Identity for AI

April 7, 2026

Table of Contents

Identity for AI

Identity for AI

Artificial intelligence (AI) no longer just supports human actions: it performs them. As AI evolves from predictive models to autonomous agents, a new class of identity challenge is emerging: Identity for AI.

What's new

Follow the latest Identity for AI developments

Get started

Start your Identity for AI journey

AI agents

Learn about AI agents

Agent identity

Classify and secure AI agents

Tutorial

Authorize an AI agent with Advanced Identity Cloud

Glossary

Learn key terms

What's New?

Keep up with new developments in Identity for AI.

Subscribe to get automatic updates: Identity for AI RSS feed

March 2026

March 31

Identity for AI general availability

We're excited to announce the general availability of Identity for AI, which extends Ping Identity's proven identity control plane to agentic architectures.

Identity for AI treats AI agents as first-class, non-human identities. This empowers your organization to safely deploy agents by ensuring they act with delegated authority, enforce least-privilege access to resources, and maintain human accountability through comprehensive auditing and human-in-the-loop (HITL) approvals.

Use the following new capabilities in Ping Identity platforms to secure AI behavior across your systems.

Agent identity

New PingOne PingOne Advanced Identity Cloud PingAM

Onboard and authorize your AI agents, enabling them to authenticate and request scoped access to enterprise tools and APIs.

- **First-class agent identity**: Securely register, update, and disable AI agents with a new dedicated admin experience. This includes assigning agent owners and modeling which applications, groups, and users an agent is authorized to interact with.
- **Delegation instead of impersonation**: OAuth 2.0 token exchange allows an agent to exchange a human user's subject token for a new, downscoped token. The delegation token securely passes the identity of the human subject alongside the identity of the agent using the `act` (actor) and `may_act` claims, maintaining a secure chain of custody for downstream authorization.
- **Least-privilege access at runtime**: Limit an agent's blast radius with fine-grained control over exactly which APIs and data sources the agent can access, including HITL approvals for sensitive actions.

Agent identity is made available as part of our new Identity for AI solution. Contact your account executive to find out more.

Learn more:

- AI agents in PingOne Advanced Identity Cloud
- Managing AI agents in PingOne Advanced Identity Cloud
- AI agents in PingOne
- Managing AI Agents in PingOne
- Configuring OAuth 2.0 token exchange in PingOne
- Configuring a CIBA flow in PingOne

Agent gateway

New PingGateway

PingGateway now provides runtime security for the backend resources and MCP servers your agents need to access, without requiring your developers to build complex security logic into individual MCP servers.

Acting as a security proxy for MCP servers, the agent gateway:

- Validates delegation tokens and enforces scopes and fine-grained authorization before agent requests reach backend resources.
- Audits, throttles, and terminates agent traffic.
- Includes specialized MCP filters.

Agent gateway is made available as part of our new Identity for AI solution. Contact your account executive to find out more.

Learn more:

- MCP security gateway
- McpAuditFilter: Record centralized audit trails of all agent requests
- McpProtectionFilter: Bind OAuth 2.0 resource server configurations to MCP endpoints
- McpValidationFilter: Validate JSON-RPC payloads and client message formats

Agent detection

Improved PingOne Protect

PingOne Protect now detects agentic automation acting on behalf of a user or system. It can:

- Identify a subset of specific agent types in the risk evaluation response.
- Use browser fingerprinting, behavioral telemetry, and device attributes to distinguish human traffic from agentic activity.

Agent detection is made available as part of our PingOne Protect solution. Contact your account executive to find out more.

Learn more in Bot detection.

Get started

To start securing your AI agents with Identity for AI:

- Configure runtime identity to secure AI agents with PingOne and PingFederate
- Integrate with AWS Bedrock and Cloudflare MCP servers
- Explore additional Identity for AI resources

February 2026

February 10

Identifying agents with token exchange

New

We've added documentation on identifying agents with token exchange.

February 5

Secure MCP servers with PingGateway

New PingGateway

We've linked to a tutorial on securing MCP servers with PingGateway.

January 2026

January 16

Secure a Cloudflare MCP with Ping Identity

New PingOne PingOne Advanced Identity Cloud DaVinci PingFederate

We've added documentation on how to integrate a Cloudflare Workers MCP server with Ping Identity products:

- PingOne
- PingOne Advanced Identity Cloud
- DaVinci
- PingFederate

Secure AWS Bedrock AgentCore Identity with Ping Identity

New PingOne PingOne Advanced Identity Cloud PingFederate

We've added documentation on how to integrate AWS Bedrock AgentCore Identity with Ping Identity products:

- PingOne
- PingOne Advanced Identity Cloud
- PingFederate

December 2025

December 22

Best practices

Improved

We've added a video to demonstrate best practices when implementing agentic architecture.

November 2025

November 13

MCP and A2A

Improved

We've added information on how MCP and A2A work together.

JWKs and JWTs

Improved PingOne Advanced Identity Cloud

We've included information in Authorize an AI Agent to Perform Tasks on Your Behalf on how to generate JWKs and JWTs. Learn more in Generate JWKs and step 2 of Get an auth request ID in the Advanced Identity Cloud documentation.

October 2025

October 9

Secure your AI agents with Identity for AI

New

We're excited to announce the launch of the **Identity for AI** portal, Ping Identity's dedicated resource for securing and governing the next generation of autonomous AI.

The rapid shift to agentic AI introduces countless security challenges. AI agents capable of actions such as booking flights and executing code must become trusted digital users. Our foundational Identity for AI framework integrates AI agents securely into your existing IAM ecosystem and helps you implement core best practices such as delegation, not impersonation and least privilege for every AI agent. Learn more in What Is Identity for AI?

Use this tutorial to start building trusted, secure, and compliant AI systems: Authorize an AI Agent to Perform Tasks on Your Behalf.

Check back often for new Identity for AI resources, including additional tutorials, use cases, and more.

What Is Identity for AI?

Artificial intelligence (AI) is transforming the way organizations operate. As AI systems evolve into autonomous agents capable of performing tasks such as scheduling appointments, answering emails, booking flights, and executing code, they transition from passive tools to active users. This shift demands that AI agents operate with well-defined and verifiable identity.

{{{ Video removed }}}

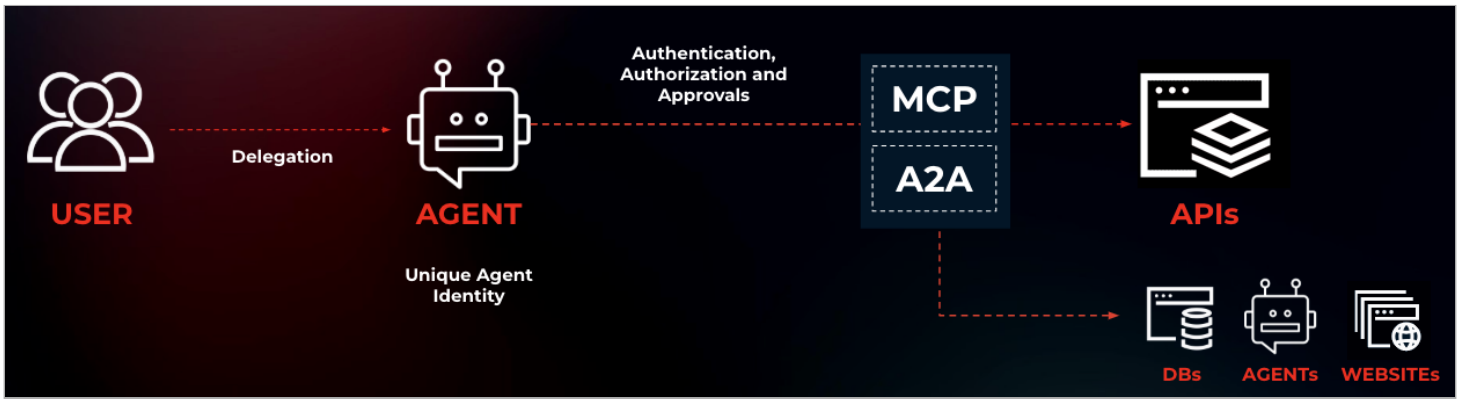
Identity and access management (IAM) solutions are essential for ensuring the security, governance, and regulatory compliance of AI agents, particularly in sensitive domains like finance and healthcare.

Identity for AI refers to the foundational mechanisms that integrate AI agents into existing security and governance frameworks. These mechanisms enable AI agents to:

- Authenticate themselves securely.
- Act with authority, either independently or on behalf of a human.
- Gain authorization to access sensitive data and perform actions.
- Have their activity audited for transparency and compliance.
- Operate within existing IAM ecosystems, leveraging established infrastructure.

IAM's central role

AI agents capable of performing tasks on behalf of a user or system introduce new IAM challenges. Agents frequently need to access sensitive data and use external tools and services, requiring a high level of trust and security.



IAM is central to managing agentic AI because:

- Agents frequently interact with external tools like APIs, web services, and databases. IAM solutions ensure that every request is properly authenticated and authorized.
- Agents often act on behalf of a human user or another system, requiring a secure method for delegation without sharing human credentials.
- An agent's access requirements can change rapidly based on its task, demanding fine-grained, dynamic authorization.

Key benefits

By adapting and extending centralized IAM policies to meet the unique challenges of agentic AI, organizations can achieve several benefits:

Secure access management

Implement robust authentication and authorization mechanisms to protect AI systems and the data they access from unauthorized use and potential breaches.

Data governance and compliance

Ensure that AI agent access and data management activities comply with industry regulations and organizational data policies.

Auditability and transparency

Maintain detailed logs of AI system interactions to support internal auditing and external compliance efforts.

Scalability and integration

Centralize identity management of AI agents to allow seamless integration of new agents with existing IAM solutions.

Risk mitigation

Define, manage, and enforce policies to identify and mitigate risks associated with autonomous AI actions through continuous monitoring and analysis.

Best practices

As AI becomes more integrated into business workflows, it's crucial to implement IAM best practices to ensure secure and compliant AI systems.

{{{ Video removed }}}

> **Tip**
>
> Check back often for additional guidance related to best practices as Identity for AI evolves.

Key principles include:

Principle	Description
Know your agents	Classify AI agents based on their capabilities, access needs, and risk profiles to tailor your IAM strategies.
Detect agents	Implement mechanisms to identify and authenticate AI agents, ensuring they are who they claim to be before granting access.
Delegation, not impersonation	Use delegated access with limited scopes instead of sharing human credentials with agents.

Principle	Description
Enforce least privilege	Grant agents only the minimum set of access rights and permissions required for their current tasks.
Human-in-the-loop (HITL)	Require explicit human approval for sensitive, high-risk, or irreversible agentic actions.
Monitor agent activity	Log and analyze agent activities to detect anomalies, policy violations, and unauthorized behavior in real time.

Get started

To get started with Identity for AI:

1. **Assess your needs**: Evaluate your organization's AI initiatives and identify security and governance requirements.
2. **Train your team**: Ensure that your team is knowledgeable about Identity for AI best practices.
3. **Implement best practices**: Follow Identity for AI best practices for securing your AI systems.
4. **Monitor and optimize**: Continuously monitor the performance of your AI systems and adjust Identity for AI controls.

> **Tip**
>
> Contact your Ping Identity representative for more information and help with getting started.

Learn more

Use the following resources to learn more about Identity for AI and how it can benefit your organization:

- Agent Types
- Authorize an AI Agent to Perform Tasks on Your Behalf

What Are AI Agents?

Artificial intelligence (AI) agents are systems that can act autonomously on behalf of a user or another system to complete tasks.

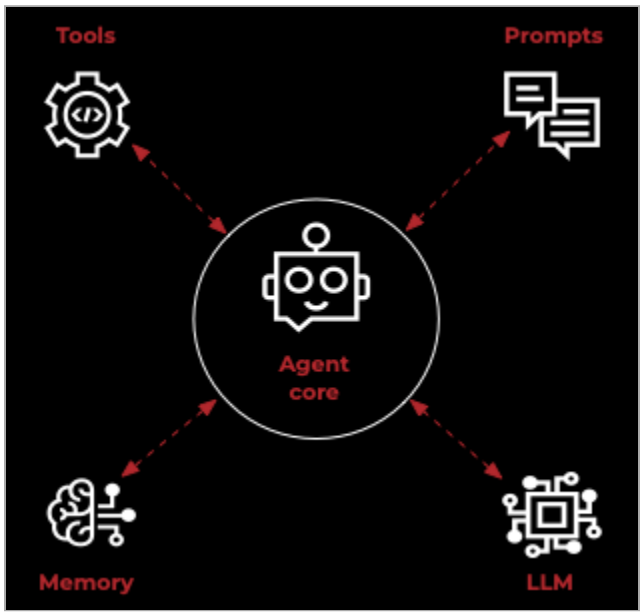
Unlike other AI models that only generate or predict text, agents combine reasoning, memory, and the capacity for real-world action. These capabilities allow agents to interact dynamically with external systems, maintain context across sessions, and solve complex problems with minimal human intervention.

Architecture

Agent architectures define how intelligence, memory, and action are organized into a functioning system. At the highest level, they specify where reasoning occurs, how agents coordinate, and how external systems integrate.

Core components

Agents are built from several core components. Each component plays a distinct role in turning user goals into real-world solutions. Together, they provide the reasoning, context, and execution layers that make agents more than just large language models (LLMs).



Large language models

An LLM is the reasoning engine of an agent. It performs the high-level cognitive tasks that define an agent's intelligence, including natural language processing, reasoning, planning, and decision making.

The LLM interprets user goals and determines when and how to call on other components, such as tools or memory, to complete a task.

Tools

Instead of relying solely on static training data, agents use tools to interact dynamically with the world. Tools are external functions, services, or APIs that an agent invokes to perform specific actions. This can include retrieving up-to-date information from the web, querying a database, sending an email, executing code, or interacting with a third-party application.

An LLM decides when and how to invoke a tool by generating a structured command. The agent's runtime executes this command, bridging the gap between language prediction and real-world action.

Agents integrate with tools in the following ways:

- **Direct-to-interface**: The agent interacts directly with a service through a well-defined interface, such as an API endpoint or an SDK library.
- **Abstract intention**: The agent expresses an intention, and a mediation layer such as Model Context Protocol (MCP) translates this intention into an API call.

Learn more about tool calling in Tools.

Memory

Memory provides agents with the ability to store and retrieve information from previous interactions, enabling them to maintain context and handle stateful, multistep tasks. Without memory, each agent interaction would be independent and isolated, limiting the agent's ability to perform complex work.

- **Short-term memory**: Keeps track of the current conversation or task, ensuring continuity. For example, remembering what the user asked three steps earlier in a workflow.
- **Long-term memory**: Stores knowledge over time, such as user preferences, historical actions, or facts relevant to recurring tasks. Long-term memory enables agents to improve across sessions and adapt to individual users.

Prompts

Prompts define how an LLM interprets and responds to tasks. They provide instructions, context, and constraints that shape agent behavior. In agentic systems, prompts aren't limited to user input. They can also include system prompts (defining role, goals, or style) and dynamically generated prompts (created as the agent reasons through steps).

Well-designed prompts serve as the operating instructions that guide the agent's reasoning.

Workflows

The structure of an agentic workflow depends on the number of agents involved, how responsibilities are delegated, and how coordination is managed. Two common patterns are single-agent workflows and multi-agent workflows.

Single-agent workflow

In a single-agent workflow, one LLM manages reasoning, tool use, and memory. This model is simple to deploy and delivers quick results because there's no orchestration overhead. It works best for narrow, task-driven scenarios such as scheduling meetings, resolving straightforward support tickets, or generating reports. Although efficient, a single agent could struggle with problems that span multiple domains or require deeper specialization.

Multi-agent workflow

Multi-agent workflows distribute tasks across specialized agents that collaborate to reach an outcome. A coordinator or manager agent often handles communication and delegation. This model mirrors human teamwork and excels in complex, multi-domain scenarios.

For example, one agent might gather research, another summarize findings, and a third generate tailored recommendations. In business contexts, agents might divide responsibilities across data extraction, compliance checks, and payment processing.

Agent lifecycle

Agents operate in a reasoning-action loop that enables them to interpret user input, determine a course of action, and dynamically adapt based on results.

1. **Input**: The process begins with a user instruction, system event, or scheduled task.
2. **Interpretation**: The LLM processes the input, consulting short-term memory to maintain continuity with previous interactions.
3. **Reasoning**: The agent evaluates next steps, determining whether the task requires external tools, additional context, or further reasoning.
4. **Action**: If the agent requires external data or capabilities, it invokes a tool to perform a specific action, such as querying long-term memory or triggering an integration.
5. **Observation**: The agent processes the retrieved context and incorporates it into its reasoning.
6. **Feedback**: The agent evaluates whether the outcome meets the intended objective. If not, it refines its reasoning or actions in the next cycle.
7. **Response**: The LLM generates a final response or executes the requested action.
8. **Memory update**: The agent stores relevant details so it can improve future interactions.

This loop is iterative. Agents repeat reasoning and action cycles until a satisfactory result is produced. By chaining together reasoning, action, and feedback, agents become continuously learning, adaptive assistants.

Use cases

Agents are automating work and augmenting human decision making across industries. Examples of such applications:

Use case	What the agent does	Example
Digital shopping assistant	Guides customers through a retail site by answering questions, recommending products, assisting with checkout, and tracking orders. Provides a conversational interface that personalizes the shopping experience.	A retail agent helps a customer find running shoes in their size, compares styles based on budget and activity, adds the chosen pair to the cart, applies a promo code, and provides delivery tracking updates.
Customer support	Resolves common inquiries (password resets, billing, order tracking), maintains continuity across channels, and escalates complex cases with summaries.	An airline agent handles rebooking and baggage claims, while routing loyalty point or status discrepancies to live agents.
Personal productivity	Manages scheduling, email triage, and travel bookings across apps. Acts directly on behalf of the user instead of just sending reminders.	A personal agent summarizes daily email threads, drafts responses, and generates weekly travel itineraries.

Use case	What the agent does	Example
Business automation	Orchestrates multi-step workflows like invoice validation, compliance checks, and HR onboarding. Adapts to system changes or exceptions.	A bank agent automates compliance tasks, such as collecting documents, verifying identities, and escalating suspicious cases.

Learn more

Use the following resources to learn more about agent types and corresponding security challenges:

- Agent Types
- Authorize an AI Agent to Perform Tasks on Your Behalf

Computer-Using Agents (CUAs)

A computer-using agent (CUA) is an agent that operates a computer like a human would, interacting with graphical user interfaces (GUIs) and command-line interfaces (CLIs) instead of structured APIs.

Using techniques like computer vision and accessibility hooks to navigate screens, CUAs can simulate mouse clicks, keystrokes, or terminal commands to perform tasks.

How CUAs differ from API-driven agents

The following table highlights key differences between API-driven agents and CUAs across identity, security, and operational categories. This comparison shows why CUAs introduce unique security challenges compared to API-based approaches.

Category	API-driven agents	CUAs
Interface type	Structured, machine-readable APIs	Human-facing GUIs and CLIs
Stability and resilience	Predictable, versioned, and contract-based	Minor UI changes can easily break workflows
Authentication model	Modern standards such as OAuth 2.0, OIDC, or workload identities	Human-style logins (usernames, passwords, shared bot accounts)
Authorization model	Fine-grained, least-privilege policies enforceable at the API layer	Broad access after sign-on; difficult to constrain privileges
Auditability and attribution	Actions tied to unique workload identities or service principals	Shared accounts reduce accountability and audit clarity
Execution environment	Cloud-native or service-based runtimes	Local desktops, VMs, or RDP sessions
Security posture	Supports IAM-native controls (short-lived tokens, conditional access)	Often relies on long-lived sessions vulnerable to hijacking or replay
Best suited for	Modern systems with robust API coverage	Legacy, desktop, or closed-source applications without APIs

Tools

Artificial intelligence (AI) agents can extend their capabilities far beyond static training data by leveraging external tools to autonomously solve complex, real-world tasks. Tools are external functions, services, or APIs that an agent can invoke to perform specific actions, retrieve information, or augment reasoning.

Why tools matter

Tools allow AI agents to perform tasks that would be difficult using only their built-in knowledge. With tools, agents can:

Access up-to-date information

Tools enable agents to query external knowledge sources in real time. For example, an agent might:

- Query a company's internal database to check a customer's policy details
- Access a public API to retrieve the latest stock prices

Perform actions on behalf of users

Agents can interact with systems or services to execute tasks, such as:

- Scheduling a meeting through a calendar API
- Sending an email
- Booking travel arrangements through an external service API

Enhance reasoning and decision-making

Agents can leverage specialized tools to improve accuracy and efficiency, including:

- Calculators or logic solvers for precise computation
- Vector databases for semantic search or similarity queries
- Analytics or simulation engines to evaluate complex scenarios

Integration with protocols

To ensure secure, auditable, and trustworthy tool usage, agents rely on protocols such as Model Context Protocol (MCP) and Agent-to-Agent (A2A).

MCP

MCP provides a standardized framework for agents to discover and invoke tools securely. It enforces identity, authorization, and trust boundaries.

For example, an agent calling a sensitive HR API would require MCP-managed credentials and scopes to perform the action safely.

Learn more in What is Model Context Protocol (MCP)?

A2A

Sometimes an agent delegates a subtask to another agent instead of calling a traditional API. The same security and trust principles apply: the calling agent must authenticate, obtain authorization, and interact according to organizational policies.

Learn more in What is Agent2Agent Protocol (A2A)?

How MCP and A2A work together

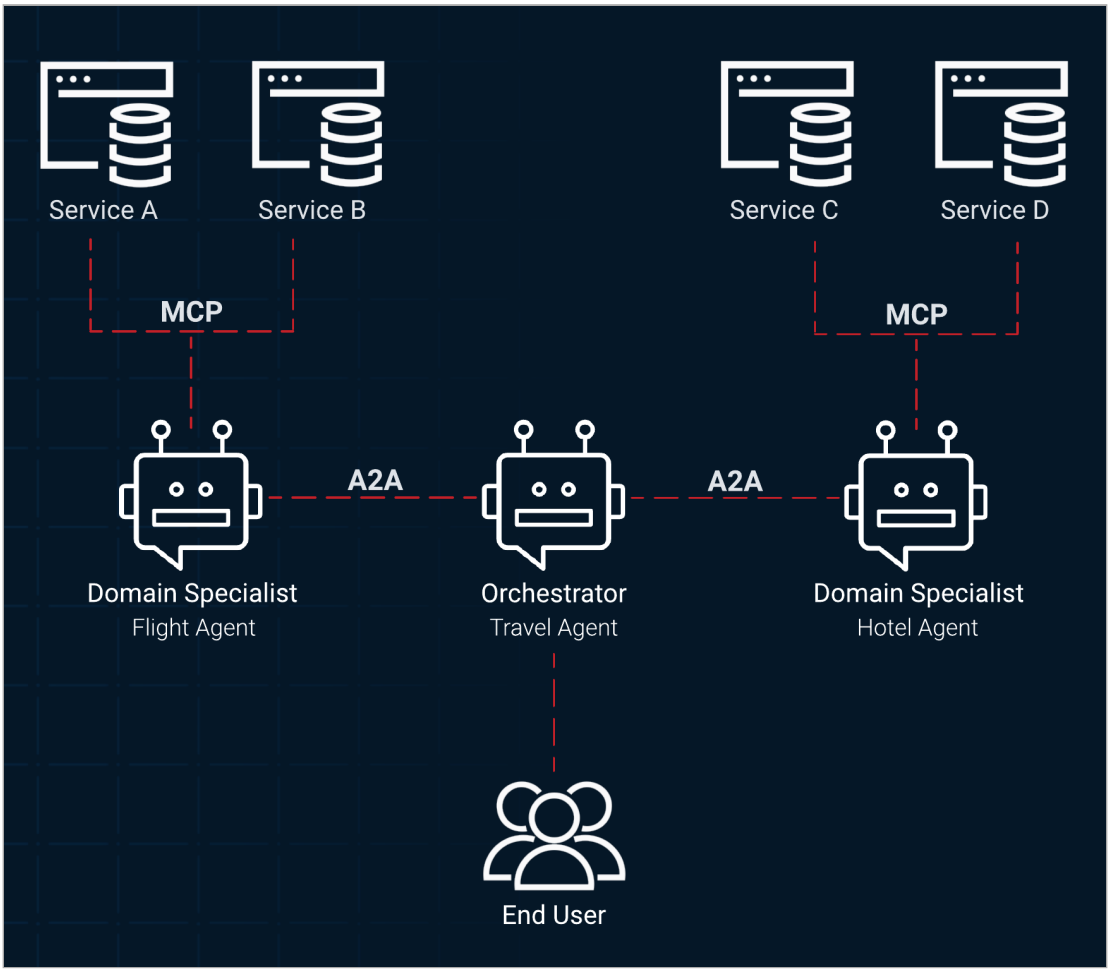
The emerging landscape of AI agents requires efficient collaboration and specialization. Complex tasks require not only individual agents but cohesive teams of agents, each excelling in a specific domain. MCP and A2A are the keys to enabling this sophisticated ecosystem.

