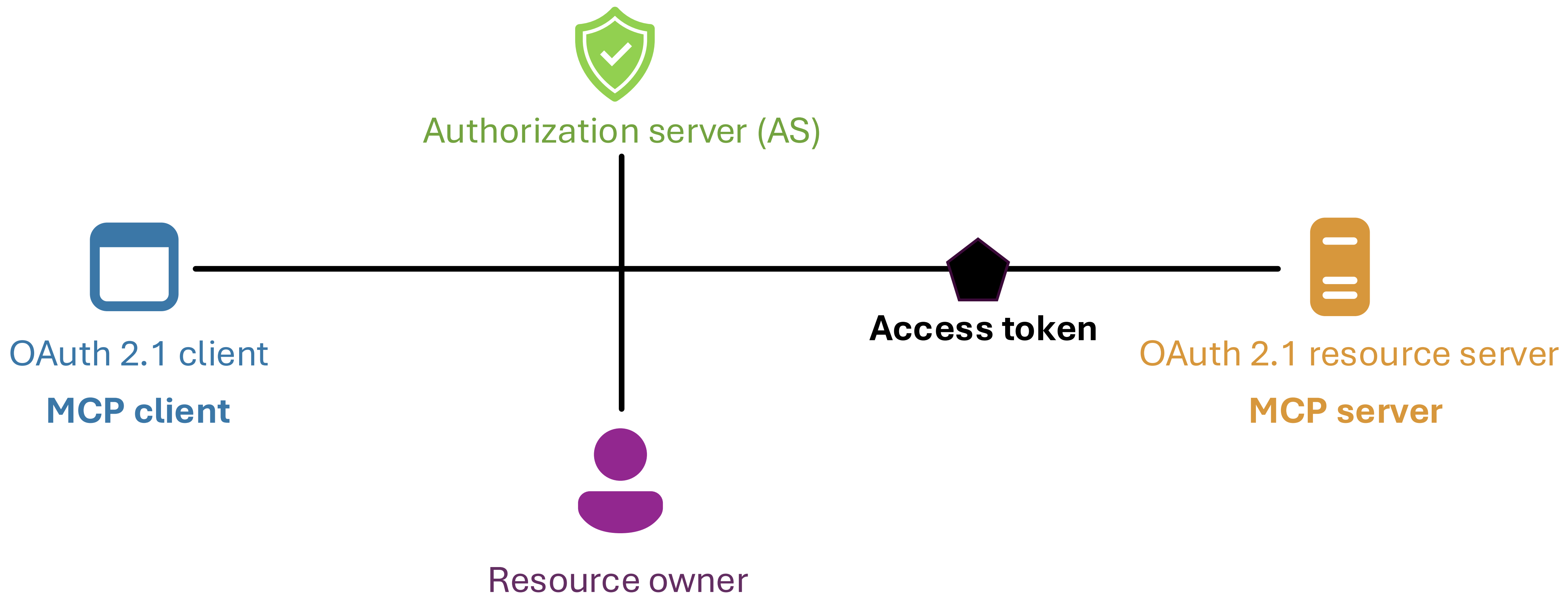
The server verifies
the access token is valid and
returns user-specific results.

https://modelcontextprotocol.io/specification/2025-11-25/basic/authorization

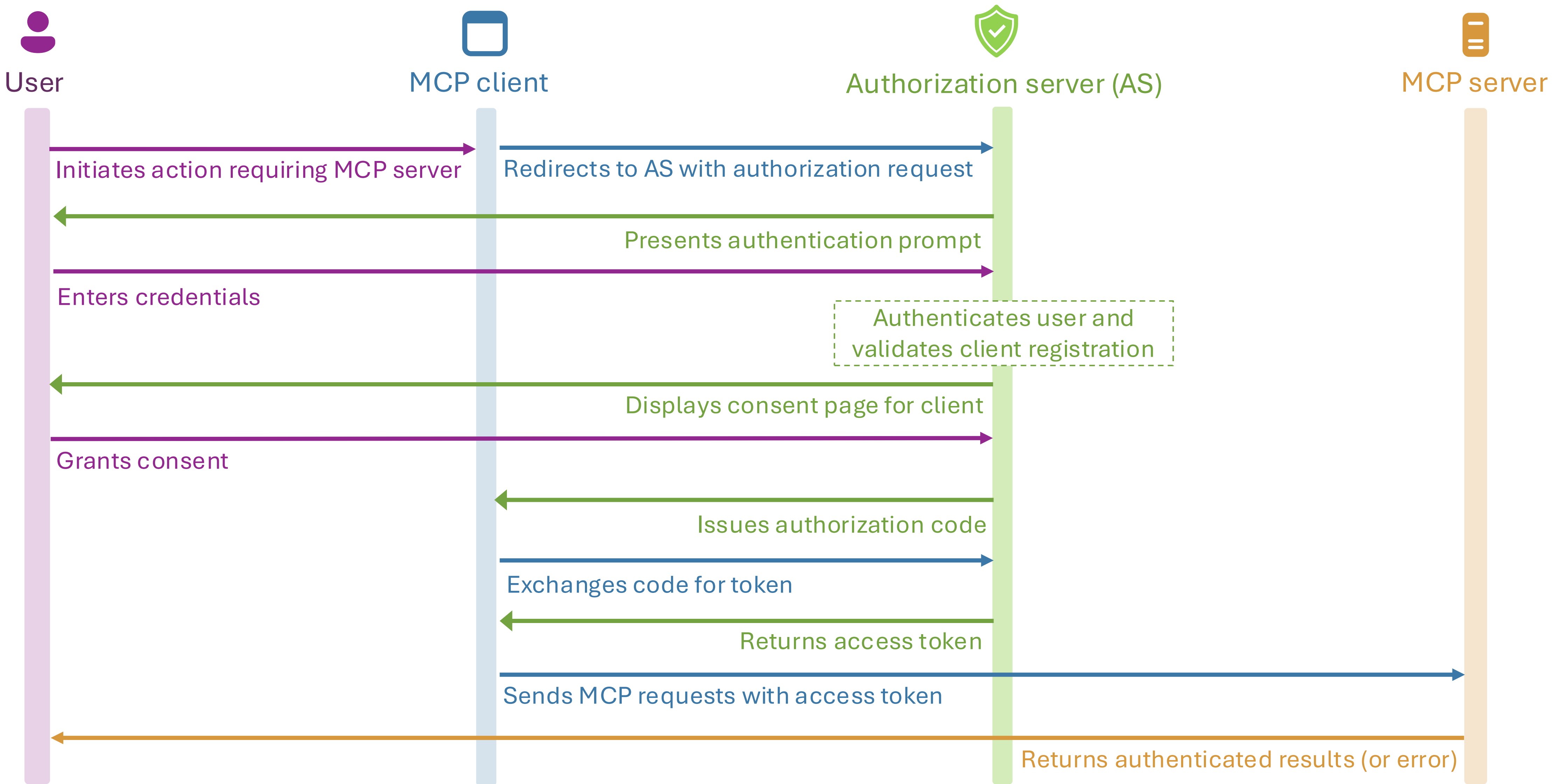
OAuth 2.1 overview

OAuth 2.1 is a standard for allowing resource owners to make authorized requests.

MCP auth is built on top of OAuth 2.1.



OAuth flow for MCP (Simplified)


User
MCP client
Authorization server (AS)
MCP server
Initiates action requiring MCP server
Redirects to AS with authorization request
Presents authentication prompt
Enters credentials
Authenticates user and validates client registration
Displays consent page for client
Grants consent
Issues authorization code
Exchanges code for token
Returns access token
Sends MCP requests with access token
Returns authenticated results (or error)

Authorization server discovery

Before starting the OAuth flow, the **MCP client** first needs to determine the authorization server and required scopes.

