SECUREAUTH
Welcome to Better Identity.

HOT TOPIC

AGENTIC AI

INSIDE THE FIREWALL: SECURING INTERNAL TOOLS

In fully internal environments, agentic AI still requires robust identity. This piece explores how to secure AI access to enterprise tools using advanced OAuth 2.1 patterns—like token exchange, rich authorization requests, and backchannel authentication—without compromising Zero Trust or least privilege principles.

ARCHITECTING IDENTITY FOR AGENTIC AI: PART 3

This is **Part 3** 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 3** of our series on deploying agentic AI systems, we focus on securing AI tool integrations inside the firewall. In an enterprise setting, both the Identity Provider (IdP) and all Model Context Protocol (MCP) servers (the AI-accessible tools and APIs) are hosted internally. This unified domain of control simplifies identity management but also demands robust OAuth 2.1 patterns to ensure each internal tool is accessed safely and with least privilege. Building on the OAuth/OIDC fundamentals and internal IdP integration from Parts 1 and 2, we will now explore advanced OAuth 2.1 techniques for fully internal deployments (no external federation yet). These include delegation with OAuth Token Exchange, fine-grained access with Rich Authorization Requests (RAR), secure token issuance via Pushed Authorization Requests (PAR), and decoupled authentication using Client-Initiated Backchannel Authentication (CIBA). We'll cover both autonomous agent scenarios and human-in-the-loop interactions, illustrating how each pattern applies to AI agents accessing internal MCP tools or APIs.

OAuth 2.1 in a Fully Internal Architecture

In an internal environment, all components—the AI agent (client), IdP, and MCP tool servers—reside within the organization's network. The AI agent can leverage Single Sign-On (SSO) with the internal IdP, and every tool trusts tokens issued by this IdP. Unlike public integrations, there's no need for multi-tenant federation or external identity trust here. However, internal doesn't mean simplistic—strong security and least privilege are still paramount. Modern **OAuth 2.1** practices should be followed to protect internal APIs from misuse or over-broad access. OAuth 2.1 consolidates the best of OAuth 2.0, mandating secure defaults like PKCE and dropping unsafe flows. For example, an AI agent that needs user identity should use the Authorization Code flow with PKCE (never the old implicit grant), even if all traffic stays inside the firewall. With the basics in place, we can introduce more advanced patterns to handle complex agent workflows.

Autonomous vs Human-In-The-Loop: Some AI agents operate autonomously (e.g. performing nightly data aggregation) while others act on direct user requests. In a fully autonomous scenario with no user present, the agent can use a machine identity—typically the **OAuth Client Credentials flow**—to get access tokens as a service. But many agent tasks involve acting on behalf of a user or obtaining user approval for certain actions. In these human-in-the-loop cases, the agent needs to integrate user authentication/authorization into its tool access flow. The following sections cover OAuth patterns addressing both situations:

- **OAuth Token Exchange (On-Behalf-Of):** Delegating access from one internal service to another, propagating user identity.
- **Rich Authorization Requests (RAR):** Requesting fine-grained, context-rich permissions beyond coarse scopes.

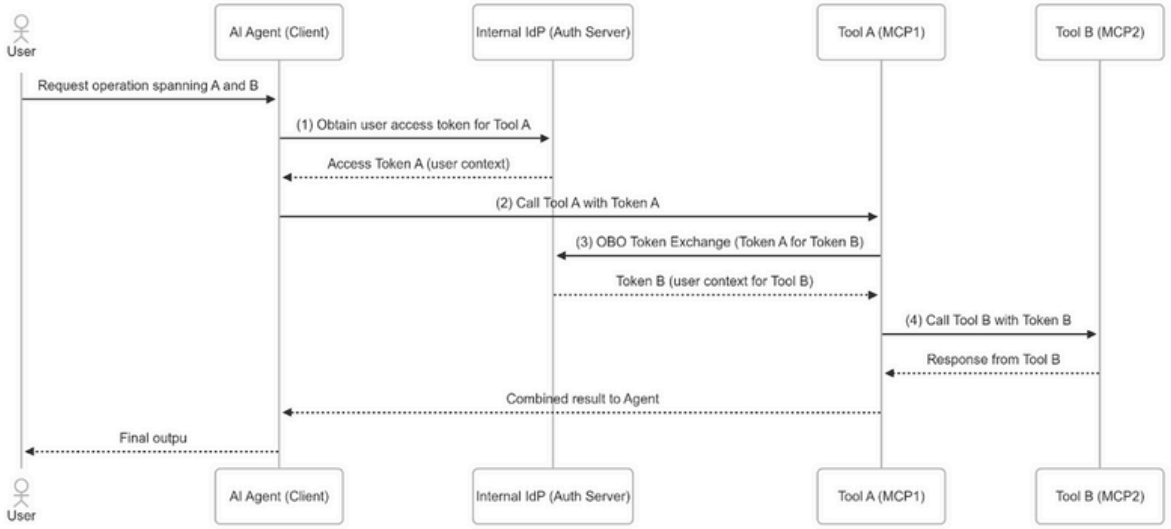
- **Pushed Authorization Requests (PAR):** Securing and simplifying the OAuth authorization flow in high-trust internal channels.
- **Client-Initiated Backchannel Authentication (CIBA):** Decoupling user interaction from the client, enabling out-of-band user auth when needed.