Rather than competing standards, these two protocols are complementary. Each addresses a distinct, yet essential layer of the agentic architecture. An agent primarily uses A2A to communicate with other agents and MCP to interact with its specific tools and resources.

Think of MCP as vertical integration, that is, how a single agent interacts with the external world of tools, data, and APIs. It acts as the agent's nervous system, standardizing the connection between a Large Language Model (LLM) and the resources required to complete a task.

In contrast, A2A addresses horizontal integration, which is how independent, autonomous agents communicate and collaborate as peers. A2A is the backbone of multi-agent architecture, governing the agent-to-agent communication that enables complex, multi-step workflows.

The following diagram illustrates how MCP and A2A work together to enable a seamless workflow from collaboration to execution:



Orchestration with A2A

A complex user request first goes to an orchestrating agent. This agent uses A2A to delegate sub-tasks to specialized peer agents. For example, a travel agent might use A2A to communicate with a flight agent and a hotel agent.

Execution with MCP

Each specialized agent, having received its delegated task, uses MCP internally to perform the specific steps. The flight agent uses MCP to call the airline API tool to check prices, book a flight, and so on.

Reporting with A2A

The specialized agents package the results of the tool executions as artifacts and return them to the orchestrating agent using the A2A protocol. The orchestrating agent synthesizes these results to provide a comprehensive, final answer back to the user.

By leveraging MCP and A2A capabilities, agents can access information, perform tasks, and collaborate with other agents while maintaining security, user consent, and auditability.

What is Model Context Protocol (MCP)?

The foundational challenge in scaling agentic systems is managing the connections between artificial intelligence (AI) agents and the vast array of tools they need to perform their work.

To solve real-world problems, AI agents require connections to external data and services, such as APIs and databases. Without a common standard, each integration must be written and maintained separately, creating brittle connections and duplicated effort. This approach doesn't scale, especially when multiple agents use the same tools.

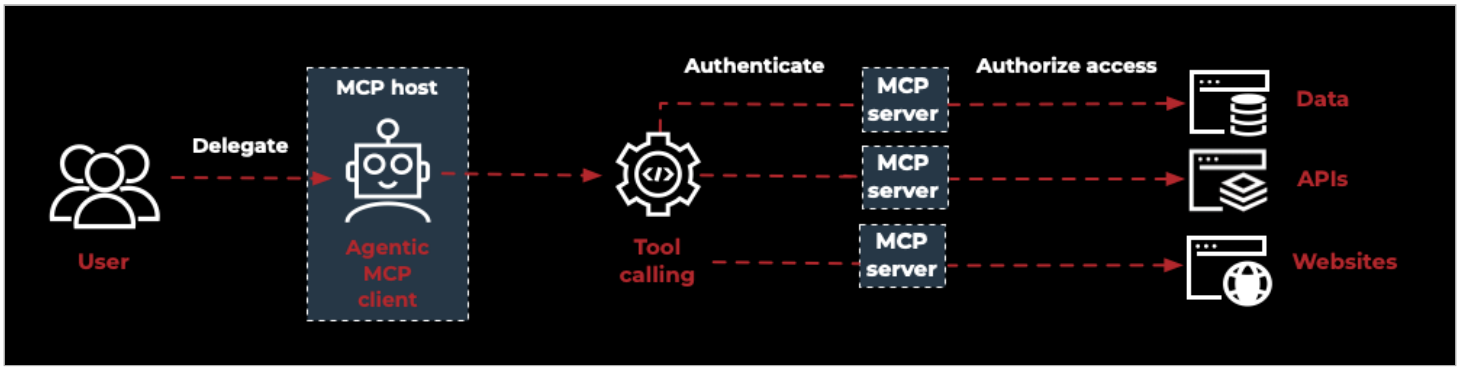
The Model Context Protocol (MCP), developed by Anthropic, addresses this challenge by standardizing how agents connect to tools. By providing a consistent layer for agent-tool interaction, MCP reduces friction for developers, simplifies maintenance, and creates a foundation for interoperable, secure agent ecosystems.

Key benefits

MCP's standardized integration layer delivers:

- **Consistent connection semantics**: Developers can swap in a new tool without rewriting core integration logic.
- **Secure authentication and authorization**: MCP servers support OAuth 2.0, eliminating the need for hard-coded secrets or long-lived credentials.
- **Capability discovery**: Agents can automatically discover available tools, resources, and prompts instead of relying on hard-coded lists.
- **Reduced duplication**: Multiple agents can reuse the same MCP server, reducing the operational burden of maintaining separate connectors.

Architecture



The MCP architecture includes three core components:

MCP host

The application that manages MCP clients and initiates connections to MCP servers. It provides the environment in which agents operate and interact with tools and resources. For example, an AI-powered IDE.

MCP client

The component inside the host that maintains a standardized connection with the MCP server. It acts as a bridge between the agent's reasoning and the server's tools, resources, and prompts. The client translates agent requests into MCP-compliant messages and processes server responses in a consistent way. For example, an embedded IDE client or chatbot agent connector.

MCP server

A program that exposes capabilities to AI applications in a standardized and discoverable way. It acts as the central repository of tools, resources, and prompts that agents can invoke. MCP servers support OAuth 2.0 for authentication and authorization, ensuring secure access, and can run locally within a host or remotely as a centralized service.

An MCP server can expose three main types of capabilities:

- **Tools**: Executable functions that an agent can invoke to perform specific actions.
- **Resources**: Contextual data that informs agent reasoning, such as files, database records, or configuration information.
- **Prompts**: Pre-defined, templated messages or workflows that agents can present to users or use internally to structure reasoning.

MCP's technical architecture is based on the JSON-RPC 2.0 message format. This ensures that all interactions, such as tool invocation, resource retrieval, and prompt execution, follow a consistent pattern. Each message contains a unique identifier, method name, and parameters.

Workflow

A typical MCP workflow unfolds as part of an agent's reasoning-action loop:

1. **Initialization**: The MCP host launches an MCP client, which connects to the MCP server. The client queries the server for available tools, resources, and prompts. The agent inspects this metadata to understand which capabilities are available before taking action.
2. **Invocation**: When the agent determines that a specific tool or resource is needed, the MCP client sends a JSON-RPC request to the server, specifying the method and any required parameters.
3. **Execution**: The server applies authentication and authorization checks using OAuth 2.0 and least-privilege scopes, ensuring the client is permitted to execute the requested action. The server then performs the requested operation.
4. **Response**: Results are returned in a standardized JSON-RPC response. If an error occurs, the response includes a structured error object. The client relays the results to the agent, which incorporates them into its reasoning.
5. **Iteration**: Based on the results, the agent might issue additional MCP requests, chaining tool calls or resource queries to achieve higher-level goals. This iterative process allows the agent to refine its reasoning and dynamically adapt to changing conditions or new data.
6. **Observability & Logging** (Optional): Hosts or clients might log requests, responses, and execution traces to support debugging, auditing, or performance monitoring.

In the following example, an agent is connected to a support team's ticketing system. The MCP server exposes a tool called `searchTickets`, which lets agents query tickets by status, assignee, or other fields.

Suppose a support agent makes the following query to an AI agent:

Show me all my open tickets.

The LLM then processes this request and makes the following MCP call:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "searchTickets",
  "params": {
    "status": "open",
    "assignee": "me"
  }
}
```

- `method`: The MCP tool being called (`searchTickets`)
- `params`: The query parameters for filtering tickets

The MCP server returns a structured list of open tickets:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": [
    {
      "ticketId": "123",
      "title": "Login issue",
      "priority": "high"
    },
    {
      "ticketId": "124",
      "title": "Billing discrepancy",
      "priority": "medium"
    }
  ]
}
```

The agent acting as the MCP client can now summarize tickets for the user, suggest next steps, or chain further requests by calling other MCP tools.

Security challenges

While MCP streamlines tool integration, it also introduces new identity and security challenges. Each MCP server must enforce strong authentication and authorization to ensure only trusted clients can access tools and resources.

Common security challenges include:

- Configuring OAuth 2.0 correctly to avoid reliance on static secrets
- Implementing least-privilege scopes so agents can only access the data they need

- Securing multi-tenant MCP servers with fine-grained access control to prevent data leakage across users
- Protecting against denial-of-service with unbounded tool calls, and mitigating risks from prompt injection that could trigger malicious tool execution

Learn more about MCP security best practices in MCP servers and OAuth 2.0.

MCP servers and OAuth 2.0

When connecting artificial intelligence (AI) agents to external tools with the Model Context Protocol (MCP), you should ensure that each MCP server enforces sufficient authentication and authorization. Following security best practices helps protect sensitive data, prevent unauthorized access, and maintain trust in agentic workflows.

With MCP, the same modularity and flexibility that enables interoperability between agents and tools also carries considerable security risks. Developers face the following challenges when building secure MCP servers and tools:

- **Overly permissive access tokens**: Static or long-lived tokens can grant overly broad access to all tools. If these tokens are compromised, they could allow unrestricted access to tools. To prevent this, use scoped, short-lived tokens or OAuth 2.0 flows with refresh mechanisms to limit exposure.
- **Insufficient authorization checks**: Authentication alone isn't enough. MCP servers must enforce fine-grained authorization to ensure clients only access capabilities they have permissions for. To adhere to least-privilege principles, apply access policies on a per-tool and per-action basis.
- **Potentially unsafe tools**: Registered tools can introduce vulnerabilities if not properly vetted. Limit tool registration to trusted sources and consider capability restrictions to mitigate risk.

The role of OAuth 2.0

OAuth 2.0 plays a pivotal role in securing MCP servers by providing a standardized framework for authentication and authorization. By leveraging OAuth 2.0, MCP servers can ensure that only authorized clients access tools, resources, and prompts.

Authorization server architecture

There are different ways to implement OAuth 2.0 with MCP, depending on the architecture and responsibilities of each component:

Native authorization server

The MCP server can act as an OAuth 2.0 authorization server, directly handling:

- Authorization code issuance.
- Token generation and refresh.
- Scope-based access control.

This approach gives the MCP server full control over security and the token lifecycle, but it requires deeper infrastructure and policy management.

Proxy to external authorization server

The MCP server exposes standard OAuth 2.0 endpoints (`/authorize` , `/token`) but proxies requests to an external authorization server, such as PingOne Advanced Identity Cloud. This architecture enables:

- Seamless integration with existing identity providers.
- Delegation of token issuance and verification.
- Use of familiar login and consent mechanisms.

Agent redirected to external authorization server

The MCP server doesn't directly expose OAuth 2.0 endpoints. Instead, it:

- Instructs the client agent to begin the OAuth 2.0 flow with an external authorization server.
- Relies on the client agent to manage tokens and present them back to the MCP server.

This architecture minimizes operational burden for the MCP server by entrusting the agent with token security and refresh logic.

Authorization server requirements

When deploying an MCP server for AI agents, your choice of authorization server is critical to ensuring secure client onboarding, token issuance, and delegated access. The authorization server should support the following:

- **Authorization Code Flow**: This flow ensures that AI agents acting as confidential clients can securely exchange authorization codes for tokens, reducing the risk of token exposure when compared to implicit flows. This flow provides a secure foundation for agent-user interactions.

  Learn more about the Authorization code grant in the Advanced Identity Cloud documentation.

- **Proof Key for Code Exchange (PKCE)**: PKCE is important for public clients, such as AI agents running in browsers or user devices, that can't safely store a client secret. By requiring a dynamically generated code verifier and challenge, PKCE mitigates code interception attacks during the Authorization Code Flow.

  Learn more about the Authorization code grant with PKCE in the Advanced Identity Cloud documentation.

- **Dynamic Client Registration (DCR)**: DCR enables new AI agents to onboard without manual administrative intervention. This is critical in environments where agents are created dynamically or updated frequently. DCR enables automatic provisioning of credentials and redirect URIs in a standardized, secure way.

  Learn more about Dynamic client registration in the Advanced Identity Cloud documentation.

- **Fine-grained scopes and claims**: Your authorization server should allow granular token scopes and claims to ensure tokens issued to AI agents follow the principle of least privilege. For example, one agent might require the `prices:read` scope, while another requires the `orders:write` scope. Narrow scopes reduce negative impact if a token is exposed or misused.

  Learn more about Scopes in the Advanced Identity Cloud documentation.

- **Token introspection and UserInfo endpoints**: Downstream services integrated with the MCP server often need to validate tokens before serving requests. Introspection and UserInfo endpoints allow the MCP server to verify token status, audience, and claims, ensuring that expired, revoked, or mis-scoped token are rejected.

You can find information about OAuth 2.0 endpoints supported by Advanced Identity Cloud in OAuth 2.0 endpoints in the Advanced Identity Cloud documentation.

Learn more

Use the following resources in the MCP specification to learn more about securing MCP servers:

- Authorization
- Security Best Practices

AWS and Ping Identity

Secure AWS Bedrock AgentCore Identity with the Ping Identity Platform

You can integrate Ping Identity's identity providers (IdPs) with AWS Bedrock AgentCore Identity to secure agent-based workloads.

Specifically, you can configure each IdP, PingOne, PingOne Advanced Identity Cloud, and PingFederate, as an:

Inbound IdP for AgentCore Gateway and Runtime

This enables agents to authenticate and authorize end users using OpenID Connect (OIDC) tokens issued by Ping Identity.

Outbound credential provider for AgentCore Identity

This enables agents to securely obtain OIDC access tokens from Ping Identity in order to call downstream APIs and protected resources.

Goals

- Centralize authentication and authorization for Bedrock AgentCore agents using the Ping Identity Platform.
- Enforce consistent OIDC controls such as audience (`aud`) validation, scopes, and grant types across inbound and outbound agent interactions.
- Support both user-based flows (authorization code grants) and machine-to-machine flows (client credentials grants) as defined in the AWS Bedrock AgentCore Identity model.

This aligns with AWS's recommended IdP integration pattern for AgentCore, as described in the AWS Bedrock AgentCore Identity documentation, and demonstrates how Ping Identity products act as both trusted token issuers and credential providers within agent-based architectures.

AWS Bedrock AgentCore Identity model

AWS Bedrock AgentCore Identity provides a standardized mechanism for:

Inbound authentication

Validating OAuth 2.0/OIDC tokens presented to AgentCore Gateway and Runtime by agent users.

Outbound credential acquisition

Securely retrieving OAuth 2.0 access tokens that agents use to access external systems.

AgentCore Identity relies on:

- OIDC discovery metadata to locate authorization, token, and JSON Web Key Set (JWKS) endpoints.
- Audience (`aud`) and scope validation to ensure tokens are issued for the correct resource.
- Explicit configuration of IdPs and credential providers.

PingOne, PingOne Advanced Identity Cloud, and PingFederate all satisfy these requirements and can be integrated with the following patterns.

PingOne integration

You can configure Ping Identity as an IdP for accessing AgentCore Gateway and Runtime, or as an AgentCore Identity credential provider for outbound resource access. This allows your agents to authenticate and authorize agent users with PingOne as the IdP and authorization server, or your agents to obtain credentials to access resources authorized by PingOne.

To add PingOne as an IdP and authorization server for AgentCore Gateway and Runtime, you must:

- Configure the discovery URL for your PingOne environment so AgentCore Identity can retrieve OAuth and OIDC metadata.
- Configure and validate expected `aud` claims to ensure access tokens are issued for the correct protected resource.

Configuring PingOne for inbound authentication

1. Sign on to the PingOne admin console.
2. Go to Applications > Applications.
3. Click the + icon to create a new application.
4. In the **Application Name** field, enter a name.
5. In the **Application Type** section, click **OIDC Web App**, and then click **Save**.
6. Configure your application as a user federation OAuth 2.0 client:
    1. Select your application and go to the **Configuration** tab.
    2. In the **Response Type** section, select the **Code** checkbox.
    3. In the **Grant Type** section, select the **Authorization Code** checkbox, the **Client Credentials** checkbox, or both depending on your use case.
    4. In the **Token Endpoint Authentication Method** list, select **Client Secret Post**.
    5. (Optional) If using the authorization code grant type, enter the **Redirect URI**.
7. Create a custom resource.
    1. Go to Applications > Resources and click the + icon to create a new resource.
    2. In the **Resource Name** field, enter a name for the resource, and then click **Next**.
    3. In the **PingOne Mappings** list, select a value to map to the `sub` attribute, and the click **Next**.
    4. Click **+ Add Scope+** to define a scope and assign it to the application.

5. Click **Save**.

> **Note**
>
> You will set this resource name as the `aud` claim for Client Credentials access tokens.

8. Configure the AgentCore inbound authentication:

    1. In the **Discovery URL** field, enter the **OIDC Discovery Endpoint** value from the **Overview** tab on the PingOne application details pane.

    2. In the **Allowed Audiences** field, enter the resource name you created in step 7.

You can find more information in the PingOne API documentation.

Configuring outbound authentication

Outbound configuration mirrors inbound configuration, with the additional step of adding the AgentCore Identity callback URL to the application's redirect URIs.

AgentCore outbound resource provider configuration

```
{
  "name": "PingOne",
  "credentialProviderVendor": "PingOneOauth2",
  "oauth2ProviderConfigInput": {
    "includedOauth2ProviderConfig": {
      "clientId": "<CLIENT_ID>",
      "clientSecret": "<CLIENT_SECRET>",
      "authorizeEndpoint": "https://auth.pingone.com/<ENV_ID>/as/authorize",
      "tokenEndpoint": "https://auth.pingone.com/<ENV_ID>/as/token",
      "issuer": "https://auth.pingone.com/<ENV_ID>/as"
    }
  }
}
```

PingOne Advanced Identity Cloud integration

You can configure PingOne Advanced Identity Cloud as an IdP for accessing AgentCore Gateway and Runtime, or as an AgentCore Identity credential provider for outbound resource access. This enables both user-based and machine-based agent interactions secured by PingOne Advanced Identity Cloud.

Configuring PingOne Advanced Identity Cloud for inbound authentication

1. In the PingOne Advanced Identity Cloud admin console, go to Applications > Custom Application.

2. Select **OIDC - OpenID Connect** and then click **Service**.

3. Complete the following fields:

    1. `Application Name`

2. `Description`
3. `Owner`

4. Create the **Client ID** and **Client Secret**.
5. On the **Sign-On** tab, configure the following fields:
    1. Authorization code or client credentials grant types.
    2. **Redirect URI** if using Authorization Code.
6. Configure AgentCore inbound authentication:
    1. In the **Discovery URL** field, enter the **OIDC Discovery Endpoint** from the **Sign-On** tab.
    2. In the **Allowed Audiences** field, enter the **Client ID**.

Configuring outbound authentication

Outbound configuration mirrors inbound configuration, with the additional step of adding the AgentCore Identity callback URL to the application's redirect URIs.

AgentCore outbound resource provider configuration

```
{
  "name": "PingOne AIC",
  "credentialProviderVendor": "CustomOauth2",
  "oauth2ProviderConfigInput": {
    "includedOauth2ProviderConfig": {
      "clientId": "CLIENT_ID",
      "clientSecret": "CLIENT_SECRET",
      "oauthDiscovery": {
        "discoveryUrl": "https://<PINGONE_AIC_TENANT>/am/oauth2/realms/root/realms/<REALM>/.well-known/
openid-configuration"
      }
    }
  }
}
```

PingFederate integration

You can configure PingFederate as an IdP for accessing AgentCore Gateway and Runtime, or as an AgentCore Identity credential provider for outbound resource access, supporting enterprise OAuth deployments and fine-grained token control.

Configuring PingFederate for inbound authentication

1. In the PingFederate admin console, go to Applications > OAuth > Clients and click **Add Client**.
2. Configure the following fields:
    1. **Client ID** and **Client Secret**.

2. **Redirect URI**, if applicable.
3. For **Allowed Grant Types**, select **Authorization Code** or **Client Credentials**.

3. Go to System > OAuth Settings > Scope Management and create one or more scopes.
4. Go to Applications > OAuth > Access Token Management and configure the `aud` claim by setting the **Audience Claim Value**.
5. Allow the client to request the appropriate scopes and grant types.
6. Configure AgentCore inbound authentication:
    1. Set **Discovery URL** to `https://<PINGFED_SERVER_HOSTNAME>/.well-known/oauth-authorization-server`.
    2. Set **Allowed Audiences** to the configured audience value.

Configuring outbound authentication

Outbound configuration mirrors inbound configuration, with the additional step of adding the AgentCore Identity callback URL to the application's redirect URIs.

AgentCore outbound resource provider configuration

```
{
  "name": "PingFederate",
  "credentialProviderVendor": "CustomOauth2",
  "oauth2ProviderConfigInput": {
    "includedOauth2ProviderConfig": {
      "clientId": "<CLIENT_ID>",
      "clientSecret": "<CLIENT_SECRET>",
      "oauthDiscovery": {
        "discoveryUrl": "https://<PINGFED_SERVER_HOSTNAME>/.well-known/oauth-authorization-server
      }
    }
  }
}
```

Result

You've successfully integrated PingOne, PingOne Advanced Identity Cloud, or PingFederate with AWS Bedrock AgentCore Identity and can apply consistent enterprise identity controls to AI agents.

Cloudflare and Ping Identity

Secure a Cloudflare Workers MCP server with PingFederate

The Cloudflare Workers Model Context Protocol (MCP) server functions as a resource server within the OAuth architecture. It validates incoming requests and facilitates token exchange, ensuring secure communication between the MCP client and downstream APIs.

This tutorial uses the `remote-mcp-ping-federate` directory in the Ping Identity cloudflare-mcp Git repository.

> **Warning**
>
> This tutorial is designed to help you get started quickly. Although we have implemented several security controls, you must implement all preventive and defense-in-depth security measures before deploying to production. Learn more in Security Best Practices in the MCP documentation.

How does it work?

This architecture bridges the stateless nature of Cloudflare Workers with the stateful requirements of an authenticated MCP session.

Authentication and client bootstrapping

When an unregistered MCP client tries to connect to the MCP server without a token, the server provides the necessary details for the client to perform Dynamic Client Registration (DCR). This allows the client to handle the user login and consent with PingFederate automatically, provisioning the tokens required to both connect to the MCP server and for the MCP server to execute delegated token exchanges.