The **MCP server** must support:

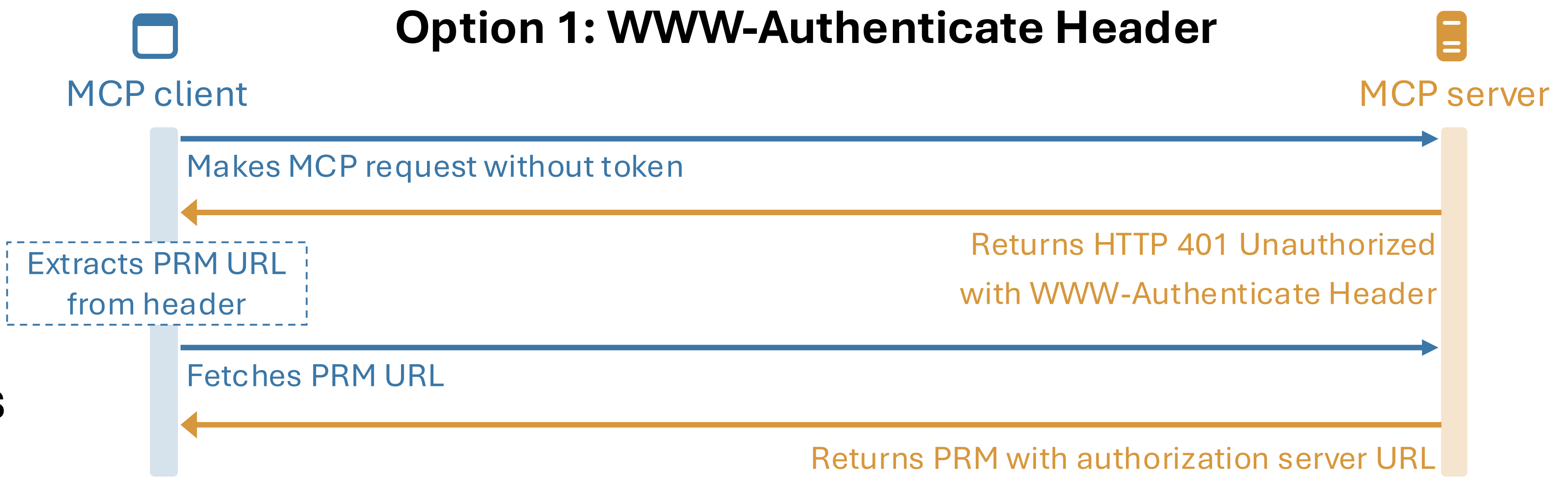
- **Protected Resource Metadata (PRM):** A document that lists the authorization servers and other resource metadata. PRM location is determined via WWW-Authenticate header or well-known PRM URL.

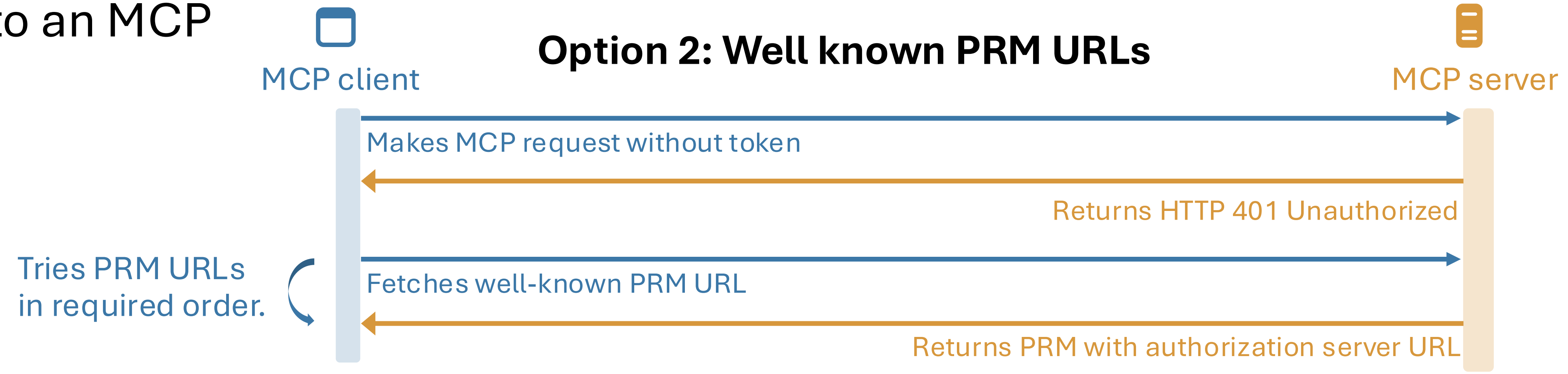
Then the **Authorization server** must support discovery of the exact authorization URLs using either…

- **OAuth 2.0 Authorization Server Metadata**
- **OIDC Discovery 1.0**

PRM flow: Discovering the authorization server

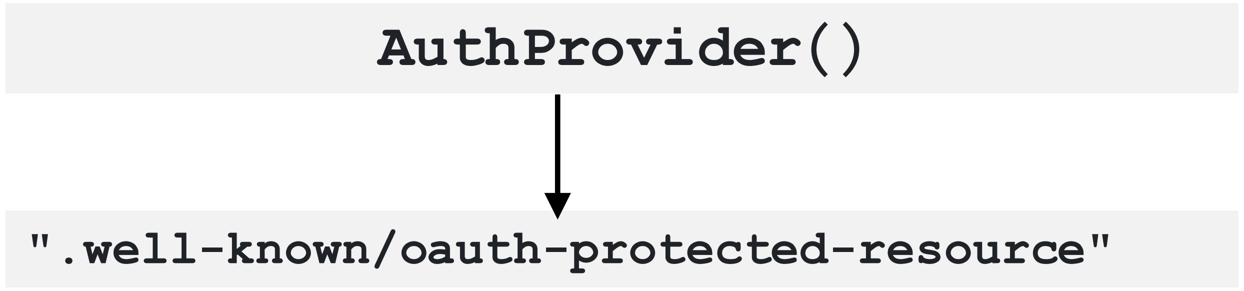
This flow happens on the first unauthenticated request to an MCP server.


Option 1: WWW-Authenticate Header
MCP client
MCP server
Makes MCP request without token
Returns HTTP 401 Unauthorized with WWW-Authenticate Header
Extracts PRM URL from header
Fetches PRM URL
Returns PRM with authorization server URL


Option 2: Well known PRM URLs
MCP client
MCP server
Makes MCP request without token
Returns HTTP 401 Unauthorized
Fetches well-known PRM URL
Returns PRM with authorization server URL
Tries PRM URLs in required order.

Support for PRM in Python FastMCP servers

When you create a FastMCP server with an auth provider, FastMCP automatically adds the PRM routes:



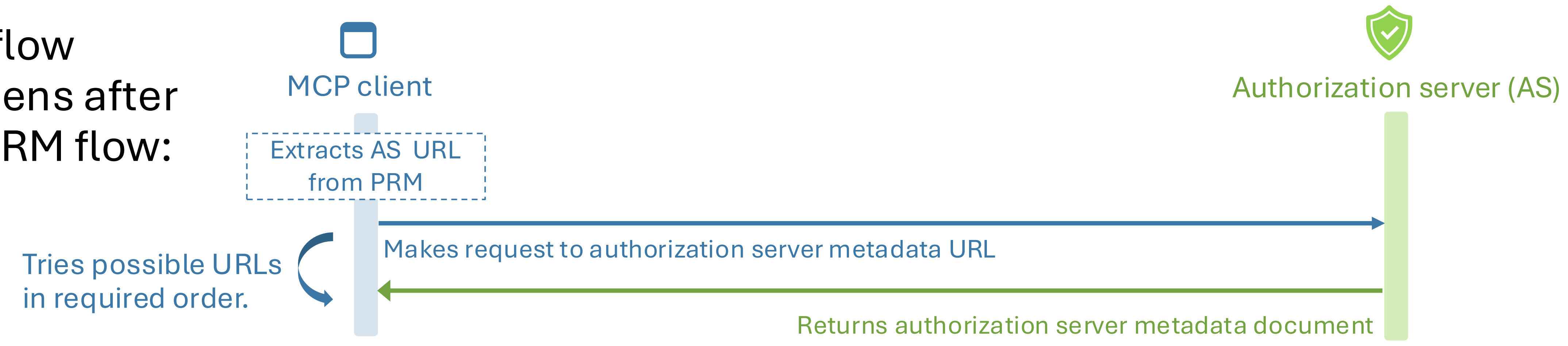
```
AuthProvider()
```

```
".well-known/oauth-protected-resource"
```

If you're writing your own MCP server from scratch, you must implement PRM route yourself.

Authorization server metadata discovery flow

This flow happens after the PRM flow:


MCP client
Authorization server (AS)
Extracts AS URL from PRM
Makes request to authorization server metadata URL
Tries possible URLs in required order.
Returns authorization server metadata document

The metadata URLs depend on whether the authorization URL has a path in it.