Delegation with OAuth Token Exchange (On-Behalf-Of)

In complex internal systems, an AI agent often needs to call multiple microservices or tools in a chain to fulfill a single high-level request. OAuth Token Exchange, defined in RFC 8693, standardizes a secure delegation flow where one service can exchange an access token for a new token to call another service on behalf of the same user. This is commonly called the On-Behalf-Of (OBO) flow. It allows a resource server (internal API) to act as a client to the IdP, trading the token it received from the AI agent for a new token intended for a downstream API. Crucially, the new token retains the user's identity and context, but can be scoped specifically for the target service.

How OBO Works: Suppose our AI agent has a user's token to call **Tool A**, but fulfilling the request requires data from **Tool B** as well. Instead of the agent directly requesting the user's credentials for **Tool B** (which is impractical), Tool A itself can perform a token exchange. It presents the original user token (now a "subject token") to the IdP's token endpoint with a special grant type (e.g. **urn:ietf:params:oauth:grant-type:token-exchange**), indicating it needs a token for Tool B's audience. The IdP verifies that Tool A is authorized to act on the user's behalf, and issues a new **access token for Tool B**, embedding the user's identity and appropriate scopes. Tool A can then call Tool B with this token, and Tool B will accept it as if the user made the call (with the scope limited to what Tool B requires). All of this happens behind the scenes—the user doesn't have to re-authenticate or even know that multiple services are involved.

Token Exchange Flow (OBO) Sequence: The diagram below illustrates a typical on-behalf-of flow in an internal AI tool chain. The AI Agent (client) calls Tool A with the user's token; Tool A exchanges that token for a Tool B token at the IdP, then calls Tool B and returns the result back to the agent:



In step (1), the agent authenticates the user (e.g. via an OAuth Authorization Code flow) and gets an access token for Tool A's API. Steps (2)-(4) show the on-behalf-of delegation: Tool A uses the Token Exchange grant to get a new token for Tool B using the user token it received. This new token is typically **narrower in scope**, limited to Tool B's needs.

Benefits of Token Exchange: This pattern provides several security and architectural benefits in an internal environment:

End-to-End User Context
Tool B receives a token that represents the user (delegated via Tool A), so it can enforce user-specific authorization and auditing, even though the call came from Tool A. The user identity and permissions propagate through the call chain seamlessly without prompting the user again.

Scoped Delegation
The token issued for Tool B can include claims and scopes specific to Tool B's functions, which might not have been in the original token for Tool A. For example, the original token might allow "read reports" on Tool A, while the exchanged token might allow "read customer data" on Tool B—each token is tailored to its audience.

Isolation of Services
Tool B doesn't need to accept the original token directly (which might be intended for Tool A only). By only honoring tokens issued specifically for itself, Tool B remains isolated and safe from calls that weren't explicitly delegated. In other words, Tool B will only process requests coming through the trusted Tool A (via token exchange), reducing its exposure to misuse.

Simplified Trust Management
Tool B does not have to handle user authentication or session management. It trusts the IdP and Tool A's delegation. As long as the IdP only allows authorized clients (like Tool A) to perform token exchange, Tool B can be confident that any request it receives with a valid token was already approved by the user and the system.

Token Type Transformation
An optional advantage—the token exchange can even change token format if needed. For instance, if Tool B is a legacy system that only accepts SAML assertions, the IdP could issue a SAML token to Tool B in exchange for an OAuth JWT from Tool A. This flexibility helps integrate diverse internal systems without exposing their quirks to the end-user or the AI agent.

Overall, OAuth token exchange (OBO) enables a **secure delegation chain** for agentic AI requests. The AI agent can orchestrate multi-step workflows across internal tools without collecting broad privileges upfront or the user logging into each service. The IdP acts as a central switchboard, issuing service-specific tokens as needed, which upholds the **principle of least privilege** throughout the architecture.