> **Note**
>
> This implementation utilizes DCR to handle client onboarding.
> Although the MCP protocol recommends Client ID Metadata Document (CIMD) as the new standard, DCR remains the only production-ready option currently supported by enterprise identity providers (IdPs) like PingFederate.
> Future versions of this architecture might transition to CIMD as support becomes available.

Cloudflare agents (state and transport)

The MCP server extends the `McpAgent` class, which automatically wraps the MCP logic in a durable object. This handles the following infrastructure requirements:

Session persistence

It creates a dedicated, isolated environment for each MCP connection and securely persists the PingFederate tokens in the durable object's storage (`this.props`).

Network transport

The agent manages the raw HTTP connection. It accepts incoming requests and keeps the response open as a Server-Sent Events (SSE) stream, enabling the durable object to push real-time updates back to the client over the single endpoint.

MCP SDK (tool logic)

The official `@modelcontextprotocol/sdk` is used to define the actual capabilities of the MCP server. Inside the agent is an `McpServer` instance that manages the serialization of JSON-RPC messages and tool definitions.

Before you begin

1. Deploy the Todo API.
2. Deploy the MCP server.

> **Note**
>
> You can use any PingFederate-secured API. This tutorial uses a Cloudflare Workers API, but you can connect any API to an MCP server with this architecture.

Tasks

1. Access the remote MCP server from the Cloudflare Workers AI LLM Playground.
    1. Go to https://playground.ai.cloudflare.com.
    2. Connect to your MCP server using the following URL pattern:

        ```
        https://remote-mcp-ping-federate.<your-subdomain>.workers.dev/mcp
        ```

2. Access the remote MCP server from Claude Desktop.
    1. Open Claude Desktop and go to **Settings** > **Developer** > **Edit Config**.

        This opens the configuration file that controls which MCP servers Claude can access.

    2. Replace the content with the following configuration, and then save:

        ```
        {
          "mcpServers": {
            "todo-mcp": {
              "command": "npx",
              "args": [
                "mcp-remote",
                "https://remote-mcp-ping-federate.<ENV>.workers.dev/mcp"
              ]
            }
          }
        }
        ```

    3. Restart Claude Desktop.

        A browser window opens showing your OAuth sign-on page.

    4. Complete the authentication flow to grant Claude access to your MCP server.

        After granting access, the tools are available for you to use.

    5. You can ask Claude to use the tools that populate the **Tools** list. For example: "Can you tell me what is in my Todo list?"

        Claude invokes the tool and shows the result generated by the MCP server.

{{{ Video removed }}}

Secure a Cloudflare Workers MCP server with PingOne

In this configuration, the Cloudflare Workers Model Context Protocol (MCP) server uses the Cloudflare Workers OAuth Provider, which delegates authentication to PingOne. This enables clients, such as AI agents, to call a protected API on behalf of an authenticated end user.

> **Important**
>
> While serving as a resource server for MCP clients, this MCP server also fulfills two distinct OAuth roles:
>
> - An OAuth server to your MCP clients
> - An OpenID Connect (OIDC) client for your PingOne environment

This configuration uses Cloudflare to manage consent. To use this configuration with PingOne DaVinci-managed consent, refer to Secure a Cloudflare Workers MCP server with PingOne DaVinci.

Before you begin

This tutorial uses the `remote-mcp-pingone/mcp` directory in Ping Identity's cloudflare-mcp Git repository.

Stack

Role	Name	Description
Platform	Cloudflare Workers	Serverless execution
Framework	Hono	Lightweight API endpoints
Agent Execution	Cloudflare Agents SDK	Base class for implementing the stateful MCP server
Session State	Cloudflare Durable Objects	Provides stateful, isolated storage for each MCP connection
OAuth Core	Cloudflare Workers OAuth Provider	Orchestrates the OAuth flow, delegating authentication to PingOne
Ephemeral State	Cloudflare Workers KV	Stores OAuth state required by the workers oauth provider

Requirements

- Node.js v20 or later
- A PingOne environment
- A Cloudflare account and Wrangler CLI enabled
- Todo API deployed
- MCP Inspector installed

Structure

```
mcp/
├── package.json                    # Dependencies and scripts
├── tsconfig.json                   # TypeScript compiler settings
├── wrangler.jsonc                  # Worker configuration
└── src/
  ├── index.ts                    # OAuth server flow and MCP server routing
  ├── mcp.ts                      # Stateful MCP server as a Cloudflare McpAgent (durable object)
  ├── config.ts                   # Worker bindings and Cloudflare durable object session data
  ├── todoApi.client.ts           # HTTP client to the downstream Todo API
  └── auth/
    ├── workers-oauth-utils.ts   # Cloudflare OAuth utility functions
    ├── ping-handler.ts          # Endpoints that connect the auth flow between OAuth provider and PingOne
    ├── ping-utils.ts            # PingOne OAuth utility functions
    └── ping-types.ts            # PingOne OAuth types
```

Tasks

Configuring the MCP server as an OIDC Client in PingOne

This step enables the Cloudflare worker to exchange authorization codes on behalf of the end user. Note that the MCP server, not the MCP client, receives the PingOne access token. The MCP server then issues a separate reference token to the MCP client for session lookups. This distinction ensures proper audience scoping: the MCP client token is intended for the MCP server, while the MCP server token is intended for the target API.

1. In the PingOne admin console, go to **Applications** and click the + icon to add a new application.
2. In the **Application Type** section, click **OIDC Web App** and then click **Save**.
3. On the **Configuration** tab of the application, ensure that:
    1. **Grant Type** is set to **Authorization Code**.
    2. **PKCE Enforcement** is set to **S256_REQUIRED**.
    3. **Redirect URI** is set to your Cloudflare worker's callback endpoint, for example `<mcp_server>/callback`.
    If the worker is not yet deployed, use a placeholder and update it later.
4. On the **Resources** tab of the application, allow the standard OIDC scopes, `openid` and `profile`, and the Todo API scopes, `todo_api:read` and `tode_api:write`.
5. On the **Policies** tab of the application, enable **Single_Factor** or your preferred policy.

Deploying to Cloudflare

1. In your terminal, install dependencies and build:

    ```
    npm install
    npm run build
    ```

2. Use the Wrangler CLI to set the following remote environment variables:

Name	Description	Example
`PINGONE_ISSUER`	PingOne environment domain	`https://auth.pingone.<REGION>/<ENV_ID>/as`
`MCP_SERVER_CLIENT_ID`	ID of the MCP server client	`0c24f3a0-0522-4f76-9bcf-89643029e3e0`
`MCP_SERVER_CLIENT_SECRET`	Secret of the MCP server client	`[A long, random, alphanumeric string]`
`API_IDENTIFIER`	ID of the downstream Todo API resource	`https://todo.api.com`
`API_URL`	URL of the downstream Todo API	`https://todo-api-ping-aic.<ENV>.workers.dev`
`COOKIE_ENCRYPTION_KEY`	Key used to sign browser cookies	`[A long, random, base64 string]`

```
wrangler secret put PINGONE_ISSUER
wrangler secret put MCP_SERVER_CLIENT_ID
wrangler secret put MCP_SERVER_CLIENT_SECRET
wrangler secret put API_IDENTIFIER
wrangler secret put API_URL
wrangler secret put COOKIE_ENCRYPTION_KEY
```

3. Configure remote KV storage.

```
wrangler kv namespace create OAUTH_KV
```

> **Note**
>
> After running this command, update `wrangler.jsonc` with the generated KV namespace ID.

4. Deploy to Cloudflare.

```
npm run deploy
```

Testing the MCP Server with the MCP Inspector

The MCP Inspector is a developer tool that allows you to test and debug MCP servers by simulating a client connection. This enables you to validate the authentication flow and tool execution interactively. It confirms that the Cloudflare OAuth Provider successfully captures user consent locally before delegating identity verification to PingOne.

1. Launch the Inspector.

```
npx @modelcontextprotocol/inspector
```

The Inspector starts on port 6277 and initiates the authentication flow. No CORS rules are needed because authentication occurs server-to-server (MCP server to PingOne), bypassing browser restrictions.

Authentication flow

1. The Inspector initiates a request to the MCP endpoint (`<mcp_server>/mcp`).
2. The MCP server intercepts the request and presents the local, worker-hosted consent dialog.
3. Upon approval, the MCP server redirects the user to PingOne for authentication.
4. The MCP server exchanges the returned authorization code for an downstream access token.
5. The server binds this downstream token to a new, isolated client session.
6. The server establishes the connection and issues a session handle to the Inspector.

2. Verify that the MCP server is working properly by using the Inspector to confirm the following behaviour:
   - Connection initiates the full Cloudflare consent and PingOne login sequence.
   - Reconnecting (without clearing cookies and using the same MCP client ID) recovers the existing session silently.
   - Clearing browser cookies forces a new consent and authentication cycle upon reconnect.
   - The server correctly populates the tool list in the Inspector.
   - The `whoAmI` tool returns the PingOne access token audienced for the Todo API.
   - Downstream API actions (adding/deleting todos) complete successfully.

Accessing the remote MCP server from Claude Desktop

1. In Claude Desktop, click your profile icon, and then click **Settings**.
2. Go to **Connectors** and then click **Add Custom Connector**.
3. In the **Remote MCP server URL** field, enter the URL in this format: `https://remote-mcp-ping-federate.<ENV>.workers.dev/mcp`

   No OAuth Client ID or Secret is required, as Claude will perform Dynamic Client Registration.
4. Click **Save**.
5. Connect to the MCP server and authenticate with PingFederate.
6. After they're connected, you can ask Claude: "Can you tell me what is in my Todo list?"
7. Claude sees the connected tools and calls the appropriate tool after asking for consent.

{{{ Video removed }}}

Secure a Cloudflare Workers MCP server with PingOne Advanced Identity Cloud

In this configuration, the Cloudflare Workers Model Context Protocol (MCP) server delegates authentication to PingOne Advanced Identity Cloud. This enables clients, such as AI agents, to call a protected API on behalf of an authenticated end user.

Before you begin

This tutorial uses the `remote-mcp-pingone-aic` directory in the Ping Identity cloudflare-mcp Git repository.

Stack

Role	Name	Description
Platform	Cloudflare Workers	Serverless execution
Framework	Hono	Lightweight API endpoints
Agent Execution	Cloudflare Agents SDK	Base class for implementing the stateful MCP server
Session State	Cloudflare Durable Objects	Provides stateful, isolated storage for each MCP connection

Requirements

- Node.js v20 or later
- A PingOne Advanced Identity Cloud tenant
- A Cloudflare account and Wrangler CLI enabled
- Todo API deployed

Structure

```
mcp/
├── package.json              # Dependencies and scripts
├── tsconfig.json             # TypeScript compiler settings
├── wrangler.jsonc            # Worker configuration
└── src/
  ├── index.ts              # Defines the HTTP interface, handling MCP server discovery and MCP server
routing
  ├── mcp.ts                # Stateful MCP server as a cloudflare McpAgent (durable object)
  ├── config.ts             # Worker bindings and durable object session data
  ├── auth.ts               # Manages auth middleware and executes token exchange (delegation grant)
  └── todoApi.client.ts     # HTTP client to the downstream Todo API
```

Tasks

Configuring PingOne Advanced Identity Cloud

1. Enable Dynamic Client Registration (DCR) for MCP client onboarding

This step allows MCP clients to automatically register themselves as public OpenID Connect (OIDC) clients and request the necessary scopes to call the downstream Todo API. The MCP client uses the MCP server's protected resource metadata to determine how to register and which scopes to request.

1. In the PingOne Advanced Identity Cloud **Access Management** console, go to Services > OAuth2 Provider.
2. On the **Client Dynamic Registration** tab, enable **Allow Open Dynamic Client Registration**.
3. On the **Advanced** tab, allow the following Todo API scopes:
    - `todo_api:read`
    - `todo_api:write`

2. Configure MCP server for token exchange

This step configures delegation and authorizes the MCP server to act on the end user's behalf. This allows the MCP server to swap the end user's token for a new token used to call the downstream Todo API. This new token is properly audienced for the target API and includes the `act` claim, identifying the MCP server as the trusted intermediary.

1. Enable grant types and scopes:
    1. From the PingOne Advanced Identity Cloud **Access Management** console, go to Applications > OAuth2 > Clients.
    2. Create a new client for the MCP server.
    3. On the **Core** tab, set the scopes to maximum allowable permissions.
    4. On the **Advanced** tab, add the **Client Credentials** and **Token Exchange** grant types to allow the MCP server to request its own token and perform the delegation, respectively.
2. Flag eligible tokens for token exchange with the `may_act` claim:

    Run a script upon user token issuance to inject a claim designating the MCP server as the trusted delegate. Because DCR prevents you from knowing the MCP client's client ID, the script will be applied to all tokens but include safeguards to only target access tokens intended for the MCP server.

    1. In the PingOne Advanced Identity Cloud **Access Management** console, go to Services > OAuth2 Provider > Scripts.
    2. Create a new script of type **OAuth2 May Act**.
    3. Copy and paste the following script, replacing placeholder values with your configuration values:

```
(function () {
  var frJava = JavaImporter(org.forgerock.json.JsonValue);
  var normalizeUrl = function(url) { return url.replace(/\/$/, '') };

  var mcpServerClientId = 'MCP_SERVER_CLIENT_ID';  // placeholder
  var mcpServerUrl = 'URL_OF_DEPLOYED_MCP_SERVER'; // placeholder

  try {
    var requestedResourceList = requestProperties.requestParams.get('resource');
    var requestedResource = String(requestedResourceList.get(0));

    // Only modify tokens intended for the MCP server
    if (normalizeUrl(requestedResource) === normalizeUrl(mcpServerUrl)) {

      // Explicitly set the MCP server as the audience of this token
      token.setField("aud", mcpServerUrl);

      // Authorize MCP server to exchange this token
      var mayAct = frJava.JsonValue.json(frJava.JsonValue.object());
      mayAct.put('client_id', mcpServerClientId);
      mayAct.put('sub', mcpServerClientId);
      token.setMayAct(mayAct);
    };
  } catch (error) {};
}());
```

4. In the PingOne Advanced Identity Cloud **Access Management** console, go to Services > OAuth2 Provider.
5. On the **Core** tab, set the **OAuth2 Access Token May Act Script**.

3. Control exchanged token audience with an access token modification script.

   This step ensures the downstream API accepts the exchanged token. Because Advanced Identity Cloud doesn't automatically set the necessary audience claim during token exchange, use an access token modification script tied specifically to the MCP server's client profile.

   1. In the Advanced Identity Cloud **Access Management** console, go to Services > OAuth2 Provider > Scripts.
   2. Create a new script of type **OAuth2 Access Token Modification (Next Gen)**.
   3. Copy and paste the following script, replacing the placeholder value with your downstream API URL:

```
(function () {
  var frJava = JavaImporter(org.forgerock.json.JsonValue);
  var normalizeUrl = function(url) { return url.replace(/\/$/, '') };

  // Only modify MCP server access tokens granted with token exchange
  var grantType = requestProperties.requestParams.get('grant_type');
  if (!grantType || grantType.toString() !== '[urn:ietf:params:oauth:grant-type:token-exchange]') {
    return;
  };

  // Only allow MCP server to call these APIs
  var whitelistedAPIs = [
    'URL_OF_DEPLOYED_API', // placeholder value
  ];

  try {
    var requestedResourceList = requestProperties.requestParams.get('resource');
    var requestedResource = JSON.stringify(requestedResourceList).slice(2, -2);

    for (var i = 0; i < whitelistedAPIs.length; i++) {
      var whitelistedApi = normalizeUrl(whitelistedAPIs[i])
      if (normalizeUrl(requestedResource) === whitelistedApi) {
        // Resulting token will be audienced for the requested & whitelisted API
        accessToken.setField("aud", whitelistedApi);
        break;
      };
    };
  } catch (error) {};
}());
```

3. Setup Cross-Origin Resource Sharing (CORS)

For external MCP clients to communicate with Advanced Identity Cloud during the OAuth flow, explicitly allow the MCP inspector.

1. In the PingOne Advanced Identity Cloud dashboard, go to Tenant Settings > Global Settings.
2. Click **Cross-Origin Resource Sharing (CORS)**, and then **Add a CORS Configuration**.
3. Configure the rule for the local MCP Inspector using the following values:
    1. In the **Accepted Origins** field, enter `http://localhost:6277`.
    2. In the **Accepted Methods** field, enter `GET` and `POST`.
    3. In the **Accepted Headers** field, enter `authorization` and `content-type`.
4. Click **Save CORS Configuration**.

Deploying to Cloudflare

1. Install dependencies and deploy the MCP server:

```
npm install
npm run deploy
```

2. Use the Wrangler CLI to configure the remote environment variables:

Name	Description	Example
PING_AIC_ISSUER	PingOne Advanced Identity Cloud environment domain	`https://<ENV>.forgeblocks.com:443/am/oauth2/alpha`
MCP_SERVER_URL	URL of the deployed MCP server	`https://remote-mcp-ping-aic.<ENV>.workers.dev`
MCP_SERVER_CLIENT_ID	ID of the MCP server client	`mcp_server`
MCP_SERVER_CLIENT_SECRET	Secret of the MCP server client	`[A long, random, alphanumeric string]`
API_URL	URL of the downstream Todo API	`https://todo-api-ping-aic.<ENV>.workers.dev`

```
wrangler secret put PING_AIC_ISSUER
wrangler secret put MCP_SERVER_URL
wrangler secret put MCP_SERVER_CLIENT_ID
wrangler secret put MCP_SERVER_CLIENT_SECRET
wrangler secret put API_URL
```

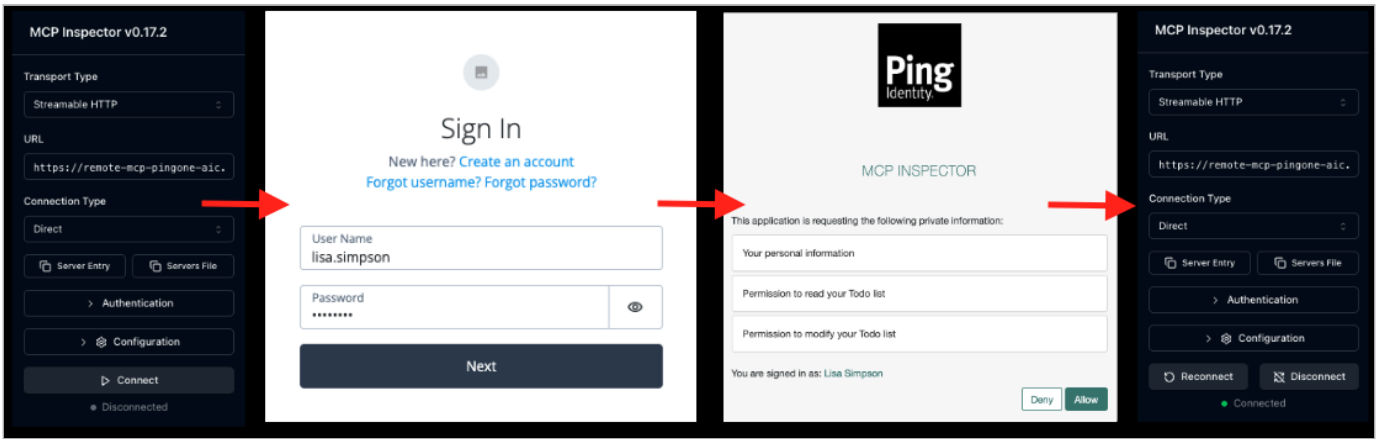
3. Ensure the `mcpServerUrl` is correct in the Advanced Identity Cloud `may_act` script that you created in step 2.2.

Testing the MCP server with MCP Inspector

1. To validate the deployed MCP server, launch MCP Inspector:

```
npx @modelcontextprotocol/inspector
```

MCP Inspector starts on port 6277, which you allowed in the CORS configuration. To confirm that DCR is operational, the Inspector acts as an unregistered client and establishes a secure connection.



Authentication flow

1. The Inspector attempts an initial, unauthenticated request to the MCP endpoint (`<mcp_server>/mcp`).
2. The MCP server responds with the protected resource metadata, triggering the discovery process.
3. The MCP inspector consumes the metadata, performs discovery of the authorization server, and self-registers with Advanced Identity Cloud as a public OIDC client.
4. The MCP Inspector initiates the OAuth authorization flow, securing the necessary user consent and establishing a stateful MCP connection.

2. To validate token exchange, observe how the MCP server acquires a new token audienced for the downstream API. Use the `who_am_I` and `peek_api_token_claims` tools to compare the two tokens and confirm the following behavior:
   - Both tokens share the same `sub` claim, confirming they identify the same authenticated end user.
   - The subject token contains the `may_act` claim, explicitly authorizing the MCP server for exchange. The resulting delegation token contains the `act` claim, identifying the MCP server as the trusted intermediary acting on the user's behalf.
   - The subject token is audienced for the MCP server, while the delegation token is audienced for the downstream API.
   - The subject token retains the user's full, broader permissions. The delegation token is limited to only whats necessary for its target audience.

Accessing the remote MCP server from Claude Desktop

1. In Claude Desktop, click your profile icon, and then click **Settings**.
2. Go to **Connectors** and then click **Add Custom Connector**.
3. In the **Remote MCP server URL** field, enter the URL in this format: `https://remote-mcp-ping-federate.<ENV>.workers.dev/mcp`

   No OAuth Client ID or Secret is required, as Claude will perform Dynamic Client Registration.
4. Click **Save**.
5. Connect to the MCP server and authenticate with PingFederate.
6. After they're connected, you can ask Claude: "Can you tell me what is in my Todo list?"
7. Claude sees its connected tools and calls the appropriate tool after asking for consent.

{{{ Video removed }}}

Secure a Cloudflare Workers MCP server with PingOne DaVinci

In this configuration, the Cloudflare Workers Model Context Protocol (MCP) server uses the Cloudflare Workers OAuth Provider, which delegates authentication to PingOne DaVinci. This enables clients, such as AI agents, to call a protected API on behalf of an authenticated end user.

> **Important**
>
> While serving as a resource server for MCP clients, this MCP server also fulfills two distinct OAuth roles:
>
> - Acts as an OAuth Server to your MCP clients
> - Acts as an OpenID Connect (OIDC) Client to your PingOne environment

This configuration uses DaVinci to manage consent. To use this configuration with Cloudflare-managed consent, see Secure a Cloudflare Workers MCP server with PingOne.

Before you begin

This tutorial uses the `remote-mcp-pingone/mcp-dv` directory in Ping Identity's cloudflare-mcp Git repository.

Stack

Role	Name	Description
Platform	Cloudflare Workers	Serverless execution
Framework	Hono	Lightweight API endpoints
Agent Execution	Cloudflare Agents SDK	Base class for implementing the stateful MCP server
Session State	Cloudflare Durable Objects	Provides stateful, isolated storage for each MCP connection
OAuth Core	Cloudflare Workers OAuth Provider	Orchestrates the OAuth flow, delegating authentication to PingOne
Ephemeral State	Cloudflare Workers KV	Stores OAuth state required by the workers oauth provider

Requirements

- Node.js v20 or later
- A PingOne environment
- A Cloudflare account and Wrangler CLI enabled
- Todo API deployed
- MCP Inspector installed

Structure

```
mcp-dv/
├── package.json                    # Dependencies and scripts
├── tsconfig.json                   # TypeScript compiler settings
├── wrangler.jsonc                  # Worker configuration
└── src/
  ├── index.ts                    # OAuth server flow and MCP server routing
  ├── mcp.ts                      # Stateful MCP server as a Cloudflare McpAgent (durable object)
  ├── config.ts                   # Worker bindings and Cloudflare durable object session data
  ├── todoApi.client.ts           # HTTP client to the downstream Todo API
  └── auth/
    ├── workers-oauth-utils.ts   # Cloudflare OAuth utility functions
    ├── ping-handler.ts          # Endpoints that connect the auth flow between OAuth provider and PingOne
    ├── ping-utils.ts            # PingOne OAuth utility functions
    └── ping-types.ts            # PingOne OAuth types
```

Tasks

Configuring the MCP server as an OIDC client in PingOne

This step enables the Cloudflare worker to exchange authorization codes on behalf of the end user. Note that the MCP server, not the MCP client, receives the PingOne access token. The MCP server then issues a separate reference token to the MCP client for session lookups. This distinction ensures proper audience scoping: the MCP client token is intended for the MCP server, while the MCP server token is intended for the target API.