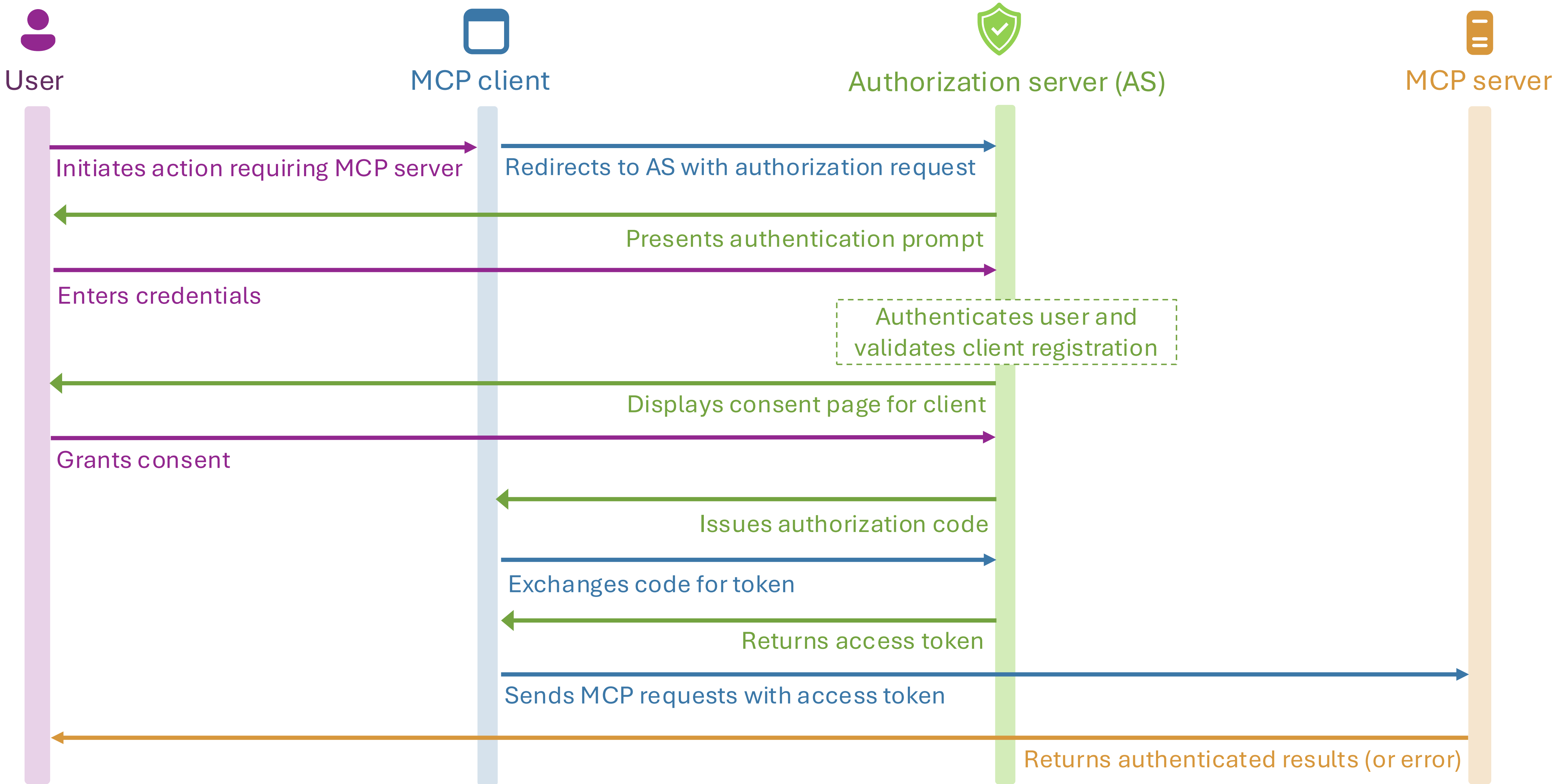
If path:

1. https://AUTHORIZATION-URL.COM/.well-known/oauth-authorization-server/PATH
2. https://AUTHORIZATION-URL.COM/.well-known/openid-configuration/PATH
3. https://AUTHORIZATION-URL.COM/PATH/.well-known/openid-configuration

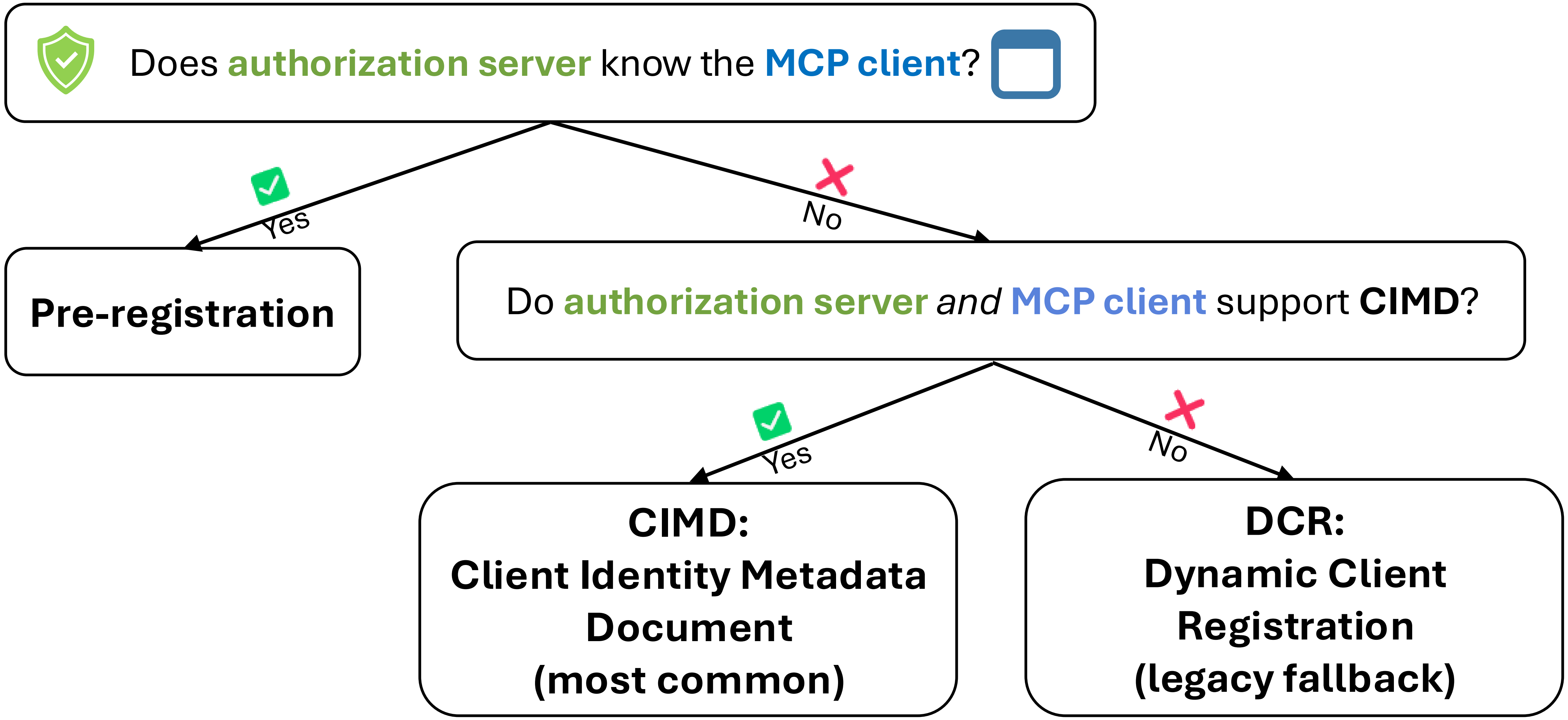
If no path:

1. https://AUTHORIZATION-URL.COM/.well-known/oauth-authorization-server
2. https://AUTHORIZATION-URL.COM/.well-known/openid-configuration

OAuth flow for MCP: Revisited


User
MCP client
Authorization server (AS)
MCP server
Initiates action requiring MCP server
Redirects to AS with authorization request
Presents authentication prompt
Enters credentials
Authenticates user and validates client registration
Displays consent page for client
Grants consent
Issues authorization code
Exchanges code for token
Returns access token
Sends MCP requests with access token
Returns authenticated results (or error)

How does authorization server validate client?