Fine-Grained Access with Rich Authorization Requests (RAR)

While OAuth scopes provide a coarse mechanism to limit what an access token can do, modern applications often need more **fine-grained control** over permissions. This is especially true for AI agents that perform a variety of actions—you may not want to grant an agent a broad scope like **admin** or **read_all** when the user only asked it to perform a specific task. OAuth 2.0 **Rich Authorization Requests (RAR)** (RFC 9396) address this by allowing clients to describe desired access in detail as part of the authorization request. Instead of a simple list of scope strings, the client can send an **authorization_details** JSON object (or an array of objects) conveying exactly what resource or operation it needs access to. The Authorization Server (IdP) can then issue a token precisely tailored to that request.

RAR in a Nutshell: RAR introduces a new request parameter authorization_details carrying a JSON structure. Each object in this structure has a mandatory "type" (defining the kind of access or resource) and can include various fields to specify scope in a rich way. According to RFC 9396, some common fields are:

- **locations** - one or more resource identifiers (e.g. URIs or server endpoints) the client wants to access.
- **actions** **-** the actions or operations the client wants to perform (e.g. **read**, **write**, **delete**).
- **datatypes** **-**the types of data involved or affected (if applicable).
- **identifier** **-** an ID or name for a specific resource instance (e.g. an account number, file ID).
- **privileges** **-** the level or role of access being requested (if applicable).

These fields go beyond what a flat scope string can convey, enabling **context-rich permission requests**. For example, an AI agent could request access to only a certain file or just a specific operation on a service.

Example: Imagine the AI agent needs to retrieve a finance report from an internal file service on behalf of a user. Rather than requesting a broad “file.read” scope on all files, it could send an authorization detail like:

```
{
"type": "file_access",
 "locations": [ "https://files.internal.corp/finance" ],
 "actions": [ "read" ],
 "identifier": "Q4-report.pdf"
}
```

This RAR tells the IdP and user exactly what the agent wants: read access to the file Q4-report.pdf in the finance file repository. Upon seeing this, the IdP can decide (based on policy or user input) whether to allow that specific access. If approved, it will issue an access token that is somehow constrained to that file and action—for instance, the token might include the file ID and an attribute indicating read-only permission.

RAR in the Authorization Flow: From the user's perspective, RAR can make consent more transparent. The authorization UI can display these details (e.g., "This AI assistant is requesting **read** access to **Q4-report.pdf** on the Finance File Server"). This is far clearer than showing a generic scope name. The user can then consent knowing the access is limited. Under the hood, the flow is still an OAuth 2 authorization code grant (or another flow), but with extra data:

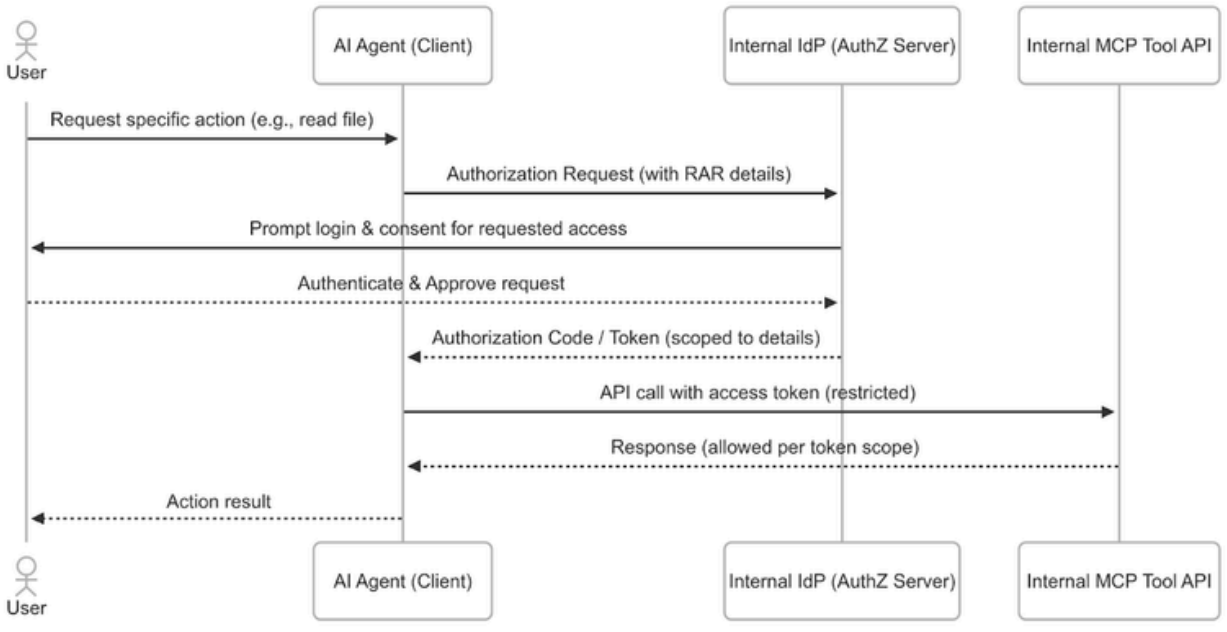
1. Agent Sends RAR Details
The AI agent (OAuth client) initiates an authorization request to the IdP, including the **authorization_details** JSON rather than (or in addition to) the usual **scope** parameter. It may also include all the standard parameters (client ID, redirect URI, PKCE code challenge, etc.).