1. In the PingOne admin console, go to **Applications** and click the + icon to add a new application.
2. In the **Application Type** section, click **OIDC Web App** and then click **Save**.
3. On the **Configuration** tab of the application, ensure that:
    1. **Grant Type** is set to **Authorization Code**.
    2. **PKCE Enforcement** is set to **S256_REQUIRED**.
    3. **Redirect URI** is set to your Cloudflare worker's callback endpoint, for example `<mcp_server>/callback`. If the worker is not yet deployed, use a placeholder and update it later.
4. On the **Resources** tab of the application, allow the standard OIDC scopes, `openid` and `profile`, and the Todo API scopes, `todo_api:read` and `tode_api:write`.
5. On the **Policies** tab of the application, leave the policy blank for now. You'll configure this in a later step.

Creating a Worker application to handle MCP server consent

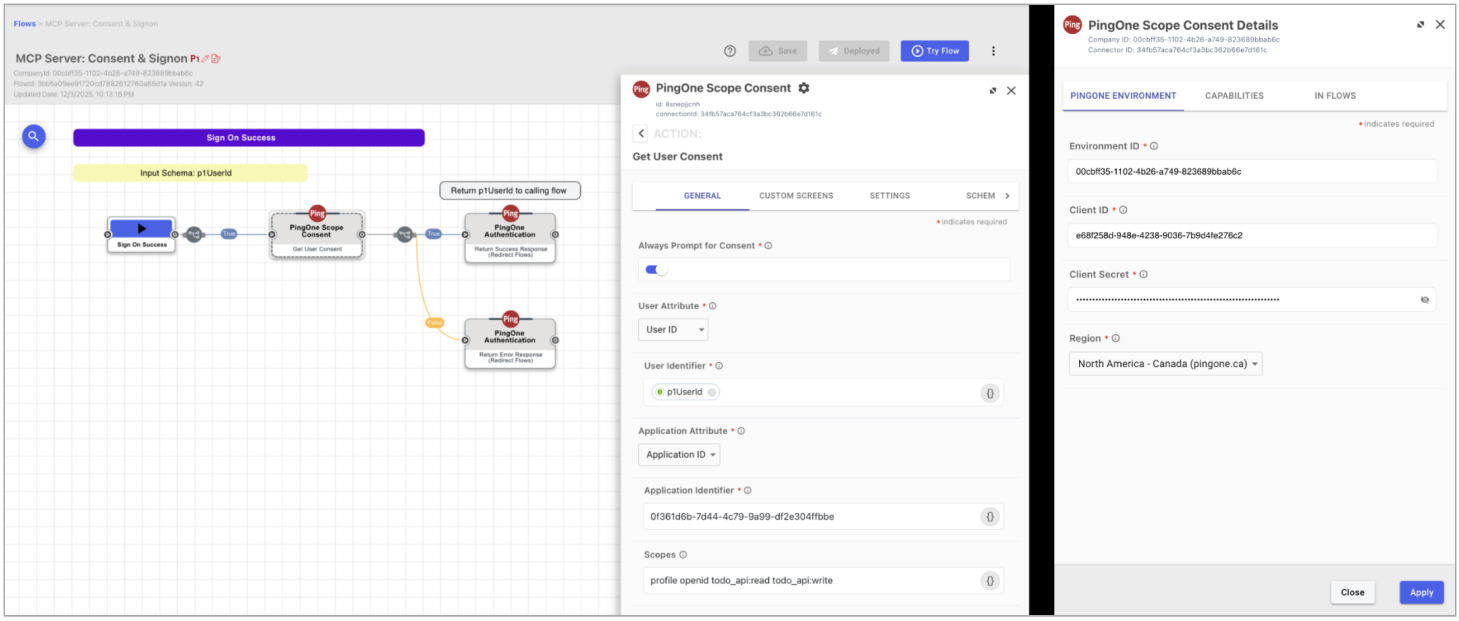
1. In the PingOne admin console, go to **Applications** and click the + icon to add a new application.
2. Select **Worker** and click **Save**.
3. On the **Overview** tab of the application, note the **Client ID** and **Client Secret**.
4. On the **Configuration** tab, set **Token Auth Method** to **Client Secret Basic**.

5. On the **Roles** tab, assign the **Environment Admin** and **Identity Data Admin** roles to the application to authorize it to manage user consent.

Creating the DaVinci flow using the MCP server OIDC client and the consent service worker

Create the DaVinci flow that orchestrates both authentication and consent for the MCP server. You'll modify a standard PingOne sign-on flow to include the consent logic.

1. In DaVinci, add the PingOne Scope Consent connector and configure it using the client ID and client secret from the consent service worker.
2. Go to **Flows** and clone the **PingOne Sign On with Registration, Password Reset and Recovery** flow.
3. Locate the **Sign on Success** node and add a **Get User Consent** node immediately after it.
4. Configure the **Get User Consent** node by entering the application ID of the MCP server OIDC Client.
5. Configure the PingOne authentication terminal nodes and make sure the flow is a PingOne flow in the flow settings.



Creating a DaVinci policy for the DaVinci flow

Add the DaVinci flow to an application and create a flow policy to control how and when the flow gets used.

1. In DaVinci, go to **Applications** and create a new application.
2. Add a PingOne flow policy to the application and target the DaVinci flow created in task 3.

Binding the DaVinci policy to the MCP server client profile

This ensures that when the MCP server initiates an OAuth request, PingOne routes the user through the DaVinci flow to capture the required consent.

1. In the PingOne admin console, select the MCP server client created in task 1.
2. On the **Policies** tab, add the DaVinci Policy created in task 4.

Deploying to Cloudflare

1. In your terminal, install dependencies and build.

```
npm install
npm run build
```

2. Use the Wrangler CLI to set the following remote environment variables:

Name	Description	Example
`PINGONE_ISSUER`	PingOne environment domain	`https://auth.pingone.<REGION>/<ENV_ID>/as`
`MCP_SERVER_CLIENT_ID`	ID of the MCP server client	`0c24f3a0-0522-4f76-9bcf-89643029e3e0`
`MCP_SERVER_CLIENT_SECRET`	Secret of the MCP server client	`[A long, random, alphanumeric string]`
`API_IDENTIFIER`	ID of the downstream Todo API resource	`https://todo.api.com`
`API_URL`	URL of the downstream Todo API	`https://todo-api-ping-aic.<ENV>.workers.dev`
`COOKIE_ENCRYPTION_KEY`	Key used to sign browser cookies	`[A long, random, base64 string]`

```
wrangler secret put PINGONE_ISSUER
wrangler secret put MCP_SERVER_CLIENT_ID
wrangler secret put MCP_SERVER_CLIENT_SECRET
wrangler secret put API_IDENTIFIER
wrangler secret put API_URL
wrangler secret put COOKIE_ENCRYPTION_KEY
```

3. Configure remote KV storage.

```
wrangler kv namespace create OAUTH_KV
```

> **Note**
>
> After running this command, update `wrangler.jsonc` with the generated KV namespace ID.

4. Deploy to Cloudflare.

```
npm run deploy
```

Testing the MCP server with MCP Inspector

MCP Inspector is a developer tool that allows you to test and debug MCP servers by simulating a client connection. This enables you to validate the authentication flow and tool execution interactively. It confirms that the Cloudflare OAuth Provider successfully captures user consent locally before delegating identity verification to PingOne.

1. Launch the Inspector.

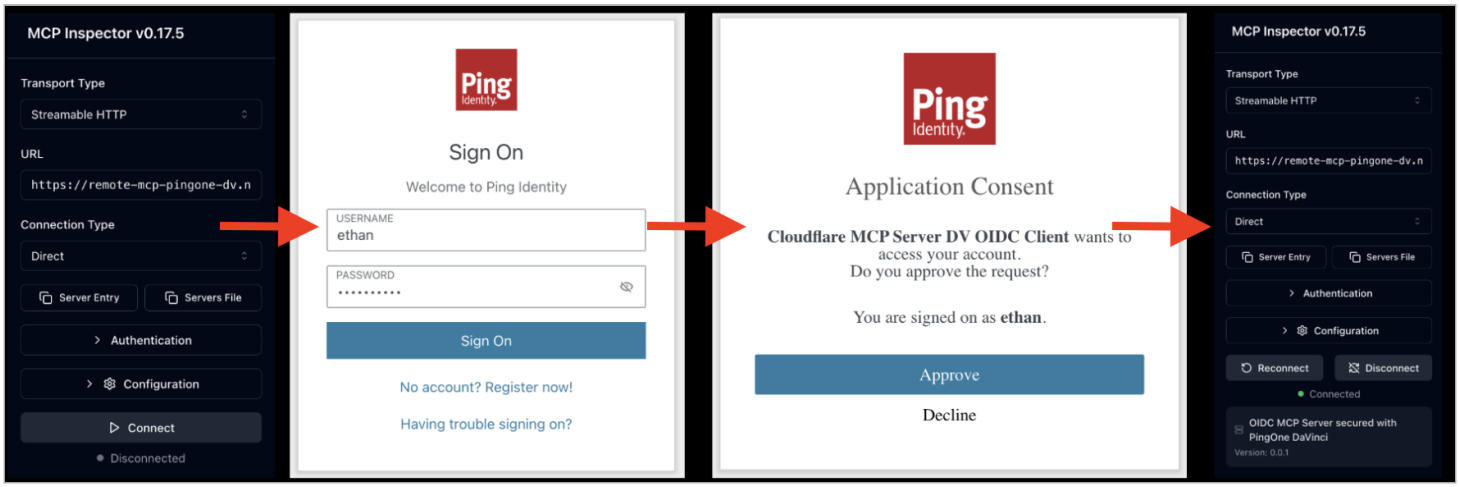
   ```
   npx @modelcontextprotocol/inspector
   ```

   The Inspector starts on port 6277 and initiates the authentication flow. No CORS rules are needed because authentication occurs server-to-server (MCP server to PingOne), bypassing browser restrictions.

   Authentication flow

   1. The Inspector initiates a request to the MCP endpoint (`<mcp_server>/mcp`).
   2. The MCP server intercepts the request and presents the local, worker-hosted consent dialog.
   3. Upon approval, the MCP server redirects the user to PingOne for authentication.
   4. The MCP server exchanges the returned authorization code for an downstream access token.
   5. The server binds this downstream token to a new, isolated client session.
   6. The server establishes the connection and issues a session handle to the Inspector.

2. Verify that the MCP server is working properly by using the Inspector to confirm the following behaviour:
   - Connection initiates the full Cloudflare consent and PingOne login sequence.
   - Reconnecting (without clearing cookies and using the same MCP client ID) recovers the existing session silently.
   - Clearing browser cookies forces a new consent and authentication cycle upon reconnect.
   - The server correctly populates the tool list in the Inspector.
   - The `whoAmI` tool returns the PingOne access token audienced for the Todo API.
   - Downstream API actions (adding/deleting todos) complete successfully.



Accessing the remote MCP server from Claude Desktop

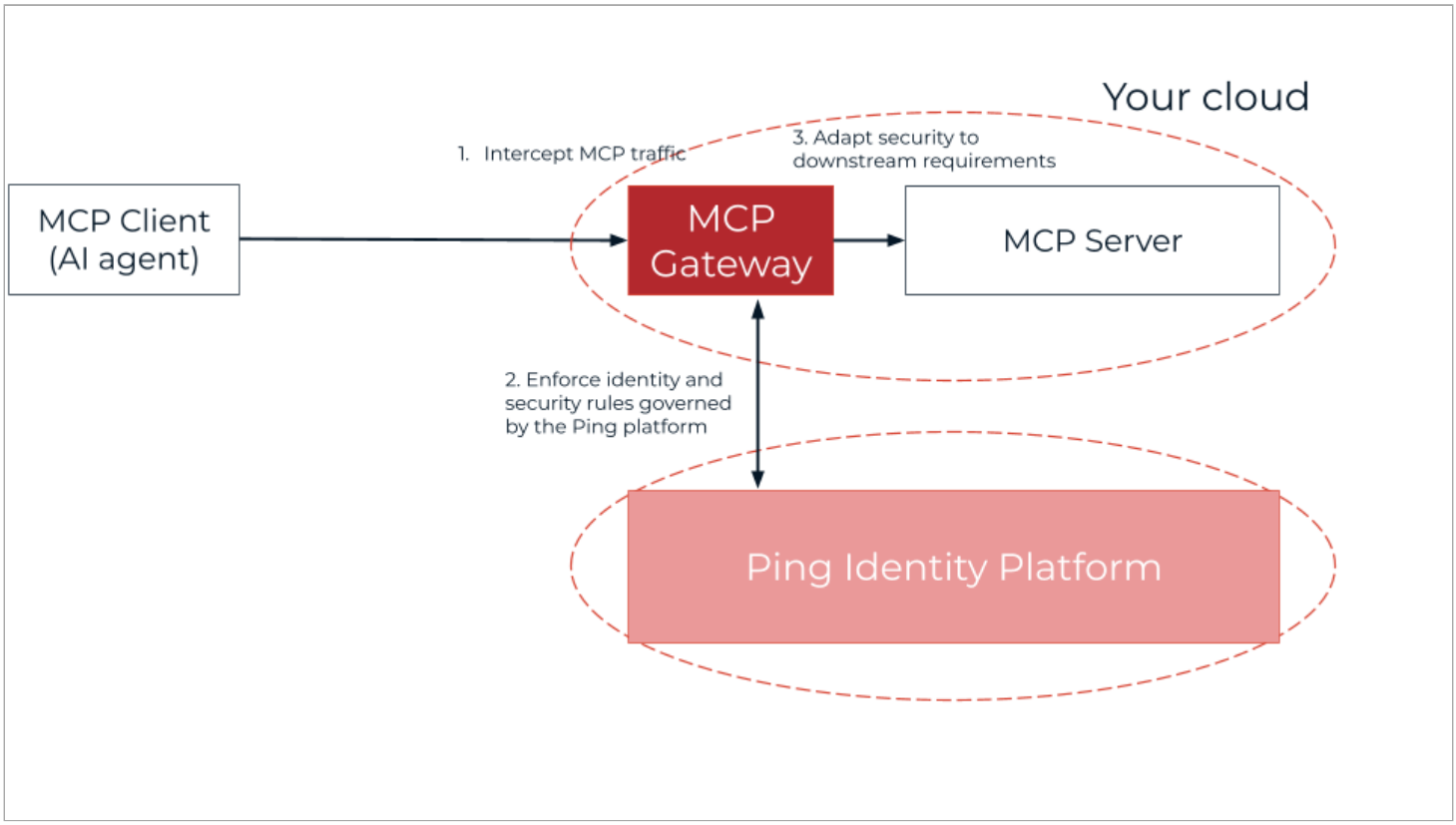
1. In Claude Desktop, click your profile icon, and then click **Settings**.
2. Go to **Connectors** and then click **Add Custom Connector**.
3. In the **Remote MCP server URL** field, enter the URL in this format: `https://remote-mcp-ping-federate.<ENV>.workers.dev/mcp`

   No OAuth Client ID or Secret is required, as Claude will perform Dynamic Client Registration.
4. Click **Save**.
5. Connect to the MCP server and authenticate with PingFederate.
6. After they're connected, you can ask Claude: "Can you tell me what is in my Todo list?"
7. Claude sees the connected tools and calls the appropriate tool after asking for consent.

{{{ Video removed }}}

Secure an MCP server with PingGateway

When securing any MCP server, implementing an appropriate, consistent, documented, auditable, and adaptable security model can be challenging.



In this architecture, PingGateway:

1. Intercepts and validates an MCP request from an AI agent to an MCP server. It optionally audits and throttles traffic.
2. Authorizes the AI agent request using OAuth 2.0.
3. Protects the MCP server by enforcing OAuth 2.0 scopes. It optionally acts as a policy decision point and transforms security tokens.

PingGateway addresses the challenges in protecting any MCP server by providing a unified layer to:

- Allow only valid MCP requests.
- Audit MCP requests and actors.
- Throttle request rates.
- Enforce coarse-grained OAuth 2.0 security controls.
- Enforce fine-grained access control policies.
- Perform token transformation mapped to your security models.

The MCP security gateway tutorial in the PingGateway documentation shows how to protect any MCP server with PingOne Advanced Identity Cloud acting as the OAuth 2.0 authorization server.

What is Agent2Agent Protocol (A2A)?

Whereas Model Context Protocol (MCP) defines how agents interact with tools, Google's Agent2Agent Protocol (A2A) defines how agents interact with each other.

In an ecosystem where agents exist in different domains and are built upon different frameworks, A2A defines a standardized way for agents to collaborate as autonomous peers. In practice, this means an HR agent could securely call out to a payroll agent to adjust a salary, or a travel agent could ask a calendar agent to reschedule meetings, all under a trusted, reliable framework.

Why use A2A?

To make agent-to-agent communication as streamlined as possible, A2A is designed around the following core principles:

Discovery: Agents can find other agents with the desired permissions and capabilities.

Interoperability: Agents can pass goals, intents, and structured requests regardless of which framework they're built on.

Delegation and Chaining: Agents can hand off tasks to better-suited agents without the end user needing to manually orchestrate it.

Trust and Identity: As with tools, agents need to authenticate with each other. A2A defines how identity and permissions are enforced when agents collaborate.

How A2A works

An A2A interaction involves the following components:

Agent card

A discoverable JSON file that acts as an agent's public profile. It advertises the agent's identity, capabilities, supported data modalities, and authentication requirements.

This allows other agents to determine the most suitable agent to interact with.

Task

The fundamental unit of work. A task has a unique ID and progresses through a defined lifecycle, such as working, completed, or input required.

Message and artifact

Messages are the units of communication exchanged between agents to manage a task, while artifacts are the final, tangible outputs generated upon task completion.

Workflow

The typical A2A workflow involves a client agent discovering an agent card, authenticating as required, and then monitoring tasks through a series of API exchanges until the remote agent delivers the artifacts to the end user.

1. A client agent, acting on behalf of the user, discovers an agent card. This could be through open discovery, a curated registry, or private API endpoints.
2. The client agent authenticates and receives a token from the remote agent's authorization server.

3. The client agent invokes the `POST /sendMessage` API to send details about the task along with its access token to the remote agent.

   The remote agent can respond with a `Message` object to negotiate the scope of a request or with a `Task` object if the agent accepts the task request, containing the `contextId` and `taskId`.

4. The client agent invokes the `POST /sendMessageStream` API with the remote agent for real-time updates of long-running tasks using Server-Sent Events (SSE).

   When the task is complete, the remote agent responds with the artifacts and the `taskId` in completed status.

Learn more

Use the following resources to learn more about implementing A2A:

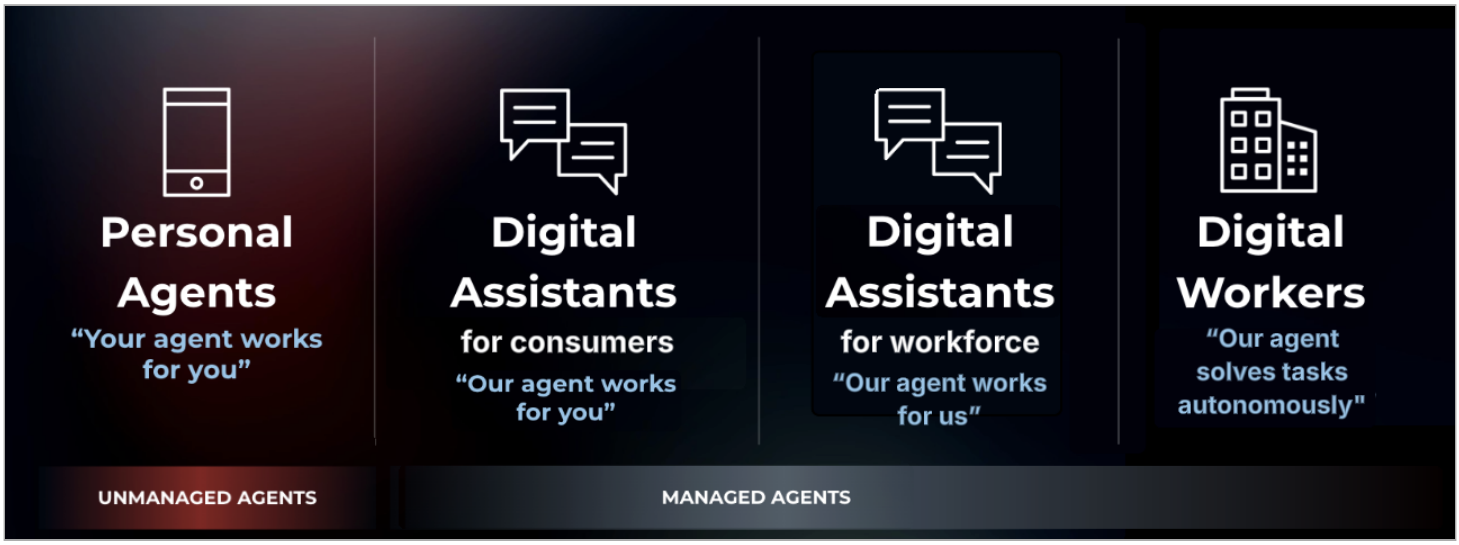
- Agent2Agent (A2A) Protocol
- Tutorials and Samples

Together, MCP defines how agents call tools, while A2A defines how agents call each other. Both rely on strong Identity for AI to ensure that every request, whether to a tool or to another agent, is authenticated, authorized, and auditable. Learn more about how MCP and A2A interact in A2A and MCP: Complementary Protocols for Agentic Systems.

Agent Types

Artificial intelligence (AI) agents open up a new channel for businesses to engage with customers. As multiple agents collaborate to complete requests, understanding each agent type is critical for ensuring security and maintaining trust.

Not all agents are equal, and the identity implications vary depending on who manages the agent, where the agent is deployed, and the purpose of the agent.



Ping Identity categorizes agents into four types, each serving a distinct use case with its own trust and security considerations:

- Personal agents
- Digital assistants for consumers
- Digital assistants for workforce
- Digital workers

These types can also be classified based on the level of control you have:

Unmanaged agents

Agents that operate outside your organization's trust boundary, such as a user's personal agent. These agents interact with your services but aren't governed or controlled by your organization.

Managed agents

Agents that your organization creates and controls. You can monitor their activity, enforce policies, and grant access to sensitive resources as needed. Managed agents include digital assistants and digital workers.

In the following sections, we'll discuss what distinguishes each agent type and the identity challenges associated with each. We'll also explore how Ping Identity brings together existing and new identity and access management (IAM) concepts to secure agentic identities.