Does authorization server know the MCP client?
Yes
No
Pre-registration
Do authorization server and MCP client support CIMD?
Yes
No
CIMD: Client Identity Metadata Document (most common)
DCR: Dynamic Client Registration (legacy fallback)

Client ID Metadata Document

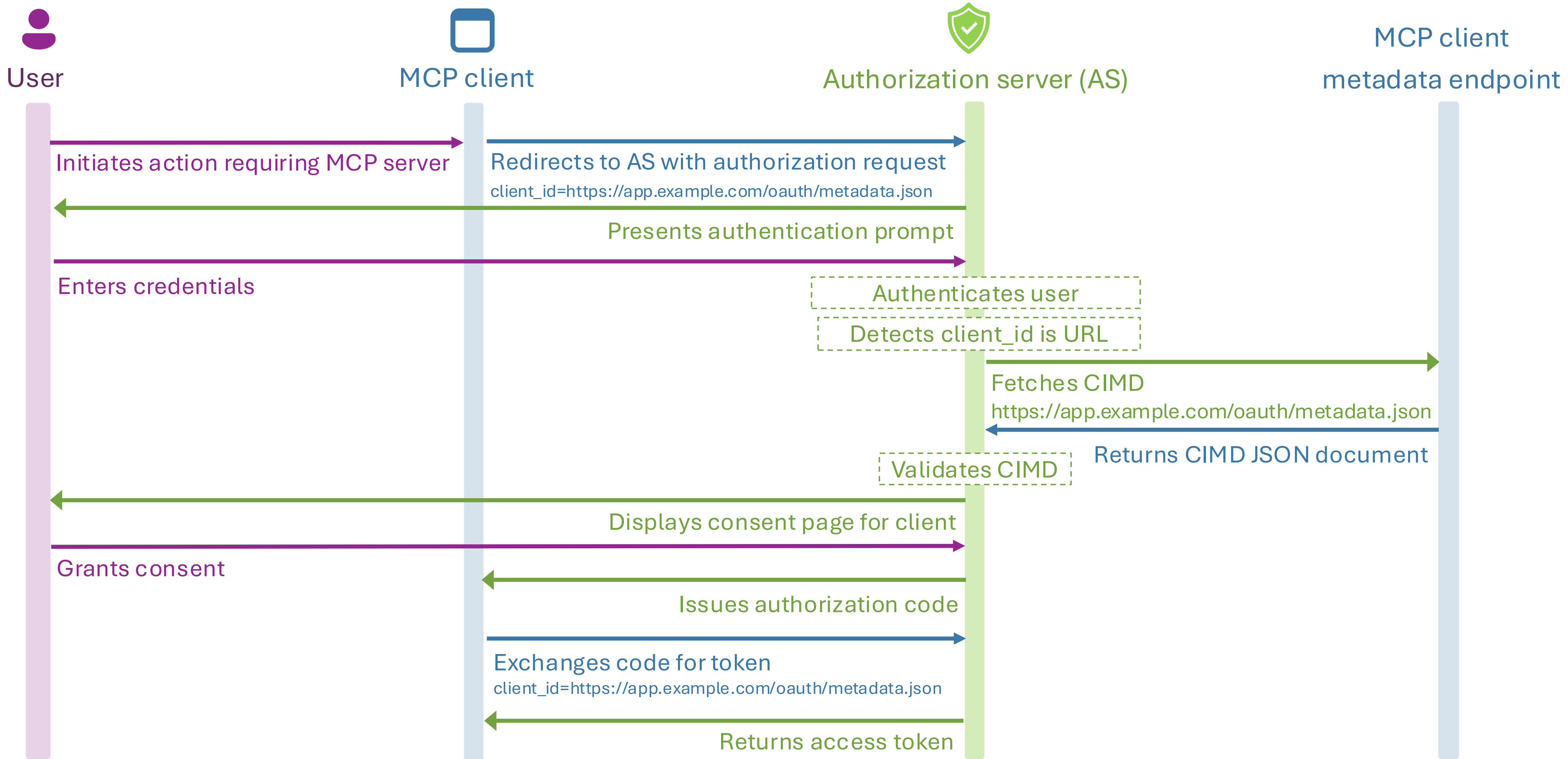
CIMD document format:

```json
{
  "client_id": "https://app.example.com/oauth/client-metadata.json",
  "client_name": "Example MCP Client",
  "client_uri": "https://app.example.com",
  "logo_uri": "https://app.example.com/logo.png",
  "redirect_uris": [
    "http://127.0.0.1:3000/callback",
    "http://localhost:3000/callback"
  ],
  "grant_types": ["authorization_code"],
  "response_types": ["code"],
  "token_endpoint_auth_method": "none"
}
```

VS Code example: https://vscode.dev/oauth/client-metadata.json

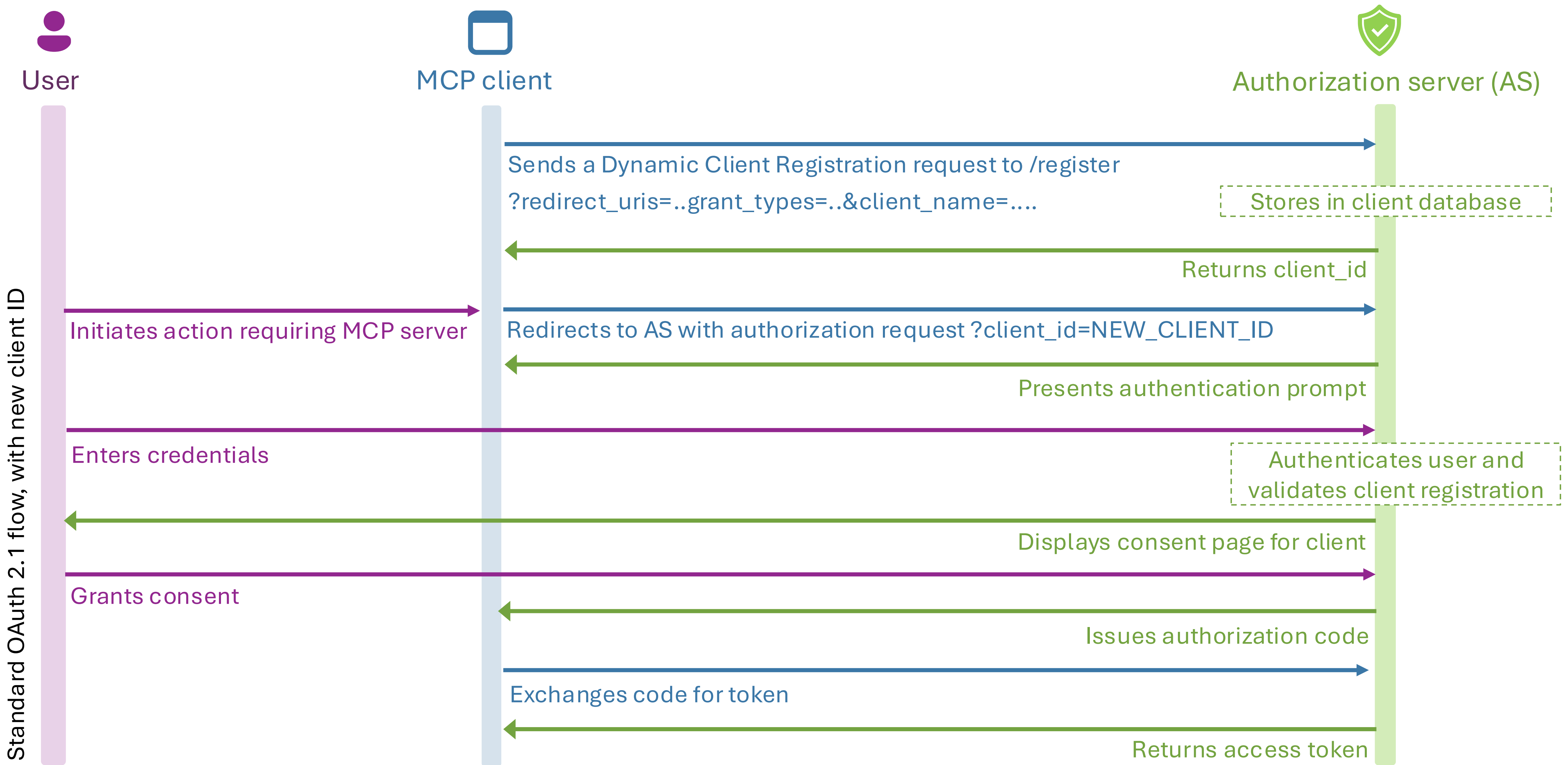
CIMD flow

Assuming CIMD at https://app.example.com/oauth/metadata.json


User
MCP client
Authorization server (AS)
MCP client metadata endpoint
Initiates action requiring MCP server
Redirects to AS with authorization request
client_id=https://app.example.com/oauth/metadata.json
Presents authentication prompt
Enters credentials
Authenticates user
Detects client_id is URL
Fetches CIMD
https://app.example.com/oauth/metadata.json
Returns CIMD JSON document
Validates CIMD
Displays consent page for client
Grants consent
Issues authorization code
Exchanges code for token
client_id=https://app.example.com/oauth/metadata.json
Returns access token