2. User Authentication & Consent
The IdP authenticates the user (if not already logged in via SSO). It then detects the rich authorization details and displays a consent page listing the requested access in detail. The user reviews and approves or denies the request.

3. Token Issuance
If the user approves, the IdP issues an authorization code (in an Auth Code flow) which the agent exchanges for an access token. The resulting **access token carries the rich authorization**—perhaps as structured claims or as a reference to stored details. The token is limited to exactly what was approved.

4. Accessing the Resource
The AI agent calls the internal API (Tool) with this token. The resource server validates the token and enforces the constraints. For instance, the file service might read the token's claims to ensure the user and file match and the action is "read" on that file.

The sequence diagram below illustrates a simplified flow with RAR in action:



At its core, RAR ensures the **agent's token is as narrowly scoped as possible** for the task at hand. This significantly limits risk if the token is misused or if the agent tries to overreach. It also aligns with the principle of least privilege: even though the internal IdP and tools are under one organization (and could technically issue broad tokens for convenience), using RAR means every request can be constrained to exactly what's needed and nothing more.

Implementing RAR: To use RAR, your internal IdP must support RFC 9396 or a compatible extension. Many modern IdP or IAM systems have added support for RAR or are in the process of doing so, given its importance for fine-grained consent. The resource servers (MCP tool, APIs) also need to understand the authorization details in tokens. In practice, tokens might include the **authorization_details** JSON or a reference/ID to it. For example, a JWT access token could have a claim with the permitted resource and action. Alternatively, the IdP might issue a structured token (like a JWT with an **auth_time** and transaction ID) and keep the details server-side.

RAR vs. Scopes: RAR doesn't replace scopes entirely—it supplements them. You might still use high-level scopes to indicate general categories of access, and use RAR for specifics within those categories. In an internal AI scenario, one approach is to define a scope like tools.access which simply signals the agent is allowed to request tool access, but require that any such request include an **authorization_details** specifying exactly which tool and action. This way, a token without **authorization_details** is essentially useless for any real action on the tools.

By leveraging Rich Authorization Requests, architects can enforce **dynamic, per-request authorization** for AI agents. Each time the agent needs something, it must declare and justify its needs, and the IdP can issue a purpose-bound token. This greatly reduces the chance of an AI agent being hijacked or misbehaving in a way that accesses data it shouldn't, since it can't get blanket credentials easily.

Integration tip: RAR pairs well with the next pattern, **Pushed Authorization Requests (PAR).** Because RAR often involves sending a complex JSON in the auth request (which might not fit cleanly in a URL), PAR is often used to securely deliver that request to the IdP. We'll discuss PAR next.

Securing Authorization Flows with Pushed Authorization Requests (PAR)

OAuth **Pushed Authorization Requests (PAR)** is an extension that makes the authorization flow more secure and reliable, especially in internal or high-security scenarios. In a traditional OAuth flow, the client (AI agent) would redirect the user's browser to the IdP's authorization endpoint with a long URL containing all the request parameters (client ID, scopes, redirect URI, state, etc., and potentially RAR details or other extension parameters). This **front-channel** request can be susceptible to tampering (query params could be modified) and may expose sensitive details in browser logs or redirects. Moreover, if the request is very large (for example, containing a big **authorization_details** JSON), it might hit URL length limits or cause other issues. PAR addresses these concerns by moving the authorization request from the front channel to a **back-channel POST** request.

How PAR Works: Instead of immediately redirecting the user with a full query string, the client first opens a direct connection to the IdP's **PAR endpoint** (a special endpoint defined by RFC 9126). It sends the authorization request parameters in an **HTTP POST** message to the IdP, which is a **secure server-to-server call** (authenticated with the client's credentials). This call includes everything the client would normally put in the redirect (client ID, redirect URI, scopes, PKCE code challenge, state, and even rich auth details like RAR). The IdP responds with a **request_uri**, which is essentially a short-lived handle (reference) to the stored request data.