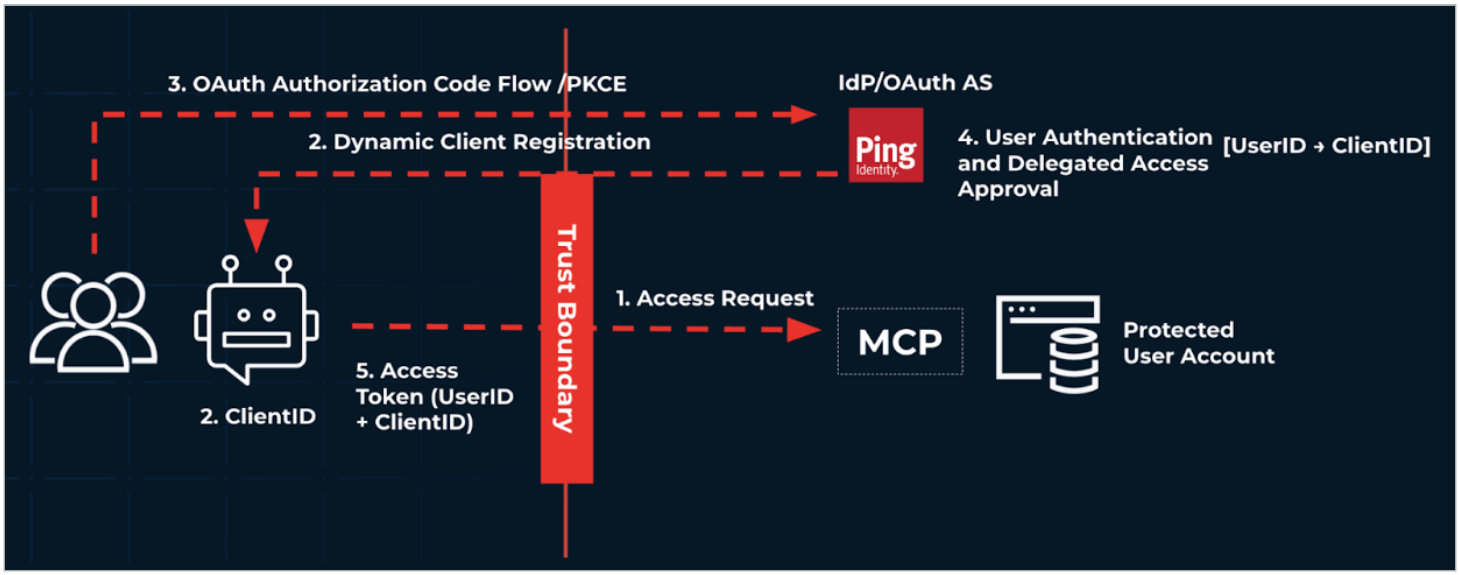
Personal agents

A personal agent can be accessed on your device and perform a wide array of tasks for you. For example, you might prompt ChatGPT to aggregate the best flight deals between airlines.

Personal agents are unmanaged agents that are governed by a third party.

In this case, the agent lies outside of your organization's trust boundary.

The following diagram shows a simplified flow of a personal agent interacting with your services:



Since the agent above could come from anywhere, it's operating outside of the organizational trust boundary. If your organization is an insurance company, the interactions in the flow could resemble the following:

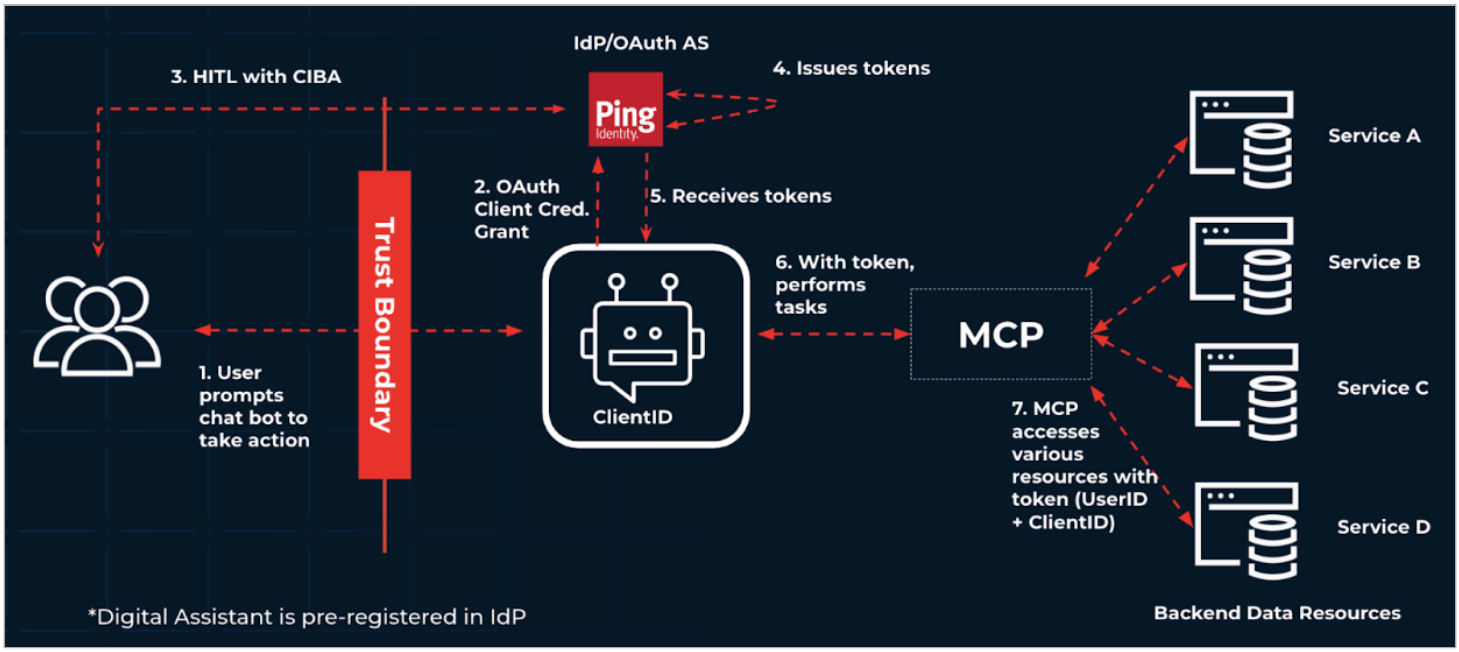
1. A user prompts their personal agent, such as ChatGPT, to shop for the latest deals on car insurance.
2. The agent attempts to access that user's account in your organization to look at their existing policy. Your organization uses Ping Identity as their identity provider and authorization server (AS), and you require the agent to be dynamically registered with the AS and tied to an end user. With the user registered with Ping Identity, the OAuth 2.0 client requires human-in-the-loop (HITL), meaning it requires the user to approve this access.
3. The user completes an authentication journey.
4. After the user authenticates successfully, the AS issues minimally scoped tokens back to the agent.
5. With an access token, the agent can now access the user's current insurance rate information.

Digital assistants for consumers

Digital assistants for consumers are customer-facing agents owned by an organization to help users complete specific tasks, such as a chatbot that assists with booking an appointment.

Unlike a personal agent, a digital assistant is a managed agent that's created and governed by your organization. They interact directly with users, but often require access to protected information to complete user requests.

The following diagram details high-level interactions between the digital assistant for consumers agent, the end user, and an organization's backend resources. It doesn't delve into how to securely integrate Model Context Protocol (MCP) servers just yet.



Compared to the personal agent, the digital assistant is now within the organizational trust boundary. An organization would create their digital assistant agent, and then preregister it using Ping Identity as the IdP and AS.

In this example, your organization is again an insurance company. This time, you want to create a digital assistant chatbot on your website. The chatbot must be able to access the backend resources and APIs required to assist customers while ensuring that:

- The user authenticates and approves the chatbot's actions.
- The chatbot operates with only the permissions granted by the user.
- You can monitor the chatbot's activities (logging and auditing).

The flow is as follows:

1. With the chatbot registered with Ping Identity as an OAuth 2.0 client, the end user prompts the chatbot to change their payments from every 6 months to a monthly basis.
2. The chatbot doesn't have sufficient access to perform this change, so it interacts with Ping Identity as the IdP through the Client Credentials grant flow to gain the necessary tokens.
3. With Ping Identity as the IdP and the AS, the flow requires a HITL interaction to grant tokens. With Client-Initiated Backchannel Authentication (CIBA), the end user is sent a push notification, and they approve the action of the chatbot.
4. Ping Identity issues tokens.
5. The chatbot receives the tokens.
6. The chatbot sends the request to the MCP server with the tokens to access protected resources.
7. The MCP server accesses the backend API to update the user's payment plan.

Digital assistants for workforce

Digital assistants for workforce are similar to digital assistants for consumers. They're agents managed by your organization, they need to access backend resources to perform tasks on behalf of the user, and they'll use MCP and Agent2Agent (A2A) protocol to do so.

However, digital assistants for workforce are targeted internally to increase the efficiency of an organization. For example, an organization might implement a HR digital assistant to help employees submit time-off requests.

Now, both the agent and the user are within the organizational trust boundary, as the end user is an employee within the organization's purview.

Digital workers

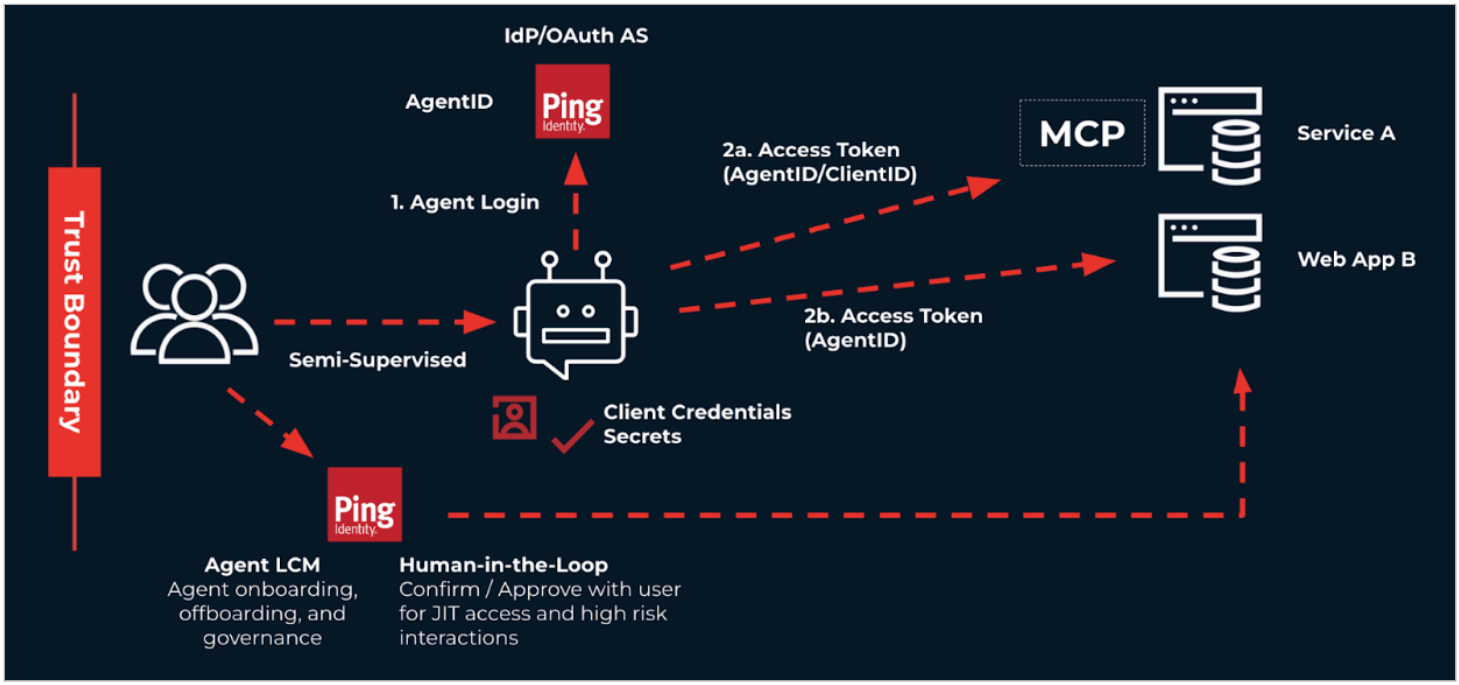
A digital worker is a managed agent that's semi-autonomous to fully autonomous within an enterprise designed to perform internal tasks.

Similar to an intern with a simple set of tasks who checks in periodically to ensure they're on the right path, a digital worker performs tasks on its own and checks in with a human when required.

For example, let's say you create a digital worker to manage inventory and coordinate logistics. This digital worker can reason and understand external factors that influence the quantity and cadence of inventory purchases. The digital worker is completely within your organization's trust boundary. It requires:

- Its own identity and set of credentials to perform actions as itself.
- Depending on business requirements, a human supervisor to approve certain actions.

Consider the following diagram:



1. Just as an end user would, the digital worker signs on with Ping Identity as the IdP and AS.

2. After signing on, the worker receives appropriately scoped tokens to access purchasing and manage inventory.

   The worker also accesses the organization's web app to interact and view the status of delivery on inventory.

3. For high-sensitivity transactions, a HITL interaction is required. Additionally, the worker is monitored just like a human user to ensure that the worker has the minimal access it needs to perform its tasks.

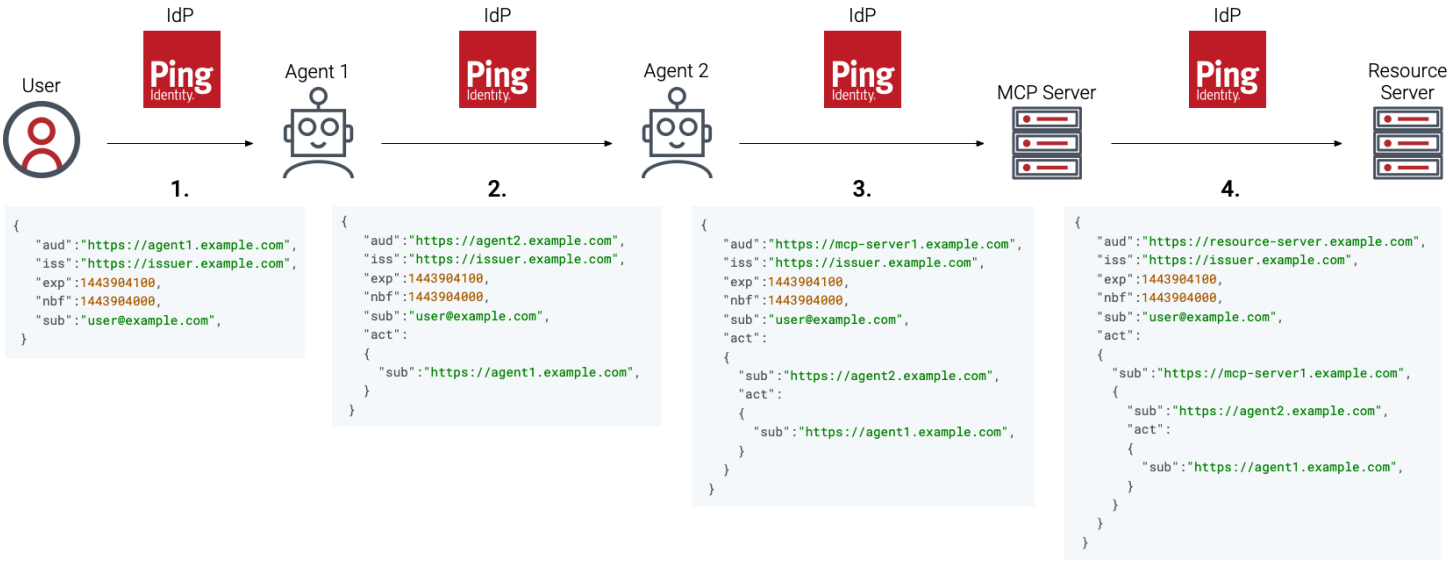
Identifying Agents with Token Exchange

In a standard OAuth flow, an application might impersonate a user, holding a token that allows it to do anything a user can do. This makes the application's behavior indistinguishable from user-performed actions. In complex agentic architectures (where multiple AI agents are collaborating with each other), this is dangerous, especially if rogue agent activity has an inaccurate audit trail.

The OAuth 2.0 Token Exchange protocol (RFC 8693) defines how to apply the concept of delegation to token exchange. In simple terms, an agent's authentication token can show "I'm Agent A, and I'm performing a specific action requested by Agent B, to complete a task for User C."

This approach is superior to others, such as secret vault retrieval, as the agent never possesses live credentials. The agent only possesses a finely-scoped access token after the end user triggers token exchange.

The following graphic illustrates a daisy-chain of these delegations, where multiple agents are working together and communicating with a Model Context Protocol (MCP) server to solve a problem on behalf of the user.



The key to this flow is the `act` (actor) claim within the JSON Web Token (JWT). With each new agent authentication, a new `act` claim is appended, creating a nested audit trail of collaboration between agents and servers.

The following describes each authentication in the flow:

1. User authentication

   The process begins with a human user authenticating with an identity provider (IdP) to use an agent.

   - **The exchange:** Ping Identity (the IdP) issues a JWT.
   - **Key claims:**
     - `sub` (subject): user@example.com (The human).
     - `aud` (audience): https://agent1.example.com (The "orchestrator" agent the human is communicating with).
   - **Meaning:** The user is interacting directly with agent 1, so there is no `act` claim yet.

2. Agent 1 delegates to agent 2

Agent 1 needs to offload a task to agent 2. Agent 1 can't just pass the user's token because that token is scoped for agent 1, not agent 2.

- **The exchange:** Agent 1 contacts the IdP and requests token exchange. It presents the user's token and asks for a new token valid for agent 2.
- **Key claims:**
  - `sub` : user@example.com (Still the user).
  - `aud` : https://agent2.example.com (Now valid for agent 2).
  - The `act` claim:

```
"act":
{
  "sub":"https://agent1.example.com",
}
```

- **Meaning:** Agent 1 is the actor requesting agent 2 to perform a task on behalf of the user.

3. Agent 2 delegates to MCP server

Agent 2 now needs to access a tool or resource through an MCP server.

- **The exchange:** Agent 2 sends the token it received from agent 1 to the IdP and requests a token scoped for the MCP server.
- **Key claims:**
  - `sub` : user@example.com (Still the user).
  - `aud` : https://mcp-server1.example.com (Valid for the MCP server).
  - The nested `act` claim:

```
"act":
{
  "sub":"https://agent2.example.com",
  "act":
  {
    "sub":"https://agent1.example.com",
  }
}
```

- **Meaning:** Now, agent 2 is the actor performing token exchange with the MCP server, having been triggered by agent 1. The user is still the original owner of the request.

4. MCP server calls the resource server

The MCP server now needs to access the resource server to complete the user's task.

- **The exchange:** The MCP server presents its token from agent 2 and requests a new token scoped for the resource server.

- **Key claims:**
  - `sub`: user@example.com (Still the user).
  - `aud`: https://resource-server.example.com (Valid for the resource server).
  - The nested `act` claim:

```
"act":
{
  "sub":"https://mcp-server1.example.com",
  {
    "sub":"https://agent2.example.com",
    "act":
    {
      "sub":"https://agent1.example.com",
    }
  }
}
```

This demonstrates a perfect call stack. Every actor has its own unique token that identifies who it was called by in each stage of the flow. This separates an agent's activity from a human's, and ensures every action is accurately accounted for.

Securing Digital Assistants

Secure your organization's AI-driven digital assistants by treating them as first-class identities, distinct from the humans they act on behalf of, and enforcing least-privilege access at runtime.

AI agents are serving both consumers and employees (for example, as chatbots on retail websites helping users navigate products, and as workforce assistants managing employee calendars and drafting emails). Securing these assistants means verifying not just what the agent is doing, but whose identity and permissions it's exercising at any given moment. Runtime identity distinguishes the agent from the human while ensuring the agent has explicit authority to act on the user's behalf.

Key Concepts

Explicit agent identity

Register every digital assistant as a unique identity with its own credentials and lifecycle. This enables centralized management of your agents, their owners, and their resource permissions from a single console across your customer and workforce use cases.

Delegation, not impersonation

Using OAuth 2.0 token exchange, agents swap a user's token for a delegation token that has an `act` (actor) claim, explicitly identifying both the user and the agent. The actor claim provides a clear audit trail of which agent took what action, enabling enforcement decisions at runtime based on both the user and the agent. The agent never sees the user's credentials.

Least-privilege permissions

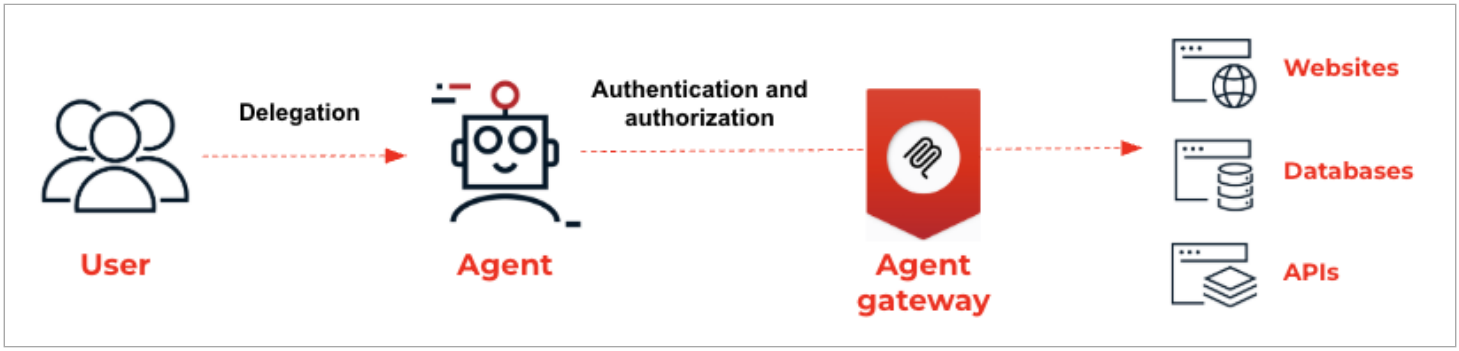
The delegation token is short-lived and might require additional token exchanges as the agent performs new actions or accesses different resources. Audience restriction ensures the token is only valid at a specific resource server, preventing acceptance of unauthorized tokens.

Runtime Enforcement

The agent gateway acts as a security proxy, validating tokens and enforcing fine-grained policies before an agent can reach backend resources such as APIs or Model Context Protocol (MCP) servers. The gateway provides a consistent security layer for AI developers building on MCP or traditional REST architectures.