DCR flow

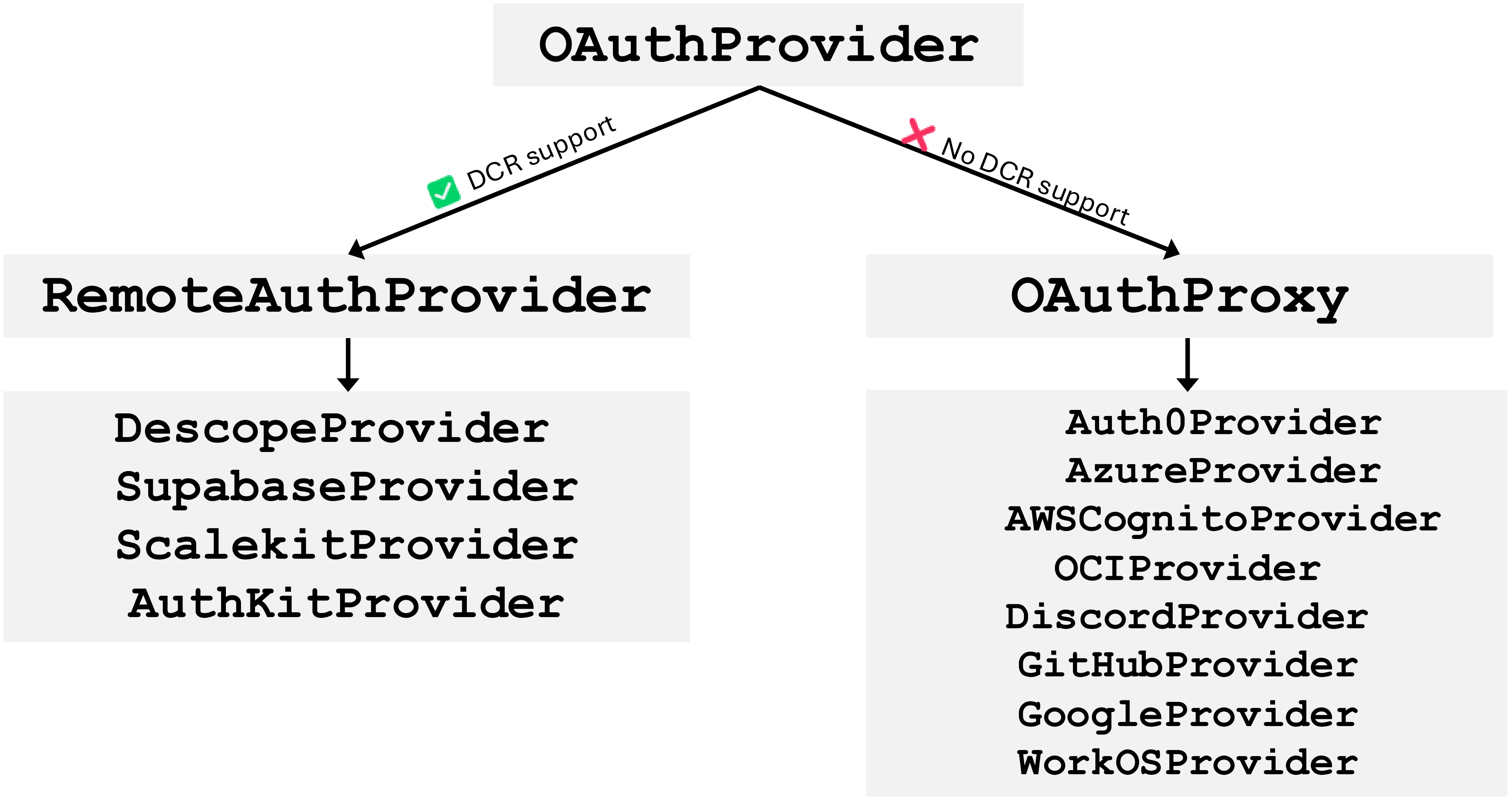
Only the initial client registration step differs.


User
MCP client
Authorization server (AS)
Sends a Dynamic Client Registration request to /register
?redirect_uris=..grant_types=..&client_name=....
Stores in client database
Returns client_id
Initiates action requiring MCP server
Redirects to AS with authorization request ?client_id=NEW_CLIENT_ID
Presents authentication prompt
Enters credentials
Authenticates user and validates client registration
Displays consent page for client
Grants consent
Issues authorization code
Exchanges code for token
Returns access token
Standard OAuth 2.1 flow, with new client ID

Support for the full MCP authorization spec

Authorization provider		AS Metadata Discovery	Client ID Metadata Document	Dynamic Client Registration
Microsoft Entra	Hosted identity server	✅ (OIDC)	❌	❌
KeyCloak	OSS identity server	✅	❌	✅ (some bugs)
Descope	Identity server (+ wrapper of Entra, etc)	✅	✅	✅
WorkOS AuthKit	Identity server (+ wrapper of Entra, etc.)	✅	✅	✅
Okta Auth0	Hosted identity server	✅	✅	✅
ScaleKit	Hosted identity server	✅	✅	✅

Support for OAuth in Python FastMCP servers



Remote OAuth with full DCR support

Using Remote OAuth in Python FastMCP server

For a hosted provider like Scalekit that is fully compliant with MCP auth, FastMCP provides an easy-to-use subclass of RemoteAuthProvider:

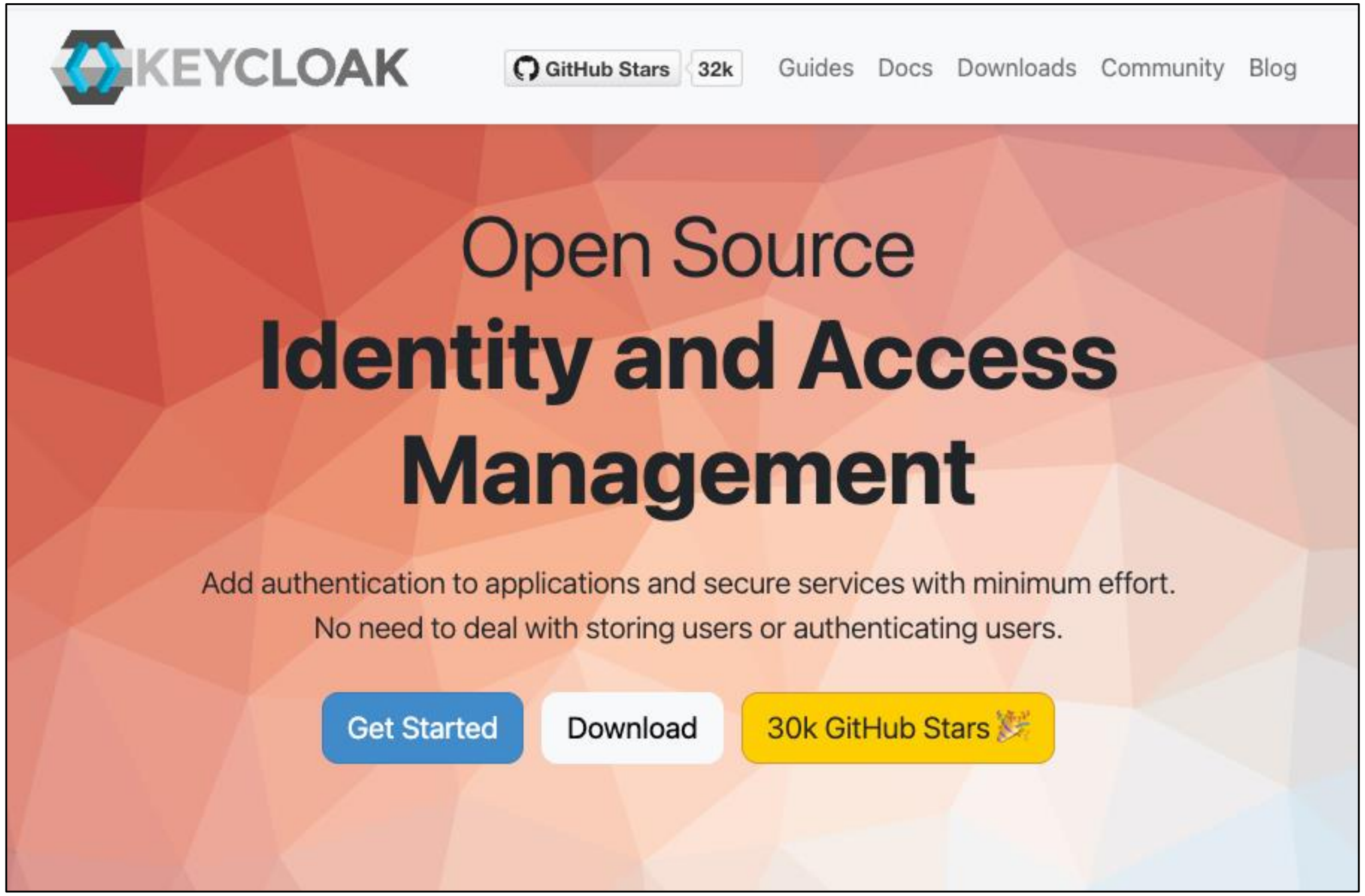
```python
from fastmcp.server.auth.providers.scalekit import ScalekitProvider

auth_provider = ScalekitProvider(
  environment_url=SCALEKIT_ENVIRONMENT_URL,
  resource_id=SCALEKIT_RESOURCE_ID,
  base_url=MCP_SERVER_URL,
  required_scopes=["read"]
)

mcp = FastMCP(name="My MCP server", auth=auth_provider)
```

Come to office hours after after for a demo of ScaleKit!