After this "pushed request" step, the client then initiates the user's browser interaction by redirecting to the IdP's normal authorization URL, but instead of all the parameters, it only needs to include **client_id** and the **request_uri** value. For example: **https://idp.internal.corp/authorize?client_id=AI-Agent-123&request_uri=urn:ietf:params:oauth:request_uri:TX123....**

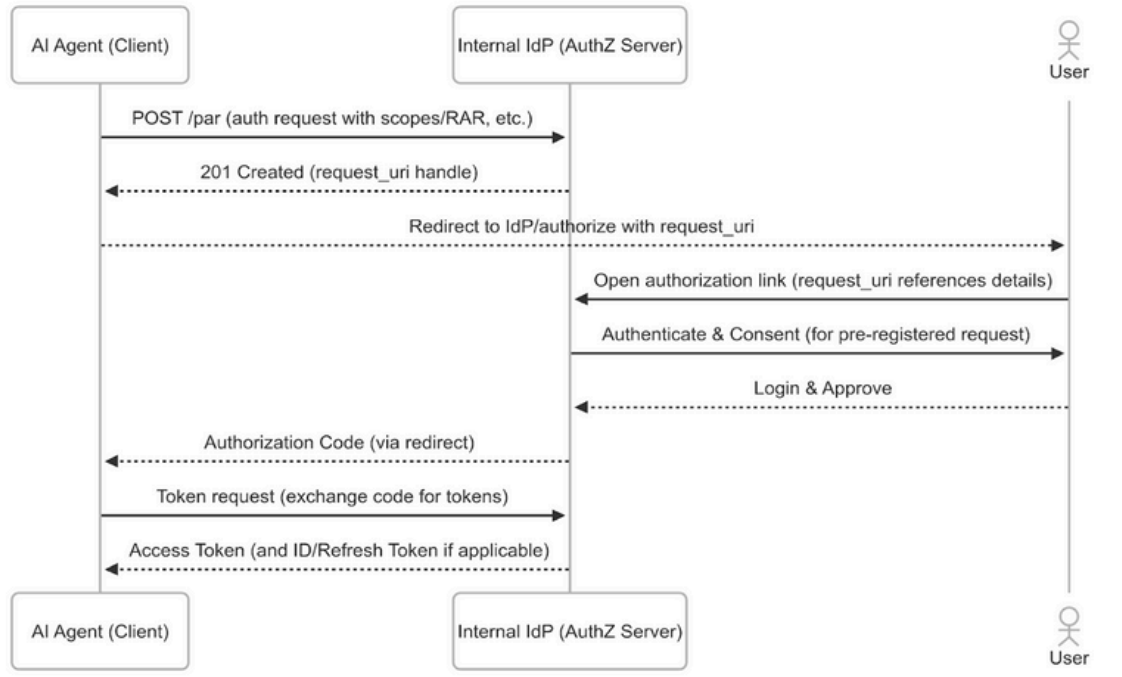
Because the IdP already has the full details from the prior PAR call (identified by that request URI), it will proceed with authentication and consent as usual, knowing the request is exactly what the client intended (and wasn't altered in transit).

PAR Benefits in Internal Tools:

- **Integrity and Confidentiality:** The authorization request details are communicated over a direct back-channel connection secured by TLS and client authentication, rather than through the user's browser. This prevents attackers from manipulating the request parameters or extracting sensitive information from the URL. For instance, if the AI agent includes a large RAR JSON with specific resource IDs, those details won't leak via browser history or logs—they stay between the agent and IdP.

- **Supports Complex Requests:** PAR enables the use of long or complex parameters (like RAR, JWT request objects, or additional OpenID Connect parameters) without worrying about URL length or encoding issues. The IdP can accept much larger payloads via POST. This is one reason PAR is recommended in **financial and healthcare contexts**, where authorization requests often carry detailed data and higher security is required.
- **Pre-Validated Requests:** Since the IdP can immediately validate the request in the PAR call (checking client auth, redirect URI, etc.), it can reject malformed or unauthorized requests upfront. The user will never see a prompt for a request that's invalid—an error would be returned to the client at the PAR step. This leads to a smoother and safer user experience.
- **Simplified Front-Channel:** The final user-facing step is simplified to just referencing the request. Even if an attacker somehow intercepts or alters the **request_uri**, it's useless without the original pushed request (and the IdP will detect any mismatch of **client_id**). Also, the **request_uri** is one-time use and short-lived.