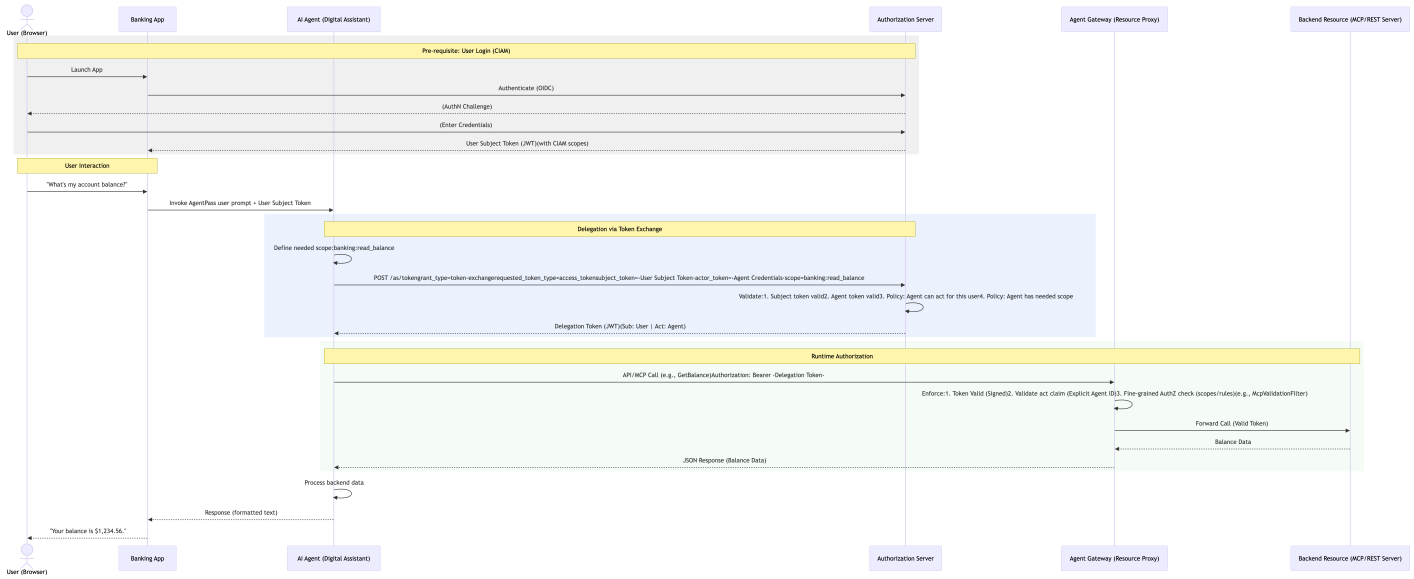
How this works

As an example scenario, consider a chatbot embedded in a banking app. Customers use the chatbot to retrieve account balances and make transactions. Acting on behalf of the customer, the chatbot accesses backend account resources.



To secure these interactions, an authorization server issues and exchanges tokens. An agent gateway validates the tokens, acting as a security proxy in front of account resources.

Example process flow:



1. **User authentication**: The customer signs on to the banking app and receives an initial access token (the subject token).
2. **Request for action**: The customer asks the chatbot for their account balance. The banking app passes the customer's subject token to the chatbot.
3. **Token exchange**: The chatbot sends the customer's token and its own credentials (the actor token) to the authorization server.
4. **Issuance of delegation token**: The authorization server issues a new, short-lived token. This token includes the customer as the `sub` (subject) and the chatbot in the `act` (actor) claim.
5. **Gateway validation**: The chatbot sends the request to the backend. The agent gateway intercepts the call, validates the `act` claim, the user (`sub`), and target resource (`aud`), and ensures the chatbot has the specific scope (`account:read_balance`) required for the task.
6. **Secure execution**: The backend account API processes the request, knowing exactly which agent performed the action for which user.

A delegation token issued by the authorization server carries a precise record of the delegation:

```
{
  "iss": "https://as.example.com",
  "aud": "https://acount.api.example.com",
  "sub": "customer@example.com",
  "act": {
    "sub": "https://bank-agent.example.com"
  },
  "scope": "account:read_balance",
  "iat": 1742200000,
  "exp": 1742200300
}
```

The token contains the following claims:

Claim	Description
`sub`	The human user who authorized the action.
`act.sub`	The AI agent performing the action.
`scope`	Constrained permissions granted to the agent for this action.
`aud`	Restricts the token to a specific downstream resource.

The delegation token provides downstream services with everything they need for authorization decisions: who is the human, who is the agent, what are they allowed to do, and how to trace this specific action.

Ping Identity authorization servers that support this scenario include PingOne, PingOne Advanced Identity Cloud, and Ping Identity software, such as PingFederate and PingAM. PingGateway serves as the agent gateway.

What's next

Get started with your preferred platform:

- Securing AI agents with PingOne using delegation and least privilege
- Securing AI Agents with PingFederate using delegated access tokens

Securing AI agents with PingOne using delegation and least privilege

PingOne PingGateway

Organizations are deploying AI-driven digital assistants to act on behalf of humans. To securely deploy these agents, treat them as first-class, non-human identities distinct from end users and enforce least-privilege access.

As an example scenario, consider a chatbot embedded in a banking app:

- Customers use the chatbot to retrieve account balances and make transactions.
- Acting on behalf of the customer, the chatbot accesses backend account resources.

- To secure these interactions, PingOne issues tokens and exchanges them as necessary, based on the agent's requested access and target resource.
- PingGateway validates the tokens, acting as a security proxy in front of account resources.

You can find more details about the process flow in Securing Digital Assistants.

Goals

- Secure a digital assistant to ensure the agent only performs authorized actions on behalf of the user.
- Configure OAuth 2.0 token exchange to issue downscoped access tokens that preserve the chain of delegation using the `act` claim.
- Protect backend Model Context Protocol (MCP) servers and APIs using PingGateway.

What you'll do

To enable secure delegation, you'll set up PingOne and PingGateway:

- Add custom resources.
- Add a consent agreement.
- Add an authentication policy.
- Add an AI agent.
- Configure PingGateway.
- Start the MCP agent.

Before you begin

To complete this use case, you'll need:

- Administrator access in a PingOne environment that's configured to work with PingGateway. Learn more in PingGateway and PingOne in the PingGateway documentation.
- PingGateway installed and configured with server-side SSL (TLS). Learn more in MCP security gateway and Configuring PingGateway for TLS (server-side) in the PingGateway documentation. The sample application isn't required for this use case.

  Example admin.json file

  This is an example `.openig/config/admin.json` connector for SSL. The user is `macuser` and the secrets folder `/Users/macuser/.openig/secrets` contains the self-signed certificate for ig.example.com.

```
{
  "adminConnector": {
    "host": "localhost",
    "port": 8085
  },
  "connectors": [
    {
      "port": 8080
    },
    {
      "port": 8443,
      "tls": "ServerTlsOptions-1"
    }
  ],
  "streamingEnabled": true,
  "heap": [
    {
      "name": "ServerTlsOptions-1",
      "type": "ServerTlsOptions",
      "config": {
        "keyManager": {
          "type": "SecretsKeyManager",
          "config": {
            "signingSecretId": "key.manager.secret.id",
            "secretsProvider": "ServerIdentityStore"
          }
        }
      }
    },
    {
      "name": "ServerIdentityStore",
      "type": "FileSystemSecretStore",
      "config": {
        "format": "PLAIN",
        "directory": "/Users/macuser/.openig/secrets",
        "suffix": ".pem",
        "mappings": [{
          "secretId": "key.manager.secret.id",
          "format": {
            "type": "PemPropertyFormat"
          }
        }]
      }
    }
  ]
}
```

- A deployed MCP server capable of exposing tools the agent will access, such as `banking:read_balance` and `banking:transfer`. You can use the sample MCP and agent software described in MCP security gateway preparation in the PingGateway documentation.

> **Note**
>
> The sample MCP and agent software is for a weather application, but you can adapt them for this use case if you don't already have an MCP server or agent.

Tasks

Adding custom resources

Add custom resources that represent the services the agent is allowed to access. You can find more details in Adding a custom resource in the PingOne documentation.

You'll add two resources that serve different purposes in the delegation flow:

- The **test** resource represents the backend service that the agent calls through PingGateway. Its attributes map the human user into `sub` and, when delegation is valid, copy the user's `may_act` into `act` so the backend can see that this agent is acting on behalf of this user for a specific audience.
- The **agent** resource represents the agent itself and the delegation relationship between the user and the agent, populated in `may_act`. This relationship establishes who is allowed to act (which client ID the user delegated to) when the agent calls the backend. The backend resource uses `may_act` information to enforce what the agent is allowed to do with protected resources.

> **Note**
>
> Settings for your own custom resources might vary from those used as examples in this use case.

1. In the PingOne admin console, go to **Applications** > **Resources** and click the + icon.
2. Add the test resource:
    1. Name the resource `test`, enter `https://ig.example.com:8443/mcp` for the **Audience**, and click **Next**.
    2. Click the **Gear** icon (⚙) next to the `sub` attribute, enter the expression `#root.context.requestData.subjectToken.sub`, and click **Save**.

       Attribute mappings can pass complex information to applications through an access token. This expression copies the user's `sub` (subject) claim from the incoming subject token to let downstream services know which human the agent is acting on behalf of. Learn more about expressions in Using the expression builder in the PingOne documentation.
    3. Click **+ Add** and name the attribute `act`.
    4. Click ⚙ next to the `act` attribute, enter the following expression, and click **Save**.

       ```
       (#root.context.requestData.subjectToken.may_act.sub ==
       #root.context.requestData.actorToken.client_id)?
       #root.context.requestData.subjectToken.may_act:null
       ```

       This expression copies the `may_act` delegation from the user's token into `act` when the delegated client ID matches the current agent's `client_id`. When there isn't a match, `act` is left empty to prevent unauthorized delegation.
    5. Click **Next**, then click **+ Add Scope**.
    6. Name the scope `test` and click **Save**.
    7. Click the **test** resource to display the **Overview** tab.

8. Copy the **Resource ID** and **Client Secret** and store them somewhere convenient. You'll use them for PingGateway configuration.
9. Close the **test** resource.

3. Add the agent resource:
    1. Click the + icon.
    2. Name the resource `agent`, keep the default `agent` for the **Audience**, and click **Next**.
    3. For the `sub` attribute, select **Username** in the **PingOne Mappings** list.
    4. Click **+ Add** and name the attribute `may_act`.
    5. Click ⚙ next to the `may_act` attribute, enter the following expression, and click **Save**.

        ```
        (#root.context.requestData.grantType == "client_credentials")?null:({ "sub": #root.context.appConfig.clientId })
        ```

        This expression omits `may_act` for pure `client_credentials` requests, and otherwise builds a simple delegation object that records this agent's `client_id` as the subject of `may_act`.
    6. Click **Next**, then click **+ Add Scope**.
    7. Name the scope `agent` and click **Save**.

Adding a consent agreement

Add an agreement for the user's consent for agent actions. You can find more details in Agreements and Adding an agreement in the PingOne documentation.

1. In the PingOne admin console, go to **User Experience** > **Agreements** and click the + icon.
2. Name the agreement `Agent Consent` and enter a description such as `Consent for a digital assistant agent`.
3. For **Reconsent Every**, select **Number of Days** and keep the default `180`.
4. Click **Save**.
5. Add a language:
    1. Click **Edit Localized Content** and click the **Languages +** icon.
    2. For **Language**, select **English (en)**, and click **Save**.
    3. In **Localized Agreement Content**, enter `I consent to allow digital assistants created by MyCompany to act on my behalf`, then click **Save**.

        The language is enabled by default.
    4. Close the language.
6. On the **Agreements page**, click the toggle to enable the **Agent Consent** agreement.

Adding an authentication policy

Add an authentication policy for the agent. The authentication policy ensures that users explicitly agree to let a digital assistant act on their behalf before any delegated access is granted. By prompting for consent during sign-on, PingOne can issue user tokens that record this approval, which token exchange uses later to build the `act` and `may_act` claims. This keeps delegation transparent, auditable, and aligned with least-privilege and human-in-the-loop (HITL) requirements. Learn more in Adding an authentication policy in the PingOne documentation.

1. In the PingOne admin console, go to **Authentication > Authentication** and click **+ Add Policy**.
2. Name the policy `Agent-Consent-Login`.
3. For **Step Type**, select **Login**, then click **+ Add step**.
4. For **Step Type**, select **Agreement Prompt**.
5. Select the **Agent Consent** terms of service agreement and click **Save**.

Adding an AI agent

Register the agent as an identity in PingOne and configure its authentication settings, such as scopes for the actions the agent can perform. You can find more details in Managing AI agents in the PingOne documentation.

In our example scenario, the agent will interact with an application hosted locally, on the 3000 port.

1. In the PingOne admin console, go to **AI Agents** and click the + icon.
2. Name the agent `MCP Tutorial`, enter a description such as `MCP agent for PingGateway tutorial`, and click **Save**.
3. On the **Configuration** tab, click the **Pencil** icon (✎) to edit configuration settings.

   > **Note**
   >
   > Keep all of the default settings.

   1. For **Grant Type**, select **Client Credentials**, **Refresh Token**, and **Token Exchange**.

      These settings allow the agent to authenticate autonomously and exchange user tokens for delegation tokens at runtime.

   2. In **Redirect URIs**, enter `http://localhost:3000/callback`.
   3. Click **Save**.
4. On the **Resources** tab, click the **Pencil** icon (✎) to edit resource settings:

   1. Select the **agent** and **test** scopes.

      These scopes ensure that the agent can get its own access token and exchange user tokens for MCP access.

   2. Click **Save**.
5. On the **Policies** tab, click **+ Add Policies**.

   1. Select the **Agent-Consent-Login** policy.
   2. Click **Save**.

6. Click the toggle at the top of the panel to enable the agent.
7. On the **Overview** tab, copy the **Client ID** and store it somewhere convenient. You'll use it when restarting the MCP agent.

Result

You've successfully configured PingOne to act as the authorization server.

Configuring PingGateway

PingGateway serves as the enforcement point that protects the MCP server and validates tokens.

1. In the `admin.json` file for PingGateway, enable streaming:

```
"streamingEnabled": true
```

> **Note**
>
> This is already enabled in the example `admin.json` file you set up as a prerequisite. PingGateway requires this setting for server-side events (SSE), an MCP transport option. Learn more about server-side events (SSE) in the Mozilla documentation.

2. Export a RESOURCE_SECRET_ID environment variable with the base64 encoded client secret for the PingOne test resource:

```
bash: export RESOURCE_SECRET_ID=$(echo -n "<Resource client secret>" | base64)
```

> **Important**
>
> Make sure you don't include a new line.

3. Add the following route to PingGateway:

Linux

`$HOME/.openig/config/routes/mcp.json`

Windows

`%appdata%\OpenIG\config\routes\mcp.json`

Route configuration

Enter the `PingOne environment ID` and `PingOne test resource ID` for your deployment.

```
{
  "name": "mcp",
  "condition": "${find(request.uri.path, '^/mcp')}",
  "properties": {
    "pingOneEnvID": "https://auth.pingone.com/<PingOne environment ID>",
    "pingOneResourceID": "<PingOne test resource ID>",
    "gatewayUrl": "https://ig.example.com:8443",
    "mcpServerUrl": "http://localhost:8000"
  },
  "baseURI": "&{mcpServerUrl}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AuditService",
      "type": "AuditService",
      "config": {
        "eventHandlers": [
          {
            "class": "org.forgerock.audit.handlers.json.JsonAuditEventHandler",
            "config": {
              "name": "json",
              "logDirectory": "&{ig.instance.dir}/audit",
              "topics": ["access", "mcp"]
            }
          }
        ]
      }
    },
    {
      "name": "rsFilter",
      "type": "OAuth2ResourceServerFilter",
      "config": {
        "requireHttps": false,
        "scopes": ["test"],
        "accessTokenResolver": {
          "type": "TokenIntrospectionAccessTokenResolver",
          "config": {
            "endpoint": "&{pingOneEnvID}/as/introspect",
            "providerHandler": {
              "type": "Chain",
              "config": {
                "filters": [
                      {
                        "type": "HttpBasicAuthenticationClientFilter",
                        "config": {
                          "username": "&{pingOneResourceID}",
                          "passwordSecretId": "resource.secret.id",
                          "secretsProvider": "SystemAndEnvSecretStore-1"
                        }
                      }
                    ],
                "handler": "ForgeRockClientHandler"
              }
            }
          }
        }
      }
```

```
      }
    ],
    "handler": {
      "type": "Chain",
      "config": {
        "filters": [
          {
            "type": "McpAuditFilter",
            "config": {
              "auditService": "AuditService"
            }
          },
          {
            "type": "UriPathRewriteFilter",
            "config": {
              "mappings": { "/mcp": "/" }
            }
          },
          {
            "type": "McpProtectionFilter",
            "config": {
              "resourceId": "&{gatewayUrl}/mcp",
              "authorizationServerUri": "&{pingOneEnvID}/as",
              "resourceServerFilter": "rsFilter",
              "supportedScopes": ["test"],
              "resourceIdPointer": "/aud/0"
            }
          },
          {
            "type": "McpValidationFilter",
            "config": {
              "acceptedOrigins": ".*"
            }
          }
        ],
        "handler": {
          "type": "ReverseProxyHandler",
          "config": {
            "soTimeout": "20 seconds"
          }
        }
      }
    }
  }
```

Features of the sample route:

- This route uses a secret obtained from an environment variable.
- PingGateway acts as an OAuth 2.0 resource server (RS) when protecting the sample MCP server.
- The McpAuditFilter audits MCP requests. PingGateway records MCP audit events in an `audit/mcp.audit.json` file.
- The UriPathRewriteFilter sends the request to the root resource of the MCP server. The MCP server expects requests at `/`.
- The McpProtectionFilter uses the RS configuration, extending it for MCP.

- PingGateway validates MCP requests with an McpValidationFilter.
- The ReverseProxyHandler uses a long `"soTimeout"` setting to accommodate an MCP agent receiving few or infrequent SSE updates.

4. Restart PingGateway to apply the route changes.
5. (Optional) Add throttling or fine-grained access control as necessary to meet your security requirements.

> **Note**
>
> This simple route doesn't include those features.

6. Check the PingGateway log to verify the route loads successfully.

Result

You've successfully configured PingGateway to protect the sample MCP server.

Start the MCP agent

1. In the directory where you unpacked the sample MCP agent, export the AI agent secret as an environment variable called AGENT_SECRET.
2. Start the sample MCP agent again with the added option `–client id`.

   You'll need the client ID from the MCP Tutorial agent. Point it to the PingGateway route for MCP requests.

```
bash-3.2$ export AGENT_SECRET=yjS0IWMX9LS91...
bash-3.2$ python3 sample-mcp-agent.py --client-id <PingOne AI agent client ID> --mcp-server-url https://
ig.example.com:8443/mcp
```

Result

You've successfully started the sample MCP agent.

Validation

To validate this use case, you'll confirm the full flow:

- The agent can discover tools from the MCP server.
- The agent can open a browser and redirect to PingOne.
- The user can sign on and grant consent.
- The agent can make a tool call and receive a response.
- PingGateway logs show the MCP request with a valid act claim.

With PingGateway protecting the MCP server, the sample MCP agent directs you to the authorization server to sign on as an end user and authorize access to make MCP requests.

1. Allow the agent to open a browser automatically, or:

    1. Select `'n'`.

    2. Copy the authorization URL that the sample MCP agent displays in your terminal.

    3. Paste the URL into a browser.

       The URL should look similar this:

       ```
       https://auth.pingone.com/082e56dd-1a0f-4e1d-a28e-77d51ecf1705/as/authorize?
       response_type=code&client_id=3fffb1bf-106c-4449-a9bd-
       df6fa76d8f41&redirect_uri=http%3A%2F%2Flocalhost%3A3000%2Fcallback&state=…
       ```

2. Sign on as the test user and consent to allow a digital assistant (if asked) before you close the browser tab.

   The following commands are available in the terminal where the sample MCP agent is running:

   ```
   [INFO] Discovered tools [https://ig.example.com:8443/mcp]:
   [INFO] - geocode: Returns a list of objects containing city name, latitude, longitude, country,
   admin1 (region), and timezone for each matching city
   [INFO] - forecast_daily: Returns a multi-day weather forecast for a given location
   [INFO] - forecast_periods: Returns weather forecasts for each representative period of the current
   day
   [INFO] - forecast_hourly: Returns an hourly weather forecast for the current day
   [INFO] - weather_at_time: Returns the forecasted weather for a specific time at a given location

   Enter your message (or 'exit|quit|q'):
   ```

3. Enter a prompt and get a response from the MCP server through PingGateway, then exit the agent.

   The following example uses the forecast_daily tool to get the daily forecast for Tokyo:

   ```
   Enter your message (or 'exit|quit|q'): What is the daily forecast for Tokyo?
   Agent: The daily forecast for Tokyo is:

   <MCP server response with forecast details>

   Enter your message (or 'exit|quit|q'): exit
   User requested exit. Goodbye!
   ```

   The following tokens are used in the transaction:

   - Actor token: The AI agent's access token.

     Agent token (actor token) example

```
{
  "aud": [
    "agent"
  ],
  "client_id": "adeeb901-7b64-453d-92bc-059bcbcc5958",
  "env": "4631af52-7ec7-48cf-848f-48cf8ca8e1ab",
  "exp": 1774043329,
  "iat": 1774039729,
  "iss": "https://auth.pingone.com/4631af52-7ec7-48cf-848f-48cf8ca8e1ab/as",
  "jti": "722238d2-4748-4681-889d-7fe7a321b037",
  "org": "84c4c0c0-0ca2-43b4-a62c-d830882855bc",
  "p1.rid": "722238d2-4748-4681-889d-7fe7a321b037",
  "scope": "agent"
}
```


- Subject token: The end user's access token. The `may_act` claim asserts that the agent is allowed to act on the user's behalf.

User token (subject token) example


```
{
  "acr": "Agent-Consent-Login",
  "aud": [
    "agent"
  ],
  "auth_time": 1774039361,
  "client_id": "adeeb901-7b64-453d-92bc-059bcbcc5958",
  "env": "4631af52-7ec7-48cf-848f-48cf8ca8e1ab",
  "exp": 1774043333,
  "iat": 1774039733,
  "iss": "https://auth.pingone.com/4631af52-7ec7-48cf-848f-48cf8ca8e1ab/as",
  "jti": "3121e22a-5958-4bf1-a369-ba296b444930",
  "may_act": {
    "sub": "adeeb901-7b64-453d-92bc-059bcbcc5958"
  },
  "org": "84c4c0c0-0ca2-43b4-a62c-d830882855bc",
  "p1.userId": "cf025143-501a-4d70-9c80-95d69121d7b3",
  "scope": "agent",
  "sid": "2dd6d4a4-0b59-4c99-8a57-4c189f09b24f",
  "sub": "demouser"
}

[INFO] Exchanging actor token and subject token for a new mcp token (scope='test')
```


- MCP token: An on-behalf-of token obtained through PingOne token exchange. The `act` and `sub` claims assert that this token is used by the agent on behalf of the test user.

Exchanged on-behalf-of token (mcp token) example

```
{
  "acr": "Agent-Consent-Login",
  "act": {
    "sub": "adeeb901-7b64-453d-92bc-059bcbcc5958"
  },
  "aud": [
    "https://ig.example.com:8443/mcp"
  ],
  "auth_time": 1774039361,
  "client_id": "adeeb901-7b64-453d-92bc-059bcbcc5958",
  "env": "4631af52-7ec7-48cf-848f-48cf8ca8e1ab",
  "exp": 1774043333,
  "iat": 1774039733,
  "iss": "https://auth.pingone.com/4631af52-7ec7-48cf-848f-48cf8ca8e1ab/as",
  "jti": "e1e90599-b09f-4549-8ea9-6a297490e53c",
  "org": "84c4c0c0-0ca2-43b4-a62c-d830882855bc",
  "p1.userId": "cf025143-501a-4d70-9c80-95d69121d7b3",
  "scope": "test",
  "sid": "2dd6d4a4-0b59-4c99-8a57-4c189f09b24f",
  "sub": "demouser"
}
```


Tip

You can decode tokens using your preferred JSON Web Token (JWT) viewer. Don't paste sensitive tokens from production environments into third-party sites.

If your delegation token is missing the `act` claim, verify:

- PingOne resource attribute mappings for act are configured correctly.
- The subject token contains a `may_act` claim.
- The agent's `client_id` matches `may_act.sub`.

4. Check the PingGateway log for additional details about the MCP request.

Result

You've successfully validated that PingGateway can protect the MCP server.

Troubleshooting

If you encounter issues while configuring this use case, use the following information to help resolve common problems.