KeyCloak: Open-source identity server



Open Source
Identity and Access Management

Add authentication to applications and secure services with minimum effort.
No need to deal with storing users or authenticating users.

Keycloak is an OAuth 2.1 compliant identity server that can be deployed via a Docker image.

Keycloak supports DCR *but* has a few open issues.

Integrating Keycloak with FastMCP server

We can use a custom subclass of RemoteAuthProvider that works around the open issues with DCR implementation in Keycloak.

```
from fastmcp.server.auth import RemoteAuthProvider

class KeycloakAuthProvider(RemoteAuthProvider):

def __init__(
  self, *,
  realm_url: AnyHttpUrl | str,
  base_url: AnyHttpUrl | str,
  required_scopes: list[str] | None = None,
  audience: str | list[str] | None = None,
  token_verifier: JWTVerifier | None = None,
):
 ....
```

```
auth = KeycloakAuthProvider(
  realm_url=KEYCLOAK_REALM_URL,
  base_url=keycloak_base_url,
  required_scopes=["openid", "mcp:access"],
  audience=keycloak_audience,
)
```

servers/auth_mcp.py

🔗 aka.ms/python-mcp-demos: servers/keycloak_provider.py

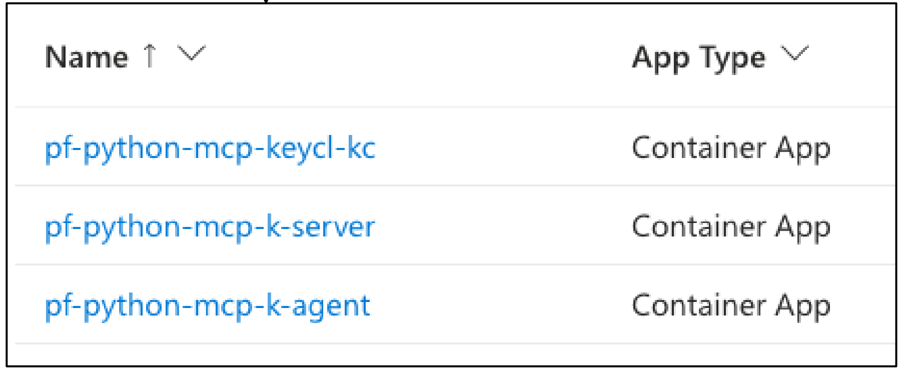
Deploying example server with KeyCloak

1. Open this GitHub repository:

   aka.ms/python-mcp-demos

2. Follow instructions in README for "Deploy to Azure with Keycloak":

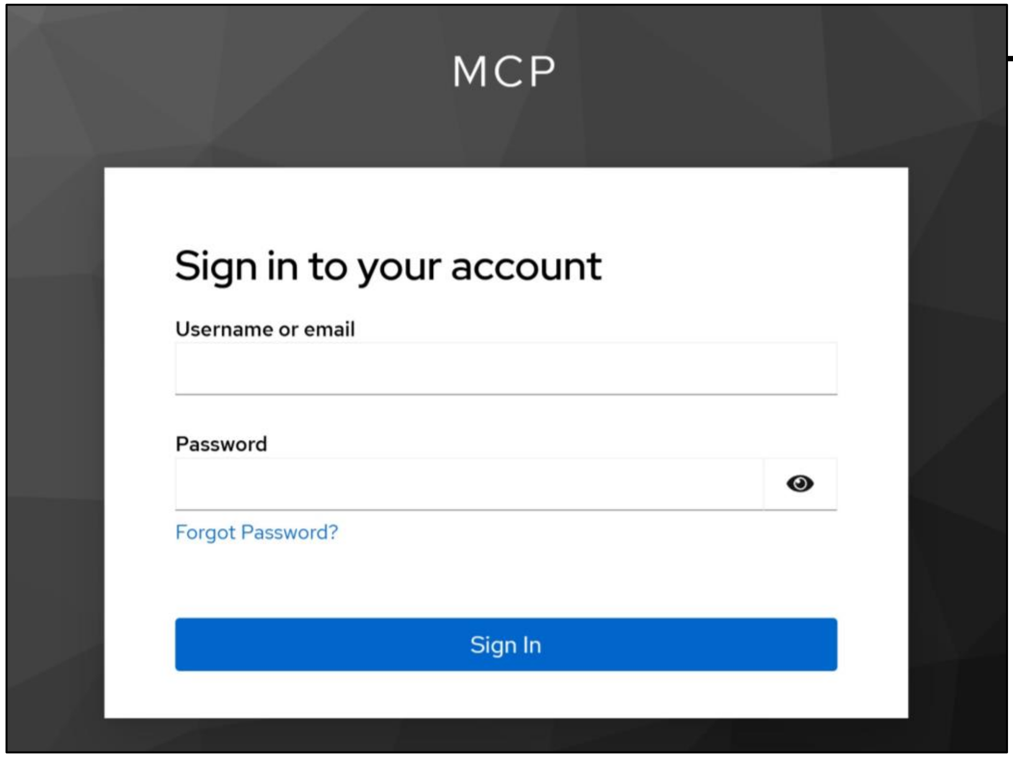
```
>> azd auth login
>> azd env set MCP_AUTH_PROVIDER keycloak
>> azd env set KEYCLOAK_ADMIN_PASSWORD "YourSecurePassword123"
>> azd up
```



Name ↑	App Type
pf-python-mcp-keycl-kc	Container App
pf-python-mcp-k-server	Container App
pf-python-mcp-k-agent	Container App

Demo: Using authenticated server in VS Code

KeyCloak user login screen:



MCP

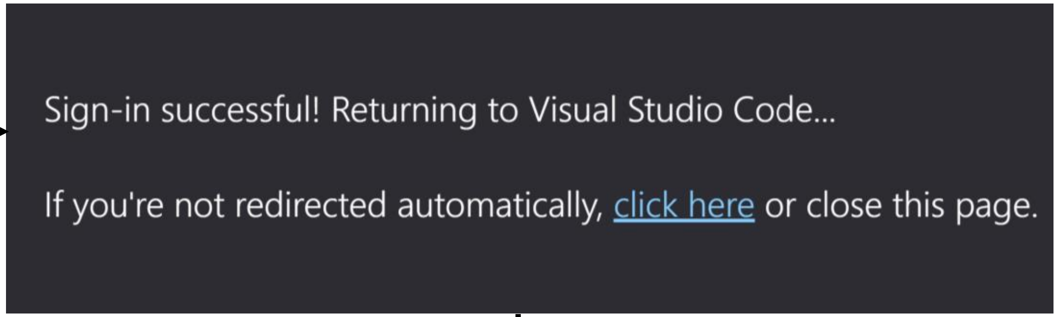
Sign in to your account

Username or email

Password

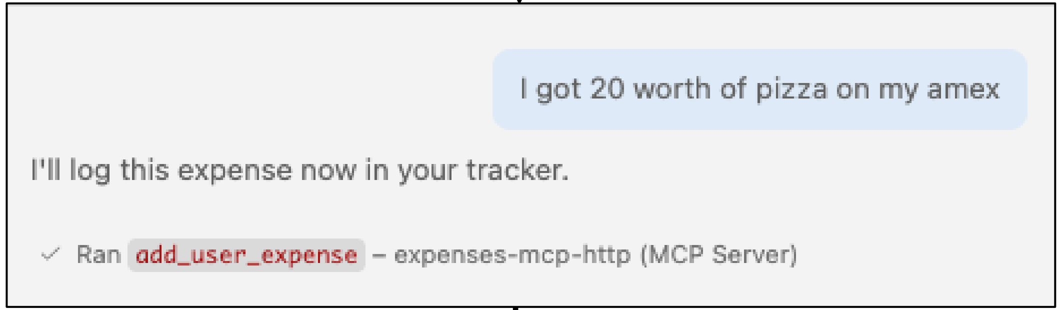
Forgot Password?