PAR Flow with an AI Agent: Let's illustrate how an AI agent would use PAR (possibly combined with RAR) in an internal OAuth flow:



In the above sequence, the critical difference is the PAR step at the beginning. The rest (user login, consent, token exchange) follows the standard OAuth 2.1 Authorization Code flow. The user is not aware that PAR was used—they just see the normal login and consent screens. But behind the scenes, PAR ensured the AI agent's exact request (say, "access specific file with read permission" as in the RAR example) was securely lodged with the IdP in advance. This eliminates certain attacks; for example, without PAR an attacker could attempt to intercept the redirect and change the **redirect_uri** or scopes, but with PAR, the IdP will only honor the exact request it received directly from the legitimate client.

When to Use PAR: In a fully internal environment, one might argue the risks of intercepting redirects are lower than on the open internet. After all, the user's browser and the IdP might both be on the corporate network. However, internal doesn't equal invulnerable. Employees could have malicious browser extensions or there could be misconfigured proxies. Moreover, internal IdPs often implement **higher security policies** (following standards like Financial-grade API (FAPI) security profiles) to prepare for future external integration. PAR is a straightforward way to harden the auth flow. It's especially useful if your AI agent runs in a context without a traditional web browser—for instance, a command-line tool or a desktop app. In such cases, constructing a complex authorization URL and sending a user to it can be error-prone; using PAR simplifies the client implementation (the agent just posts JSON to the IdP).

Many IdPs now support PAR, and some even require it when using RAR or other advanced claims. To implement PAR in your system, ensure your Authorization Server's software supports RFC 9126. The client (AI agent application) will need the capability to make an HTTP POST to the PAR endpoint and handle the response. Note that the client must authenticate to the PAR endpoint (with its client credentials or other method)—this prevents arbitrary actors from pushing fake requests. Once the **request_uri** is obtained, the rest of the flow uses that URI. The **request_uri** typically expires quickly (e.g. in 5 minutes) and can only be used once.

By using PAR, our AI agent's internal OAuth flows gain **security comparable to external flows** in highly regulated industries. This ensures that as we scale up trust and perhaps later introduce external parties, our internal house is already in order.

Decoupled Authentication with Client-Initiated Backchannel Authentication (CIBA)

The last pattern we'll explore is OpenID Connect **Client-Initiated Backchannel Authentication (CIBA)**. CIBA is an **OAuth/OIDC flow for scenarios where the typical front-channel redirect for user login is not possible or desired.** It enables a client (the AI agent) to trigger an authentication process without directly involving the user's interaction on the client device. Instead, the user completes authentication on a separate device or via another out-of-band mechanism. CIBA is ideal for **decoupled** or asynchronous user consent scenarios—think of it as "login by push notification."

In an internal tool context, when might CIBA be useful? Consider a few examples:

- **Hands-Free or Unattended Scenarios:** The user initiates an action through a voice interface or a phone call with an AI agent. The agent needs to verify the user's identity and get approval to fetch sensitive data. With CIBA, the agent can send an authentication request that prompts the user on their smartphone (or other registered device) to confirm their identity, without the user having to navigate a login page on the voice device.

- **Secure Approval for Sensitive Actions:** An AI assistant might autonomously attempt an action (like a large fund transfer or deleting critical data) but policy requires explicit user approval. Using CIBA, the system can send a push notification to the user's device asking, "Do you authorize this action?" The user's response will determine if the agent gets a token to proceed.
- **Devices with Limited UI:** Internal IoT or kiosk devices might not have full browsers. If an AI agent on such a device needs a user to log in, CIBA allows the user to use another device (like scanning a QR code or responding on their phone) to authenticate and grant access.

How CIBA Works: There are several modes in CIBA (poll, ping, push), but the core concept is consistent. Here's a step-by-step outline (assuming the **ping/push** mode for brevity):