Token exchange fails and `act` claim is null

Verify that the attribute expressions in your custom resources accurately match client IDs and that the `act` claim is properly mapped in the token.

PingGateway drops MCP traffic

Verify that `streamingEnabled` is set to true in your `admin.json` file. PingGateway requires this setting for SSE utilized by the MCP protocol.

Gateway routing errors

Verify that the UriPathRewriteFilter is properly stripping the `/mcp` path down to `/` before it reaches the backend MCP server, as most MCP servers expect requests at the root.

Consent prompt issues

If the user doesn't receive an agent consent prompt, verify these things in PingOne:

- The Agent-Consent-Login policy is assigned to the agent on the **Policies** tab in **AI Agents**.
- The **Redirect URI** is correct on the **Configuration** tab in **AI Agents**.

If the user receives a consent prompt on every sign-on, verify these things in PingOne:

- The **Reconsent Every** interval isn't too short. You can edit the interval on the **Overview** tab in **User Experience > Agreements**.
- Scopes aren't changing between sessions.

Agent can list tools but not call them

Verify the following:

- The MCP scope (**test** in the example) isn't missing or incorrect.
- The PingGateway route includes MCP filters and the **Resource ID** is configured correctly.

What's next

Add fine-grained authorization policies to secure agent actions using real-time contextual signals. Learn more about authorization in the PingOne Authorize documentation.

Securing AI Agents with PingFederate using delegated access tokens

PingFederate

PingFederate supports OAuth 2.0 Token Exchange (RFC 8693), allowing you to issue standard access tokens containing delegation claims that provide a full audit trail of agent activity. Each token can be scoped to the strict minimum required for the specific task. This enforces the concept of least-privilege across the entire chain, ensuring that even if one agent is compromised, the damage is restricted to its specific, short-lived scope.

Consider an enterprise expense management system where employees interact with an AI-powered assistant through a web portal. The assistant needs to:

- Read the employee's expense reports
- Look up budget data for the employee's cost center
- Approve or submit expenses on the employee's behalf

Without using a delegation model such as OAuth and OpenID Connect (OIDC), organizations face two bad choices:

1. Share user credentials with the agent, with no audit trail distinguishing agent actions from human actions. If the agent is compromised, the attacker has full user access.
2. Give the agent its own service account, where there is no link between the human who authorized the action and the agent performing it.

PingFederate configured for token exchange leverages existing OAuth and OIDC flows to provide constrained delegation with full traceability.

Goals

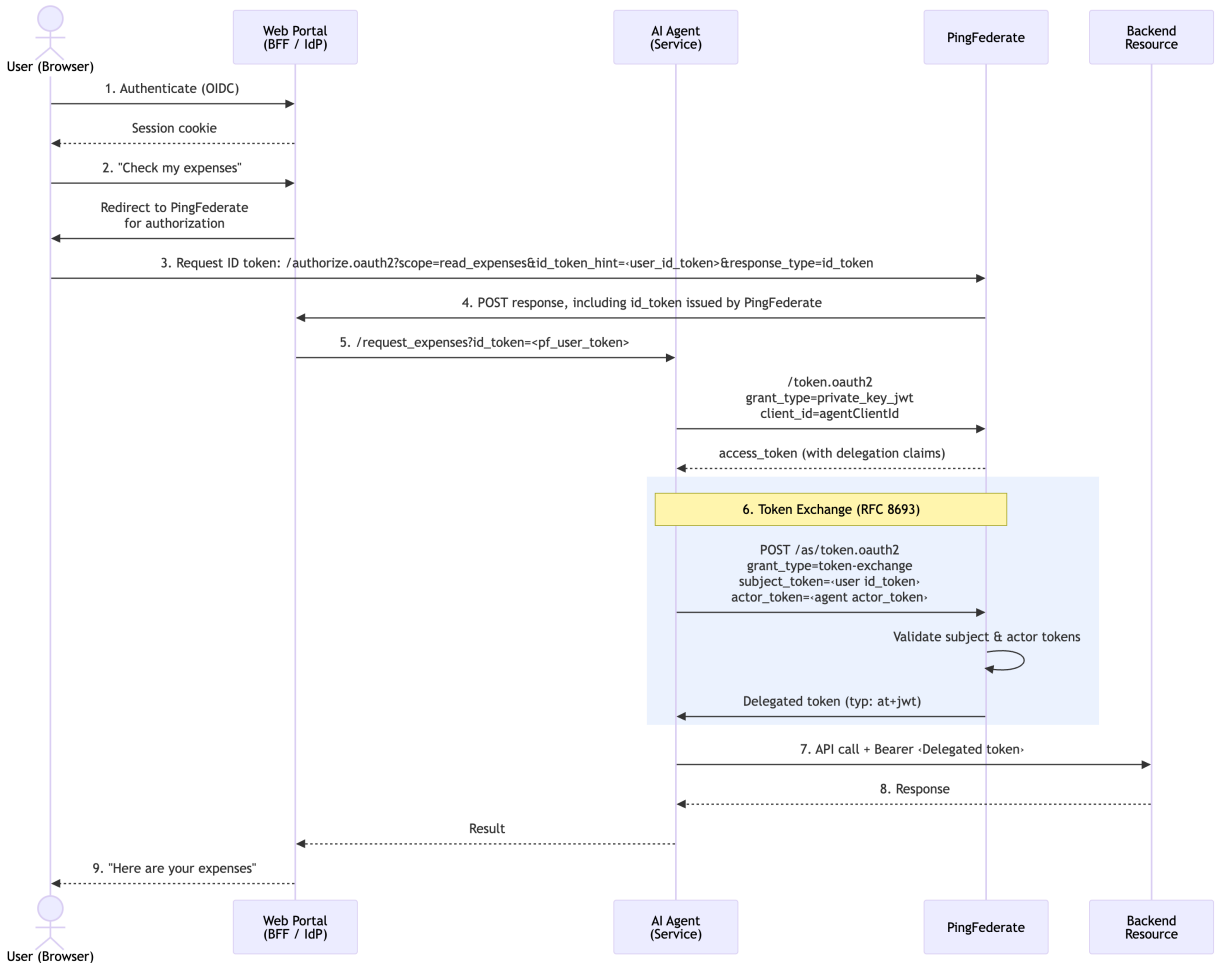
- Configure PingFederate to process token exchange requests and issue delegated access tokens
- Optionally configure CIBA for step-up authentication for sensitive transactions where human approval is required
- Optionally configure SPIFFE/SPIRE to achieve even stronger cryptographic proof of agent identity compared to a standard JWT.

What you'll do

1. Defining the access token manager
2. Defining scopes for downstream resources
3. Configuring token processors for subject and actor validation
4. Creating a token exchange processor policy
5. Mapping the token exchange policy to the ATM
6. Registering the agent as an OAuth client
7. Optionally configure step-up authorization with CIBA
8. Optionally configure cryptographic agent identity with SPIFFE/SPIRE

Token exchange architecture

PingFederate converts a user's identity assertion into a delegated access token that encodes exactly who authorized the action, which agent is performing it, and which operations are permitted.



A delegation access token issued by PingFederate carries a precise record of the delegation:

```
{
  "typ": "at+jwt",
  "iss": "https://pingfederate.example.com",
  "aud": "https://api.example.com",
  "sub": "alice@example.com",
  "act": {
    "sub": "expense-agent"
  },
  "scope": "expenses:read tools:list",
  "department": "Finance",
  "iat": 1742200000,
  "exp": 1742200300
}
```

The token contains the following claims:

Claim	Description
`sub`	The human user who authorized the action

Claim	Description
`act.sub`	The AI agent performing the action
`scope`	Constrained permissions granted to the agent for this action
`department`	Custom attribute passed downstream for attribute-based access decisions
`aud`	Restricts the token to a specific downstream resource
`typ`	Standard JWT access token type (`at+jwt` per RFC 9068)

This gives downstream services everything they need to make authorization decisions: who is the human, who is the agent, what they're allowed to do, and how to trace this specific action.

Delegated access tokens offer many security benefits:

Short-lived

Delegated access tokens have a tight time-to-live (TTL), such as 5 minutes. They are not stored, cached, or refreshed, and each new action requires a new exchange.

Audience-restricted

The token is only valid at a specific resource server, preventing it from accepting an unauthorized token.

Agent never holds user credentials

The agent never sees the user's session or password. It only receives a constrained delegation token.

Tasks

Defining the access token manager

Create a dedicated Access Token Manager (ATM) for delegated access tokens. This ATM issues standard JWTs (`at+jwt`) with an extended attribute contract that carries delegation claims.

Admin console

Steps

1. In the PingFederate admin console, go to **Applications > OAuth > Access Token Management** and click **Create New Instance**.
2. On the **Type** tab, configure the following values:

Field	Value	Description
Instance Name	Transaction Token Manager	Descriptive name for the ATM
Instance ID	`TxnTokenMgr`	ID referenced by token exchange mappings
Type	**JSON Web Tokens**	Standard JWT ATM format

3. On the **Instance Configuration** tab, click **Show Advanced Fields** and configure the following values:

Setting	Value	Description
Token Lifetime	`300`	Time in seconds
JWS Algorithm	**RSA using SHA-256**	Use centralized signing key
Issuer Claim	`https://pingfederate.example.com`	Your PingFederate issuer
Audience Claim	`https://api.example.com`	Target resource server
Type Header Value	`at+jwt`	Standard access token type per RFC 9068
Expand Scope Groups	**True**	Expands scope group names to individual scopes

4. On the **Access Token Attribute Contract** tab, add the following extended attributes:

Attribute	Description
`sub`	Human user identity mapped from the subject token
`act`	Agent identity as a JSON object `{"sub": "<agent-id>"}`
`scope`	Delegated scopes (space-delimited)

5. On the **Resource URIs** tab, set the **Resource URI** to the target audience, for example `https://api.example.com`.

   PingFederate will automatically select this ATM when the token exchange targets this resource.

6. Click **Save**.

Developer API

Steps

1. Use a `POST` request to the `/pf-admin-api/v1/oauth/accessTokenManagers` endpoint to configure the PingFederate ATM.

```
POST /pf-admin-api/v1/oauth/accessTokenManagers
```

```
{
  "id": "TxnTokenMgr",
  "name": "Transaction Token Manager",
  "pluginDescriptorRef": {
    "id": "com.pingidentity.pf.access.token.management.plugins.JwtBearerAccessTokenManagementPlugin"
  },
  "configuration": {
    "fields": [
      { "name": "Token Lifetime", "value": "300" },
      { "name": "Use Centralized Signing Key", "value": "true" },
      { "name": "JWS Algorithm", "value": "RS256" },
      { "name": "Issuer Claim Value", "value": "https://pingfederate.example.com" },
      { "name": "Audience Claim Value", "value": "https://api.example.com" },
      { "name": "JWT ID Claim Length", "value": "22" },
      { "name": "Include Key ID Header Parameter", "value": "true" },
      { "name": "Include Issued At Claim", "value": "true" },
      { "name": "Scope Claim Name", "value": "scope" },
      { "name": "Space Delimit Scope Values", "value": "true" },
      { "name": "Expand Scope Groups", "value": "true" },
      { "name": "Type Header Value", "value": "at+jwt" }
    ]
  },
  "selectionSettings": {
    "resourceUris": [ "https://api.example.com" ]
  },
  "attributeContract": {
    "extendedAttributes": [
      { "name": "sub" },
      { "name": "act" },
      { "name": "scope" }
    ]
  }
}
```

Defining scopes for downstream resources

Define fine-grained scopes that represent the operations your AI agent can perform. Group baseline scopes that every token exchange receives, and keep elevated scopes separate for step-up authorization.

1. In the PingFederate admin console, go to **System > OAuth Settings > Scope Management**.
2. Create the following scopes:

Scope	Description	Included in Base Group
`expenses:read`	Read expense reports	Yes
`tools:list`	List available tools	Yes
`expenses:approve`	Approve expense reports	No (requires CIBA)
`expenses:submit`	Submit expense reports	No (requires CIBA)
`budget:read`	View budget information	No (requires CIBA)

3. Create a **Scope Group Value** containing the base scopes, for example `base-agent-scopes`, and add the relevant scopes.

   When the ATM has **Expand Scope Groups** enabled, the group name is expanded to the individual scope strings in the issued token.

4. Click **Save**.

Configuring token processors for subject and actor validation

Create two token processors:

Subject token processor

Validates the JWT that asserts the user's identity. It can come from an upstream identity provider, a reverse proxy, or another PingFederate instance.

Actor token processor

Validates the JWT that identifies the AI agent making the request. This can be any verifiable JWT, such as a client-credentials token, a service mesh identity, or a SPIFFE JWT-SVID.

> **Note**
>
> If your infrastructure uses SPIFFE/SPIRE for workload identity, the agent can present a JWT-SVID as the actor token. This provides cryptographic proof of the agent's identity tied to its runtime environment. Unlike a client ID, a JWT-SVID is an attestation that this specific workload running in this specific namespace is the one making the request. PingFederate validates the JWT-SVID signature using the SPIRE OIDC Discovery Provider's JWKS endpoint.

Admin console

Steps

1. In the PingFederate admin console, go to **Authentication > Token Exchange > Token Processors**. Click **Create New Instance**.
2. In the **Type** field, click **JWT Token Processor 2.0**. Click **Next**.
3. On the **Instance Configuration** tab, click **Show Advanced Fields** and configure the following values for the subject token processor:

Setting	Value	Description
Issuer	`https://pingfederate.example.com:9031`	The upstream token issuer
JWKS URL	`https://pingfederate.example.com:9031/pf/JWKS`	The issuer's JWKS endpoint for signature validation
Require Audience	**True**	Whether the `aud` claim is required in the token
Require Expiration Time	**True**	Whether the `exp` claim is required in the token
Allowed Clock Skew	`10`	Time in seconds

4. In the **Attribute Contract** tab, map the `sub` claim, and any other optional claims such as `email`, `name`, and `department`. Click **Save**.
5. Repeat steps 1 and 2 to create the actor token processor.
6. On the **Instance Configuration** tab, click **Show Advanced Fields** and configure the following values for the actor token processor:

Setting	Value	Description
Issuer	`https://spire.example.com`	Agent identity provider (for example, SPIRE OIDC endpoint, or your own issuer)
JWKS URL	`https://spire.example.com/jwks`	The agent identity provider's JWKS endpoint
Require Audience	**True**	Whether the `aud` claim is required in the token
Require Expiration Time	**True**	Whether the `exp` claim is required in the token

7. In the **Attribute Contract** tab, map the `sub` claim from the actor token and click **Save**.

Developer API

Steps

1. Use `POST` requests to the `/pf-admin-api/v1/idp/tokenProcessors` endpoint to configure the token processors.

```
POST /pf-admin-api/v1/idp/tokenProcessors
```

```
{
  "id": "SubjectTokenProcessor",
  "name": "Subject Token Processor",
  "pluginDescriptorRef": {
    "id": "com.pingidentity.pf.tokenprocessors.jwt.JwtTokenProcessor"
  },
  "configuration": {
    "fields": [
      { "name": "Require Audience", "value": "true" },
      { "name": "Require Expiration Time", "value": "true" },
      { "name": "Allowed Clock Skew", "value": "10" }
    ],
    "tables": [
      {
        "name": "Allowed Issuers",
        "rows": [{
          "fields": [
            { "name": "Issuer", "value": "https://upstream-idp.example.com" },
            { "name": "JWKS URL", "value": "https://upstream-idp.example.com/jwks" }
          ]
        }]
      },
      {
        "name": "Allowed Audiences",
        "rows": [{
          "fields": [
            { "name": "Audience", "value": "expense-agent" }
          ]
        }]
      }
    ]
  },
  "attributeContract": {
    "coreAttributes": [{ "name": "sub" }],
    "extendedAttributes": [
      { "name": "email" },
      { "name": "name" }
    ]
  }
}
```

Creating a token exchange processor policy

The token exchange processor policy defines which subject and actor token processors PingFederate will use when presented with a token exchange request with the specified token types.

Admin console

Steps

1. In the PingFederate admin console, go to **Applications > Token Exchange > Processor Policies** and click **Add Processor Policy**.
2. Enter a **Name** and **ID** for the policy, and select **Actor Token Required**. Click **Next**.
3. In the **Token Processor Mapping** tab, click **Map New Token Processor**. Configure the following values and then click **Next**:

Setting	Value	Description
Subject Token Processor	**Your Subject Token Processor**	The UI name of the subject token processor you created in Configuring token processors for subject and actor validation
Subject Token Type	`urn:ietf:params:oauth:token-type:jwt`	Standard JWT token type
Actor Token Processor	**Your Actor Token Processor**	The UI name of the actor token processor you created in Configuring token processors for subject and actor validation
Actor Token Type	`urn:ietf:params:oauth:token-type:jwt`	Standard JWT token type

4. In the **Contract Fulfillment** tab, map the following values:

Processor Policy Contract	Source	Value
`subject`	**Subject Token**	**sub**
`actor_sub`	**Actor Token**	**sub**

5. Click **Save**.

Developer API

Steps

1. Use a `POST` request to the `/pf-admin-api/v1/oauth/tokenExchange/policies` endpoint to configure the token exchange policy.

```
POST /pf-admin-api/v1/oauth/tokenExchange/policies
```

```
{
  "id": "TokenExchangePolicy",
  "name": "User Token Exchange Policy",
  "actorTokenRequired": true,
  "attributeContract": {
    "coreAttributes": [{ "name": "subject" }],
    "extendedAttributes": [{ "name": "actor_sub" }]
  },
  "processorMappings": [{
    "subjectTokenType": "urn:ietf:params:oauth:token-type:jwt",
    "subjectTokenProcessor": { "id": "SubjectTokenProcessor" },
    "actorTokenType": "urn:ietf:params:oauth:token-type:jwt",
    "actorTokenProcessor": { "id": "ActorTokenProcessor" },
    "attributeContractFulfillment": {
      "subject": {
        "source": { "type": "SUBJECT_TOKEN" },
        "value": "sub"
      },
      "actor_sub": {
        "source": { "type": "ACTOR_TOKEN" },
        "value": "sub"
      }
    }
  }]
}
```

Mapping the token exchange policy to the ATM

An access token mapping binds the token exchange policy to the ATM. This mapping controls how claims from the validated subject and actor tokens are transformed into claims in the issued access token.

Admin console

Steps

1. In the PingFederate admin console, go to **Applications > OAuth > Access Token Mappings**.
2. Configure the following values and then click **Add Mapping**:

Setting	Value	Description
Context	**Token Exchange Processor Policy**	The name of the token exchange policy you created in Creating a token exchange processor policy
Access Token Manager	**Transaction Token Manager**	The name of the ATM you created in Defining the access token manager

3. In the **Contract Fulfillment** tab, configure the following values:

Contract	Source	Value	Description
`sub`	**Token Exchange Processor Policy**	**subject**	Maps the `sub` claim in the issued token to the `subject` attribute from the token processor policy
`act`	**Expression**	`Collections.singletonMap("sub", actor_sub)`	Constructs the `act` claim as a JSON object containing the agent's identity
`scope`	**No Mapping**		PingFederate populates from granted scopes

> **Note**
>
> The `act` claim is constructed as a JSON object `{"sub": "<agent-identity>"}` per RFC 8693 Section 4.1. Using an OGNL expression in PingFederate allows this nested structure to be built at issuance time.

4. Click **Save**.

Developer API

Steps

1. Use a `POST` request to the `/pf-admin-api/v1/oauth/accessTokenMappings` endpoint to configure the Access Token Mapping.

```
POST /pf-admin-api/v1/oauth/accessTokenMappings
```

```
{
  "context": {
    "type": "TOKEN_EXCHANGE_PROCESSOR_POLICY",
    "contextRef": { "id": "TokenExchangePolicy" }
  },
  "accessTokenManagerRef": { "id": "TxnTokenMgr" },
  "attributeContractFulfillment": {
    "sub": {
      "source": { "type": "TOKEN_EXCHANGE_PROCESSOR_POLICY" },
      "value": "subject"
    },
    "act": {
      "source": { "type": "EXPRESSION" },
      "value": "@java.util.Collections@singletonMap(\"sub\", #this.get(\"tepp.actor_sub\") != null ?
#this.get(\"tepp.actor_sub\").toString() : \"\")"
    },
    "scope": {
      "source": { "type": "NO_MAPPING" }
    }
  }
}
```

Registering the agent as an OAuth client

The agent requires the **Token Exchange** grant type and optionally **CIBA** for step-up authorization flows.

Admin console

Steps

1. In the PingFederate admin console, go to **Applications > OAuth > Clients** and click **Add Client**.
2. Configure the following values:

Setting	Value	Description
Client ID	**exampleAgent**	A service identifier or SPIFFE ID
Client Authentication	**PRIVATE KEY JWT**	PingFederate supports multiple authentication methods
JWKS URL	**https://example.spire.com/jwks**	PingFederate fetches the agent's public keys from here
Allowed Grant Types	**Token Exchange**, **CIBA**	Token exchange is required. CIBA enables step up
Default Access Token Manager	**Transaction Token Manager**	Delegation access tokens issued by default
Exclusive Scopes	`base-agent-scopes`	Scopes reserved for this agent

Client Authentication Methods: PingFederate supports multiple methods for the agent to authenticate during token exchange:

Method	Description	Best For
Private Key JWT	Agent signs a JWT with its private key (X.509, SPIFFE SVID, and so on)	Zero-trust, certificate-based environments
Client Secret	Shared secret	Simpler deployments
Client TLS Certificate	Mutual TLS with client certificate	Infrastructure-level mTLS

Developer API

Steps

1. Use an `POST` request to the `/pf-admin-api/v1/oauth/clients` endpoint to configure the OAuth client.

```
POST /pf-admin-api/v1/oauth/clients
```

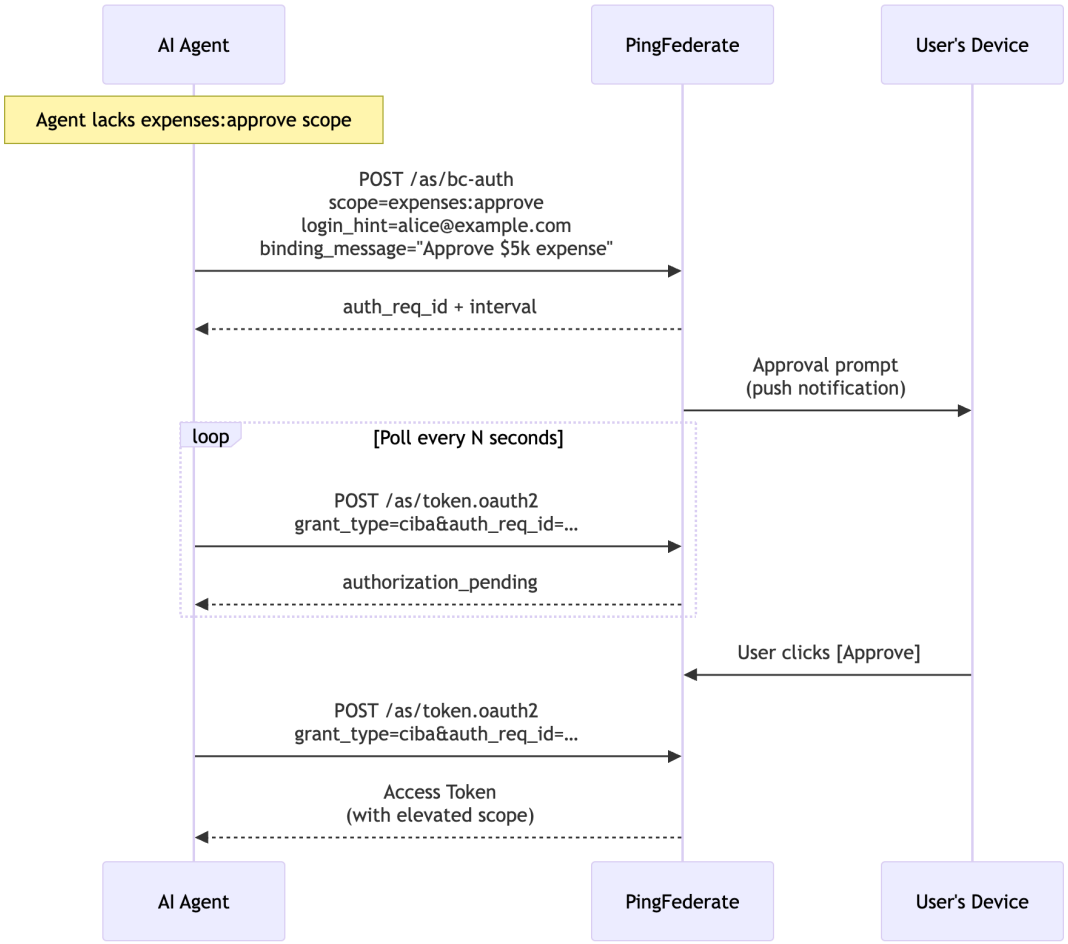
```
{
  "clientId": "expense-agent",
  "name": "Expense AI Agent",
  "enabled": true,
  "clientAuth": {
    "type": "PRIVATE_KEY_JWT"
  },
  "jwksSettings": {
    "jwksUrl": "https://agent.internal/jwks"
  },
  "grantTypes": [ "TOKEN_EXCHANGE", "CIBA" ],
  "defaultAccessTokenManagerRef": { "id": "TxnTokenMgr" },
  "exclusiveScopes": [ "base-agent-scopes" ],
  "bypassApprovalPage": true
}
```

Advanced: Step-up authorization with CIBA

Some operations require explicit user consent before the agent can proceed. PingFederate supports Client-Initiated Backchannel Authentication (CIBA) to enable a step-up authorization flow:

1. The agent attempts an operation that requires an elevated scope, such as `expenses:approve`.
2. The resource server returns a structured error indicating insufficient permissions.
3. The agent initiates a CIBA request to PingFederate, specifying the user (`login_hint`), desired scope, and a human-readable description (`binding_message`).
4. PingFederate delivers an approval prompt to the user's device through a webhook, push notification, or custom authenticator.
5. The user reviews and explicitly approves the action.
6. The agent polls the PingFederate token endpoint and receives a new access token with the elevated scope.
7. The agent retries the operation with the new token.