Sign In



Sign-in successful! Returning to Visual Studio Code...

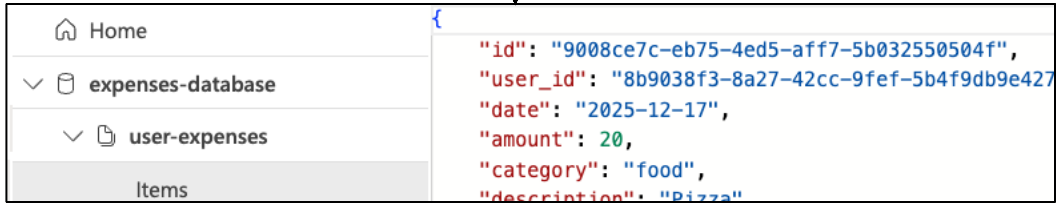
If you're not redirected automatically, click here or close this page.



I got 20 worth of pizza on my amex

I'll log this expense now in your tracker.

✓ Ran `add_user_expense` – expenses-mcp-http (MCP Server)



Home

expenses-database

user-expenses

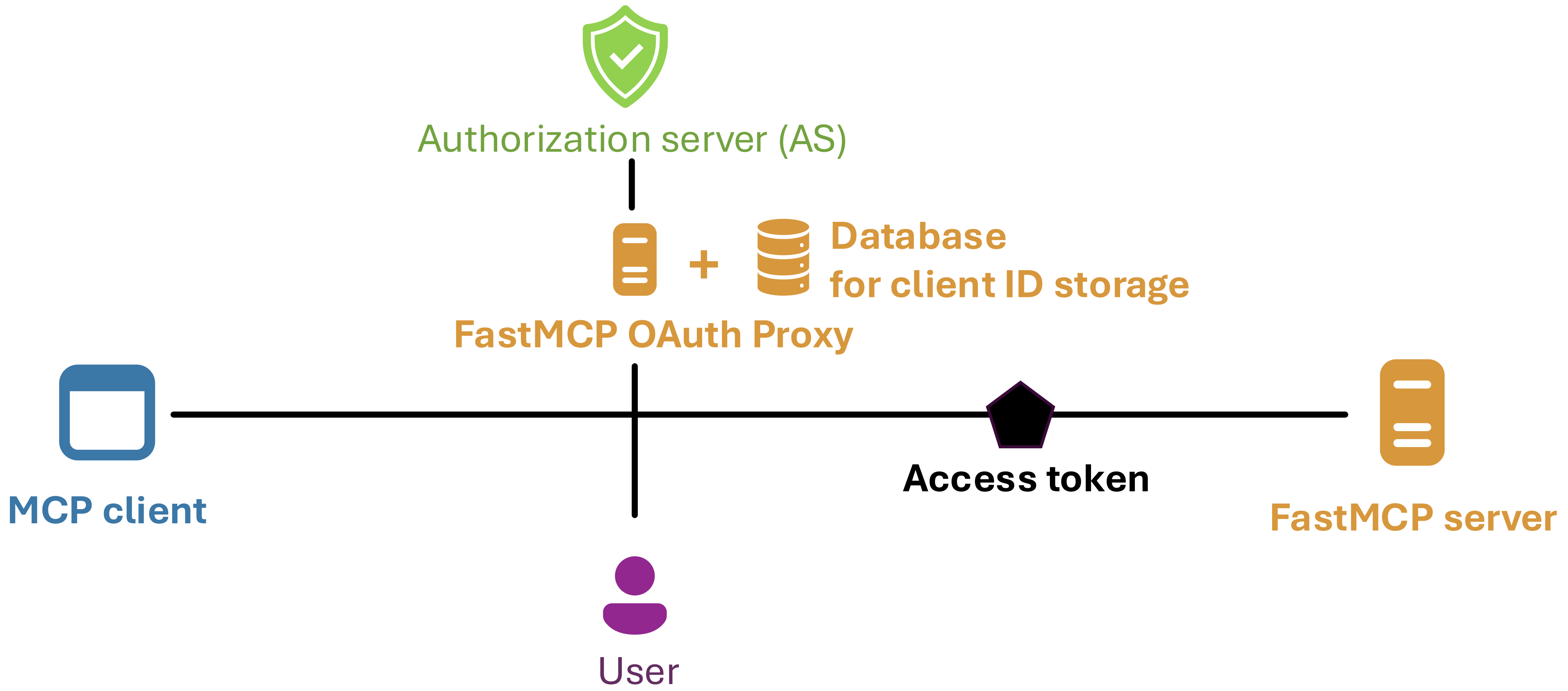
Items

```
{
    "id": "9008ce7c-eb75-4ed5-aff7-5b032550504f",
    "user_id": "8b9038f3-8a27-42cc-9fef-5b4f9db9e427
    "date": "2025-12-17",
    "amount": 20,
    "category": "food",
    "description": "Pizza"
```

Entra via OAuth Proxy

Entra support via OAuth Proxy

To compensate for Entra's lack of DCR support, we can implement with a proxy:



Integrating Entra with FastMCP server

FastMCP provides AzureProvider, a subclass of OAuthProxy that implements DCR:

```
from fastmcp.server.auth.providers.azure import AzureProvider

oauth_container = cosmos_db.get_container_client(os.environ["AUTH_CONTAINER"])
oauth_client_store = CosmosDBStore(container=oauth_container,
                                   default_collection="oauth-clients")

auth = AzureProvider(
  client_id=os.environ["ENTRA_PROXY_AZURE_CLIENT_ID"],
  client_secret=os.environ["ENTRA_PROXY_AZURE_CLIENT_SECRET"],
  tenant_id=os.environ["AZURE_TENANT_ID"],
  base_url=os.environ["ENTRA_PROXY_MCP_SERVER_BASE_URL"],
  required_scopes=["mcp-access"],
  client_storage=oauth_client_store,
)
```

🔗 aka.ms/python-mcp-demos: servers/auth_mcp.py

Deploying example server with Entra Proxy

1. Open this GitHub repository:

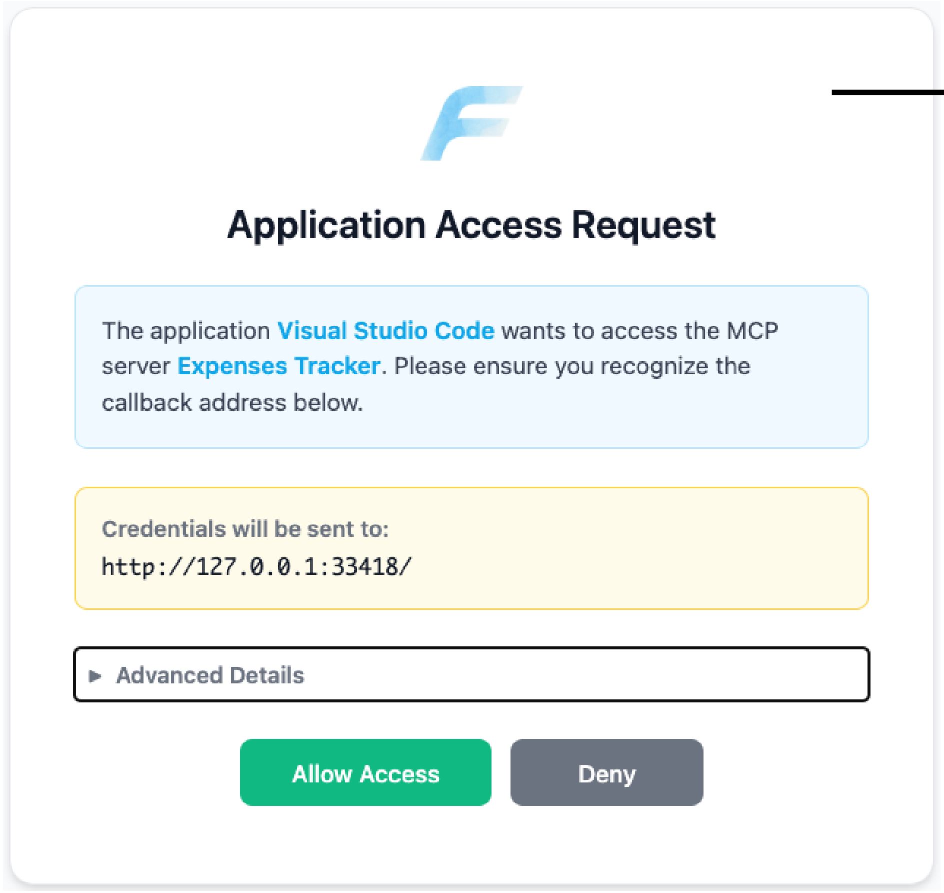
   aka.ms/python-mcp-demos

2. Follow README steps for "Deploy to Azure with Entra OAuth Proxy":

```
>> azd auth login
>> azd env set MCP_AUTH_PROVIDER entra_proxy
>> azd env set AZURE_TENANT_ID your-tenant-id
>> azd up
```