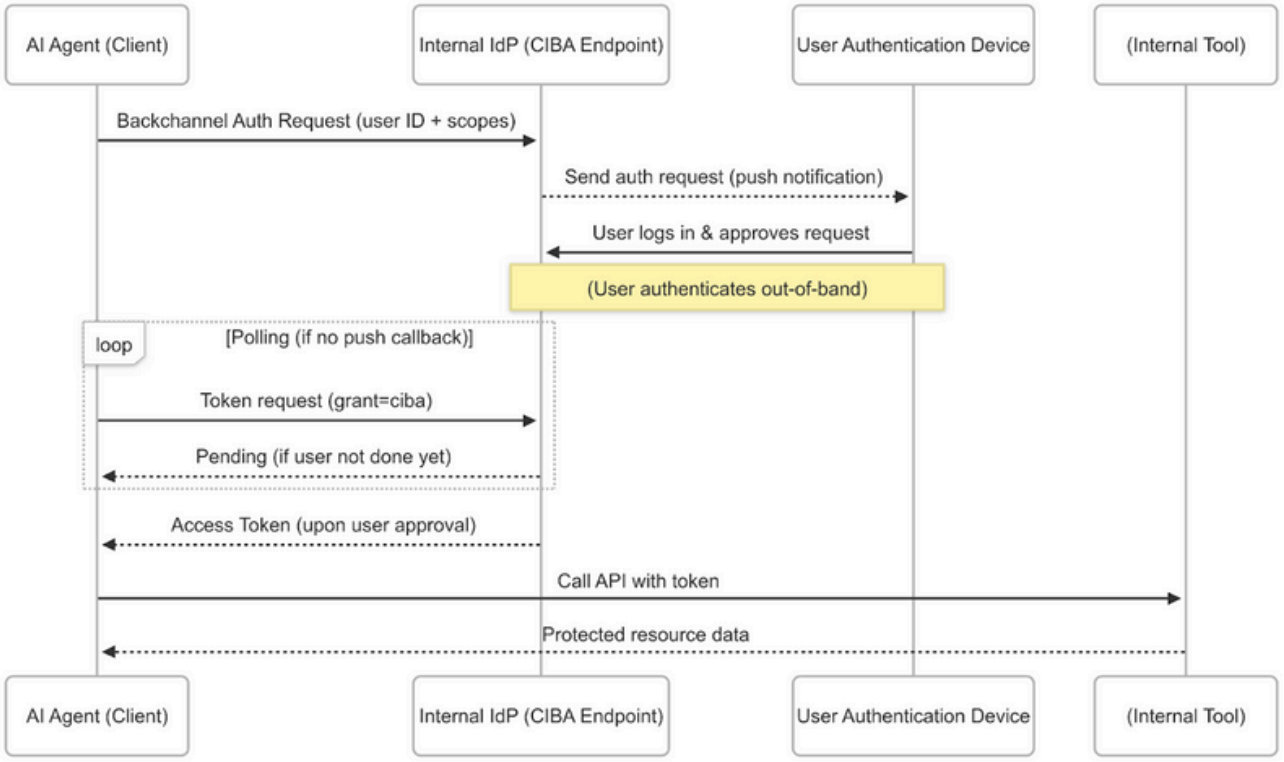
1. Backchannel Auth Request
The AI agent (client) sends a request directly to the IdP's Backchannel Authentication Endpoint. This request includes the client's credentials (to authenticate itself), the scopes or resources it wants (just like an OAuth auth request), and an identifier for the user. Since there is no immediate user interaction, the client must indicate which user it wants to authenticate—this could be a user ID, username, email, phone number, or a pseudonymous identifier (CIBA defines parameters like **login_hint**, **id_token_hint**, or **login_hint_token** for this purpose). The request may also include an indicator of what kind of interaction the user will get (like a hint to how to reach the user's device).

2. User Notification
Once the IdP validates the request, it **initiates an authentication with the user via a separate channel.** Typically, the IdP's system will send a push notification to the user's registered device (for example, an authenticator mobile app or an SMS with a code, depending on implementation). The user's device is often called the **Authentication Device**, in contrast to the client which is the **Consumption Device**. The user might see a message like "AI Agent XYZ is requesting access to Resource ABC. Do you approve?" The user then logs in (if not already) on that device and approves the request, possibly with MFA as required.

3. Token Delivery
Meanwhile, the AI agent is waiting for the outcome. In **poll** mode, the agent would periodically call the IdP's token endpoint (with a grant type like **urn:openid:params:grant-type:ciba**) to check if the user completed auth. In **ping** mode, the IdP will call a pre-registered endpoint on the client to notify it that the user is done. In **push** mode (or one flavor of ping), the IdP directly sends the tokens to the client's backend. Either way, once the user successfully authenticates and consents on their device, the IdP issues an **access token (and possibly an ID token)** to the client. If the user denies or a timeout occurs, the client is informed of the failure instead.

4. Agent Proceeds with Token
Now the AI agent has an access token representing the user (and the scopes originally requested). It can use this token to call the internal MCP tool or API just as it would in a normal OAuth flow.