Demo: Using authenticated server in VS Code

FastMCP OAuth proxy screen:

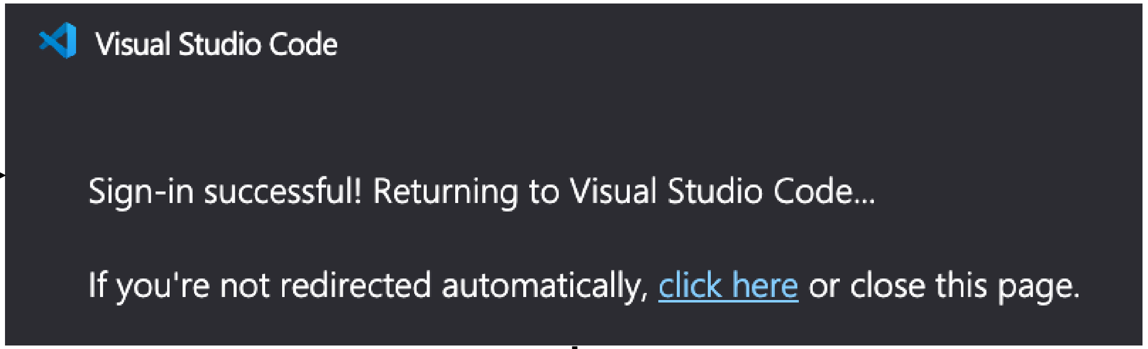


Application Access Request

The application **Visual Studio Code** wants to access the MCP server **Expenses Tracker**. Please ensure you recognize the callback address below.

Credentials will be sent to:
`http://127.0.0.1:33418/`

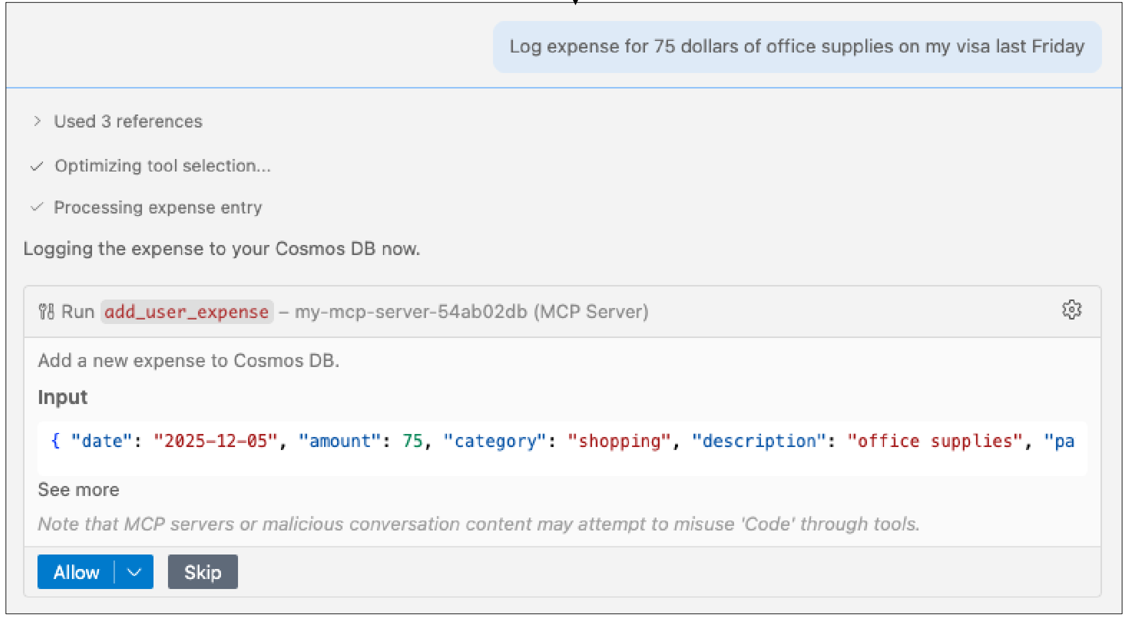
▸ Advanced Details



Visual Studio Code

Sign-in successful! Returning to Visual Studio Code...

If you're not redirected automatically, click here or close this page.



Log expense for 75 dollars of office supplies on my visa last Friday

› Used 3 references
✓ Optimizing tool selection...
✓ Processing expense entry

Logging the expense to your Cosmos DB now.

Run `add_user_expense` – my-mcp-server-54ab02db (MCP Server)

Add a new expense to Cosmos DB.

Input

```
{ "date": "2025-12-05", "amount": 75, "category": "shopping", "description": "office supplies", "pa
```

See more

Note that MCP servers or malicious conversation content may attempt to misuse 'Code' through tools.

Alternative: Only support pre-registered clients

If your MCP server does not need to be usable by arbitrary MCP clients, then you don't need to worry about DCR support.

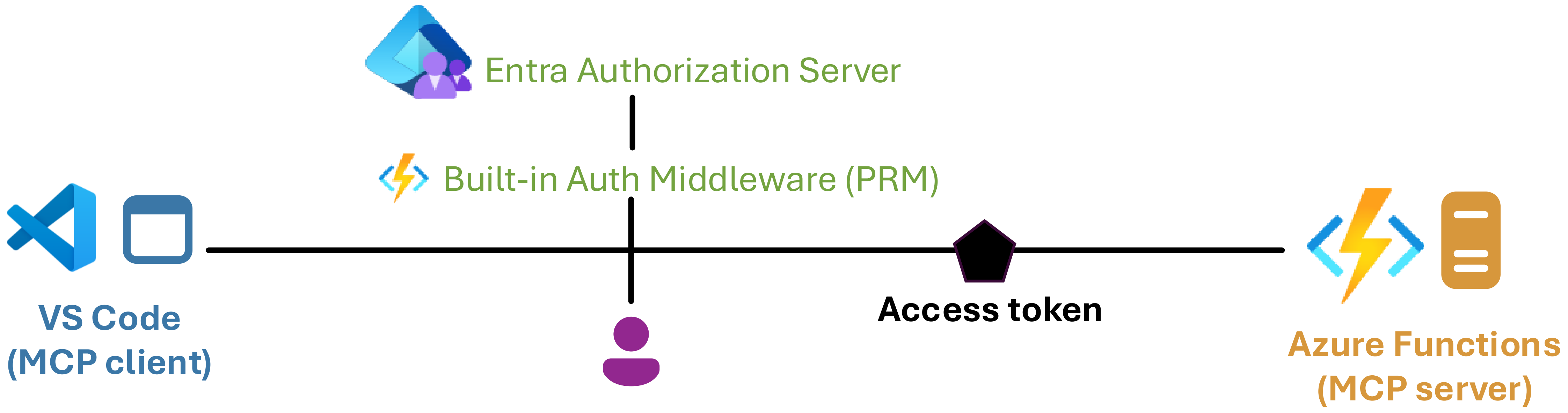
Known client IDs:

- VS Code (**`aebc6443-996d-45c2-90f0-388ff96faa5`**)
- Other Microsoft products
- Your own custom client applications

Deploying Azure Function with Pre-registration

1. Open this GitHub repository:
   github.com/Azure-Samples/mcp-sdk-functions-hosting-python
2. Follow instructions in README for deploying:

```
>> azd env set PRE_AUTHORIZED_CLIENT_IDS aebc6443-996d-45c2-90f0-388ff96faa56
>> azd up
```


Entra Authorization Server
Built-in Auth Middleware (PRM)
VS Code (MCP client)
Access token
Azure Functions (MCP server)

Next steps

Watch past recordings:

aka.ms/pythonmcp/resources

Come to office hours after each session in Discord:

aka.ms/pythonai/oh

Learn from MCP for Beginners:

aka.ms/mcp-for-beginners

🛠️ Dec 16:

Building MCP servers with FastMCP

☁️ Dec 17:

Deploying MCP servers to the cloud

👤 Dec 18:

Authentication for MCP servers