The sequence below demonstrates a CIBA flow in an internal setting (using poll mode for clarity):



In the above, the agent's request triggers a push to the user's device. The user authenticates and approves. The agent polls until it gets the token (the polling loop would break once the token is ready). In a real ping mode, instead of polling, the IdP would hit the agent's notification endpoint to tell it to fetch the token or include it directly, but the end result is the same.

Notable Points about CIBA:

- The **user experience** is decoupled: The user is interacting with the IdP on their phone (for example), not with the AI agent's interface. The AI agent's interface might simply show "Waiting for confirmation..." until it gets the token. This fits scenarios where the user cannot or should not be redirected in a browser from the agent's context.
- The client must have a way to **identify the user** to the IdP. This typically means the AI agent already knows who the user is (e.g., the user provided an email or username when starting the interaction, or scanned a QR code that encodes a user ID). Privacy and security are important here: the agent shouldn't be able to trigger random people's devices. Often, the user will explicitly say something like "Yes, use my phone for authentication" and perhaps provide an identifier.
- CIBA is an **OIDC (OpenID Connect) specification**, so it often yields both an ID Token and Access Token. It's like an Authorization Code flow, except the "authorization code + redirect" part is replaced by this backchannel negotiation. One quirk: CIBA typically doesn't create a long-lived OAuth consent grant the way an interactive flow might. Each CIBA request stands alone (which can be good for one-time approvals).

- **User Consent and Security:** Because the user isn't on the client device, the consent prompt on the authentication device needs to convey what's being requested. In internal use, this might be simplified (the user likely trusts the agent highly if using such a flow). However, it's still wise to show details (like "AI Finance Agent requesting access to account 1234 data"). The IdP can leverage RAR here too—indeed, CIBA can carry an **authorization_details** in its request, enabling rich description of what the user is approving, even though it's out-of-band.
- CIBA is often used in combination with **strong customer authentication** methods (like biometric app approval) to ensure the person approving is truly the user. This can significantly increase security for sensitive tool operations triggered by AI.

Use in Internal AI Systems: Implementing CIBA requires that your IdP (or Access Management system) supports the CIBA extension (OIDC Decoupled Authentication). Many commercial IAM products have support or partial support for CIBA given its importance in banking (it's part of the OpenID Foundation's Financial-Grade API standards). Within the enterprise, deploying CIBA means registering the AI agent as a CIBA client, often including a **notification endpoint** and/or configuring how the agent will poll. The users need to have an "authentication device" registered—typically this is just the phone they use for MFA in the company (so leveraging existing infrastructure like push MFA apps or SSO mobile apps).

For our AI agent, CIBA offers a powerful way to involve the human user in the loop when it really matters, without forcing the human to be at the same terminal or UI as the agent. It's the ultimate fallback for trust—if the AI wants to do something and isn't sure if it should (or policy says it must ask), it can fire off a CIBA request to get explicit human approval. Internally, this means even headless or autonomous AI processes can still require a human's go-ahead for critical actions, using standard OAuth mechanisms rather than custom ad-hoc confirmations.

Conclusion

In this article, we examined key OAuth 2.1 patterns that help secure internal tool access for AI agents. Within a fully internal environment—where all players (IdP, agents, and tools) are under enterprise control—these techniques enable fine-tuned authentication and authorization:

- **Token Exchange (OBO):** Allows microservices or tool APIs to seamlessly call each other on behalf of a user, ensuring user identity and permissions carry through without over-privileging any single component.
- **Rich Authorization Requests (RAR):** Lets the AI agent request precisely scoped permissions, using structured JSON to go beyond simplistic scope strings. This means tokens can be narrowly tailored to each task, greatly limiting potential damage from misuse.

- **Pushed Authorization Requests (PAR):** Strengthens the OAuth flow by moving sensitive request details to a backchannel. This is especially useful in our internal context to prevent tampering and to handle large, complex authorization details (like those from RAR).
- **Client-Initiated Backchannel Authentication (CIBA):** Provides a way to authenticate users and get consent when direct interaction with the client app isn't feasible. It's a critical tool for human-in-the-loop control, enabling out-of-band approvals for AI actions.

Together, these patterns help software architects implement a **defense-in-depth strategy** for internal agentic AI systems. Each tool access is authenticated with modern OAuth protocols and authorized with the minimum necessary scope and explicit user involvement when needed. By adhering to OAuth 2.1 standards now, your internal AI integrations are not only secure for today's closed-world deployments, but also laying the groundwork for future expansions—such as federation with external IdPs or third-party tool integrations—which will be tackled in subsequent parts of this series. The result is an AI ecosystem that is both **powerful** (capable of autonomous multi-step operations) and **safe** (every step is governed by robust identity and access controls).

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