This ensures that sensitive actions (such as approving expenses or modifying budgets) require the human to explicitly authorize each action, even when delegating to an AI agent.



Security Considerations

Agents never hold user credentials

The AI agent never receives the user's session token, access token, or credentials. It only receives delegation access tokens, which are constrained, short-lived, audience-restricted assertions of delegated authority. If the agent is compromised, the attacker cannot escalate to the user's full privileges or access other services.

Audit trail

Every delegation access token carries the full delegation chain (`sub` + `act`). This enables:

- Correlating agent actions back to the authorizing user
- Distinguishing between human-initiated and agent-initiated operations
- Detecting anomalous agent behavior tied to specific users

Least privilege with scope groups

By defining scope groups with **Expand Scope Groups** enabled on the ATM, administrators control exactly which operations agents can perform by default. Elevated scopes require explicit CIBA step-up consent, enforcing the principle of least privilege at the token level.

Short Token Lifetime

A 5-minute time-to-live (TTL) on delegated access tokens means each agent session requires a fresh token exchange. There is no long-lived credential to steal, and token revocation is effectively automatic.

Advanced: Cryptographic Agent Identity with SPIFFE and SPIRE

In the previous steps, you configured PingFederate to validate an actor token that identifies the AI agent. In many deployments, this is a client-credentials token or a pre-provisioned JWT. These credentials must be distributed, rotated, and stored securely, and if an attacker extracts the agent's credentials, they can impersonate the agent from any location.

Secure Production Identity Framework for Everyone (SPIFFE) and its reference implementation SPIRE offer a stronger alternative: an automatically rotated workload identity that binds the agent's credential to its runtime environment.

SPIFFE and SPIRE provide the following benefits:

Platform attestation

The agent's identity is issued only to a workload running in a verified environment (such as a Kubernetes namespace, service account, or node) and not to anyone who possesses a static secret.

Automatic rotation

JWT-SVIDs and X.509-SVIDs are short-lived and automatically rotated by the SPIRE Agent.

No secret distribution

Agents receive identities through a local Unix domain socket (SPIFFE Workload API) so that no secrets stored are in config files, environment variables, or vault systems.

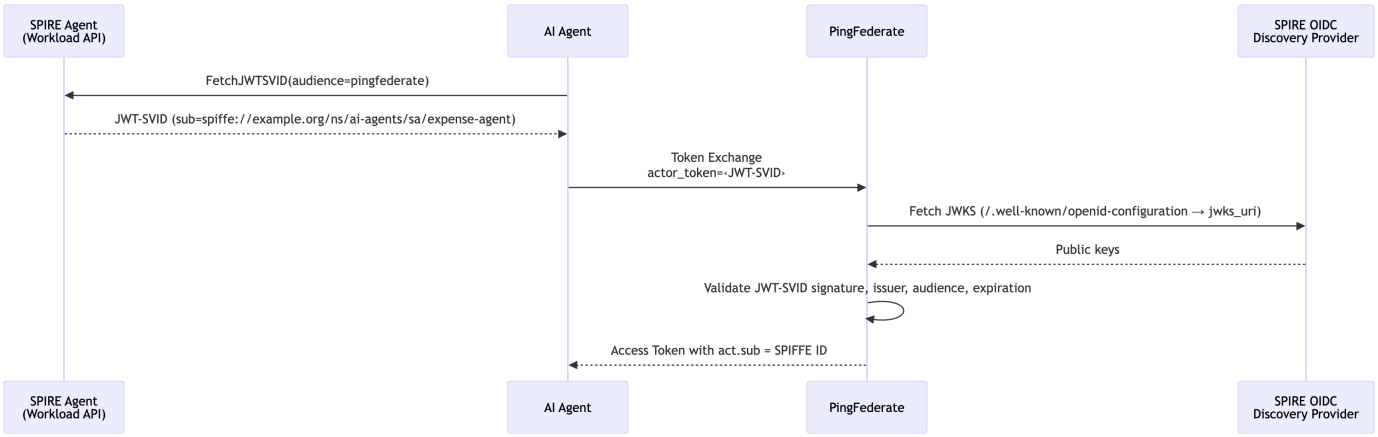
Uniform identity model

Every workload gets a SPIFFE ID URI (for example, `spiffe://example.org/ns/ai-agents/sa/expense-agent`) that works consistently across Kubernetes, VMs, and cloud environments.

> **Note**
>
> Learn more in the following resources:
>
> - SPIFFE Concepts: Trust domains, SPIFFE IDs, SVID types
> - SPIRE Architecture: Server, Agent, Workload API
> - SPIRE Quickstart for Kubernetes: Deploy SPIRE with Helm
> - JWT-SVID Specification: JWT format for SPIFFE identities

How it works with PingFederate

When an AI agent runs in a SPIRE-managed environment, it can mint a JSON Web Token SPIFFE Verifiable Identity Document (JWT-SVID) on demand with the SPIFFE Workload API. This JWT-SVID is used as the `actor_token` in the token exchange request, proving the agent's identity to PingFederate without any pre-shared secrets.



The SPIRE OIDC Discovery Provider exposes a standard `/.well-known/openid-configuration` endpoint with a `jwks_uri`, making JWT-SVIDs verifiable by any standard OIDC-compliant relying party, including PingFederate.

`private_key_jwt` with X.509-SVIDs

SPIFFE also provides X.509-SVIDs, which are short-lived client certificates that can be used for `private_key_jwt` client authentication when the agent calls the PingFederate token endpoint. In this model, the agent signs its `client_assertion` JWT with the private key from its X.509-SVID, and PingFederate validates the signature using the agent's JWKS endpoint (which serves the public key from the same SVID). This eliminates all static secrets from the token exchange flow, because both the client authentication and the actor token are backed by SPIRE-issued, automatically rotated credentials.

Configuring PingFederate

To validate JWT-SVIDs as actor tokens, configure the Actor Token Processor (JWT Token Processor 2.0) with the SPIRE OIDC Discovery Provider as the trusted issuer.

Admin console

Steps

1. In the PingFederate admin console, go to **Authentication > Token Exchange > Token Processors**.
2. Select the actor token processor you created in step 3 and click the **Instance Configuration** tab.
3. Click **Show Advanced Fields** and configure the following values:

Field	Value	Description
Issuer	`https://spire-oidc.example.org`	SPIRE OIDC Discovery Provider URL. Must match the `iss` claim in JWT-SVIDs.
JWKS URL	`https://spire-oidc.example.org/keys`	SPIRE OIDC Discovery Provider JWKS endpoint where PingFederate fetches signing keys.
Audience	`https://pingfederate.ping-identity.workers.dev`	Set to PingFederate's own issuer URL. When the agent mints the JWT-SVID, it requests this value as the `aud` claim, binding the SVID to PingFederate specifically.
Require Audience	`true`	JWT-SVID must contain the PingFederate issuer URL as the audience.
Require Expiration Time	`true`	Require the `exp` claim in the token. JWT-SVIDs are short-lived by design.
Max Future Validity	`1`	The time in minutes that specifies how far in the future the `exp` claim can be. SPIRE SVIDs have short lifetimes. Reject anything with an unreasonably long expiry.

4. Click the **Extended Contract** tab and map the `sub` claim.

> **Note**
>
> For JWT-SVIDs, the `sub` claim contains the full SPIFFE ID (for example, `spiffe://example.org/ns/ai-agents/sa/expense-agent`), which flows into the access token's `act.sub` claim.

Developer API

Steps

1. Use a `POST` request to the `/pf-admin-api/v1/idp/tokenProcessors` endpoint to configure PingFederate.

```
POST /pf-admin-api/v1/idp/tokenProcessors
```

```
{
  "id": "AgentActorProcessor",
  "name": "Agent Actor Token Processor",
  "pluginDescriptorRef": {
    "id": "com.pingidentity.pf.tokenprocessors.jwt.JwtTokenProcessor"
  },
  "configuration": {
    "fields": [
      { "name": "Require Audience", "value": "true" },
      { "name": "Require Expiration Time", "value": "true" },
      { "name": "Allowed Clock Skew", "value": "10" },
      { "name": "Max Future Validity", "value": "60" }
    ],
    "tables": [
      {
        "name": "Allowed Issuers",
        "rows": [{
          "fields": [
            { "name": "Issuer", "value": "https://spire-oidc.example.org" },
            { "name": "JWKS URL", "value": "https://spire-oidc.example.org/keys" }
          ]
        }]
      },
      {
        "name": "Allowed Audiences",
        "rows": [{
          "fields": [
            { "name": "Audience", "value": "https://pingfederate.example.com" }
          ]
        }]
      }
    ]
  },
  "attributeContract": {
    "coreAttributes": [{ "name": "sub" }]
  }
}
```

Result

When the AI agent needs to act on behalf of a user, it sends a token exchange request to PingFederate:

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

grant_type=urn:ietf:params:oauth:grant-type:token-exchange
&subject_token=eyJhbG...                        # User's identity assertion (JWT)
&subject_token_type=urn:ietf:params:oauth:token-type:jwt
&actor_token=eyJhbG...                          # Agent's identity token (JWT)
&actor_token_type=urn:ietf:params:oauth:token-type:jwt
&client_assertion_type=urn:ietf:params:oauth:client-assertion-type:jwt-bearer
&client_assertion=eyJhbG...                     # Agent's client authentication JWT
```

PingFederate processes the request as follows:

1. Authenticates the client by validating the `client_assertion` against the agent's registered JWKS.
2. Evaluates the policy and checks that the token types match a configured processor mapping.
3. Validates the subject token with the configured token processor, checking signature, issuer, audience, and expiration.
4. Validates the actor token with the configured token processor.
5. Applies the attribute mapping and signs the access token with the ATM.

PingFederate responds with the access token:

```
{
  "access_token": "eyJhbG...",
  "token_type": "Bearer",
  "expires_in": 300,
  "issued_token_type": "urn:ietf:params:oauth:token-type:access_token"
}
```

The `access_token` is a standard JWT (`typ: at+jwt`) enriched with the claims described in Token exchange architecture.

If SPIFFE is configured, the issued access token carries the agent's full SPIFFE ID in the `act` claim:

```
{
  "sub": "alice@example.com",
  "act": {
    "sub": "spiffe://example.org/ns/ai-agents/sa/expense-agent"
  },
  "scope": "expenses:read tools:list"
}
```

This SPIFFE ID is useful as it encodes the trust domain (`example.org`), namespace (`ai-agents`), and service account (`expense-agent`), which downstream services can use to make decisions. For example, PingAuthorize could enforce that only agents in the `ai-agents` namespace can access certain data.

Backend services that receive delegation access tokens validate them as standard JWTs:

1. Fetch the PingFederate JWKS from the well-known endpoint (`/.well-known/openid-configuration`).

2. Verify the signature against the PingFederate public key.
3. Validate standard claims: `iss`, `aud`, `exp`.
4. Verify `typ: at+jwt` per RFC 9068.
5. Enforce permissions according to the `scope` claim.
6. Log the delegation chain using `sub` (user) and `act.sub` (agent) for audit.

Services can also make attribute-based decisions using custom claims. For example, a `department` claim to allow only Finance department users to access budget data.

PingFederate provides the authorization infrastructure that makes AI agents enterprise-ready: every action is authorized and traceable, and every sensitive operation requires human consent.

Next steps

While application-level JWT validation is sufficient for many deployments, you can offload and centralize access token enforcement using other Ping Identity products:

PingGateway

Deployed as a reverse proxy in front of backend resources, PingGateway can validate access tokens with PingFederate token introspection (RFC 7662) before requests reach the application. It can also enforce MCP protocol-level rules, ensuring that only well-formed JSON-RPC requests carrying valid tokens are forwarded to downstream services. This removes token validation logic from application code entirely.

PingAuthorize

For fine-grained, attribute-based authorization, PingAuthorize acts as a Policy Decision Point (PDP). PingGateway or other policy enforcement points send the access token claims, including `sub`, `act`, `scope`, and `department`, to PingAuthorize through its sideband API. PingAuthorize evaluates these attributes against centrally managed policies and returns permit or deny decisions. This enables rules such as "only users in the Finance department can approve expenses over $1,000" or "agent X can only read expenses, never approve them" without modifying application code.

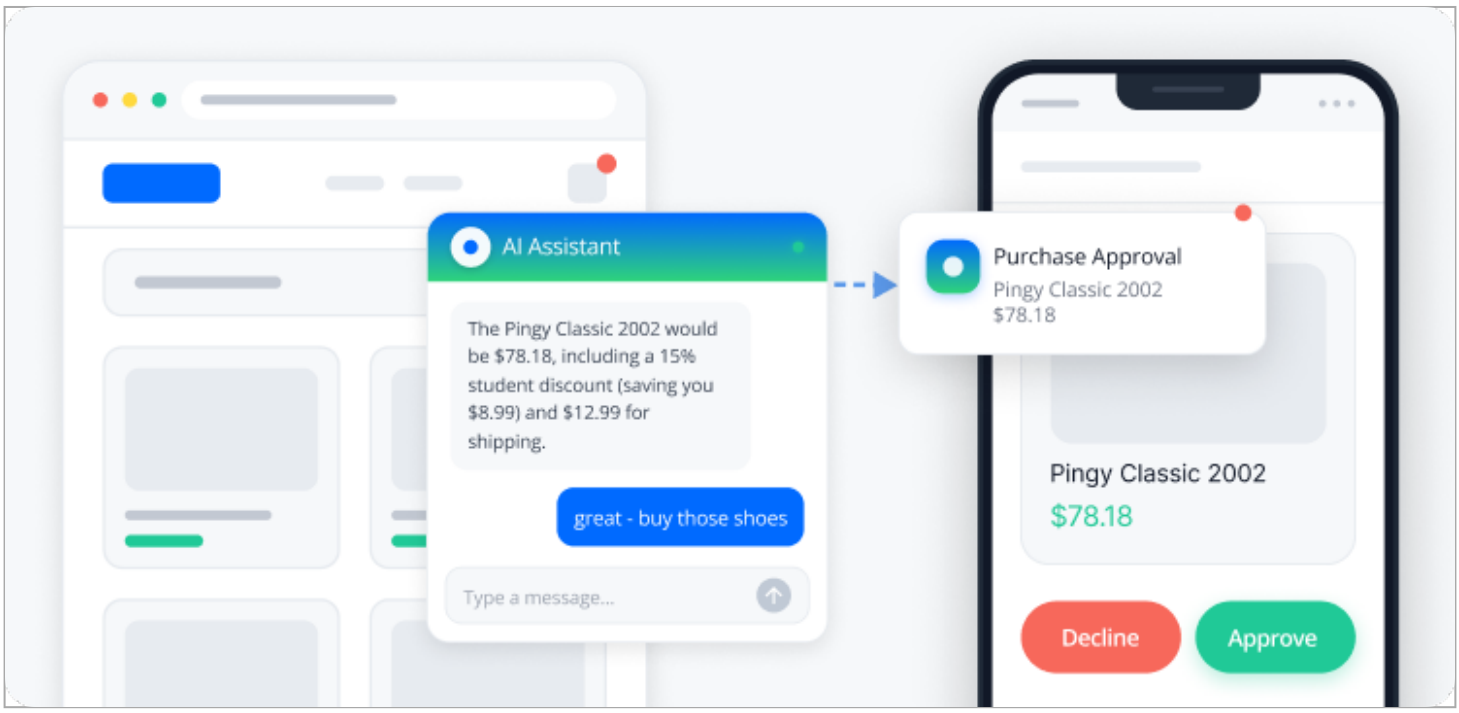
PingAccess

Acts as a Backend-for-Frontend (BFF) reverse proxy between the user's browser and backend services. PingAccess manages the user's OAuth session (tokens stored server-side, browser receives only an encrypted HttpOnly cookie), and can inject identity assertions into downstream requests using identity mapping. This keeps user tokens invisible to the browser and provides a clean separation between user authentication and agent delegation.

Together, these products enable a layered enforcement model: PingAccess secures the user session, PingFederate issues constrained delegation access tokens, PingGateway enforces protocol and token validation at the edge, and PingAuthorize applies fine-grained authorization policies, all without requiring backend resources to implement complex security logic.

Authorize an AI Agent to Perform Tasks on Your Behalf

> **Note**
>
> This introduces our tutorial series. Check back often for new tutorials.



In this tutorial, a digital assistant agent helps customers research and purchase shoes from an online store. You'll securely authorize your agent to perform tasks both autonomously and with explicit user approval. The agent will receive an access token it can then use to call tools and other agents. You'll configure this using OAuth 2.0 and OpenID Connect (OIDC) flows with PingOne Advanced Identity Cloud as the identity provider and authorization server.

This supports the following Identity for AI principles:

- AI agents must always act on behalf of a user and never impersonate them.
- AI agents should never directly prompt an end user for their credentials, but instead be granted tokens with least-privileged scopes after the user approves the transaction.

The agent will need to access tools such as "Get Latest Prices" which do not require user authorization but should still be secured. These are autonomous actions. This flow is detailed in Authorize autonomous actions with the Client Credentials flow.

When a customer wants the agent to assist with a user-specific task like shipping and purchasing, the agent accesses tools such as "Place Order" to act on behalf of the customer and use their information and payment details. The customer is sent a push notification to approve the transaction. These are on-behalf-of actions with a human-in-the-loop (HITL). This flow is detailed in Authorize on-behalf-of actions with CIBA.

Goals

- Configure Advanced Identity Cloud identities for use by your existing AI agent
- Configure the sign-on flows in Advanced Identity Cloud
- Test the flows to receive an access token

- Integrate the flows within your sample agent, allowing your agent to initiate authorization

What you'll do

In this tutorial, you will:

1. Create demo identities in Advanced Identity Cloud to represent your end user and agent owner.
2. Create an application in Advanced Identity Cloud for your AI agent.
3. Authorize autonomous actions with the Client Credentials flow.
4. Authorize on-behalf-of actions with CIBA.

Agent architecture for Advanced Identity Cloud integration

Before integrating the OAuth 2.0 flows, consider that AI agents can be built using many different architectures. Some developers use frameworks like LangChain or LlamaIndex to manage prompts and tools, while others write custom agents in plain code for maximum control.

You can define tools locally within your agent's code or manage them externally with an MCP (Model Context Protocol) server. You can also build some agents using hosted services like n8n or Zapier.

This tutorial focuses on a common and transparent pattern: a custom AI agent application written in TypeScript and run using Node that uses locally-defined tools.

This demonstrates the core security integration with Advanced Identity Cloud without the added abstraction of a specific framework or tool server. It shows how an agent can securely manage its own credentials to initiate flows and handle short-lived, access-controlled tokens that can be verified by the APIs it needs to access.

Core components of your sample agent

This tutorial assumes that you have a working prototype AI agent such to add authorization calls to. The following is a list of components your agent should have.

Agent

The main agent logic that provides user requests to an LLM service together with a system prompt, conversation history, and information about the available tools. The LLM indicates when a tool should be invoked and with what arguments.

Tool orchestrator

The tool orchestrator collates and presents tool definitions for the agent logic. In this example, it contains mappings between available tools and the authentication types and scopes that are required for the tool to interact with the APIs it relates to.

Example: Permission-aware tool configuration

```
const TOOL_AUTH_CONFIG = {
  'getProducts': null,                                                    // Public access
  'getAvailableDeals': { type: 'client_credentials', scope: 'prices' }, // Business API access
  'getTailoredQuote': { type: 'ciba', scope: 'address' },                // User address data
  'purchaseItem': { type: 'ciba', scope: 'payment' },                    // Transaction consent
};
```

This mapping enables the tool orchestrator to automatically determine the level of authorization required before executing any tool, making the security integration transparent to the main agent logic.

Advanced Identity Cloud authentication service

A dedicated module within the agent responsible for all communication with Advanced Identity Cloud. It handles both the Client Credentials and CIBA flows, abstracting the OAuth 2.0 complexity from other components.

Example: Clean abstraction for agent components

```
class PingOneAuthService {
  async getClientCredentialsToken(scope: string): Promise<string>
  async getCibaToken(scope: string, context?: string): Promise<string>
}
```

Tools

The functions that perform actions, such as calling internal company APIs. Tools receive the appropriate access tokens from the tool orchestrator and use them to authenticate with the protected resources.

> **Note**
>
> The agent application will have its own `client_id` and `client_secret` to authenticate with Advanced Identity Cloud. Do not hard code these sensitive credentials in the source code. The standard practice, which we assume in our code snippets, is to store them securely using environment variables or a dedicated secrets management service, such as AWS Secrets Manager or HashiCorp Vault.

Before you begin

The tutorial assumes you have:

- A basic understanding of PingOne Advanced Identity Cloud, OAuth 2.0, and OpenID Connect grant flows.
- Access to an Advanced Identity Cloud tenant.
- The PingID iOS or Android app configured to demonstrate the CIBA flow.
- The Push Notification Service configured. You can find configuration instructions in tasks 1 to 3 in Login with MFA using push notifications in the Advanced Identity Cloud documentation.

- A working prototype AI agent to add authorization calls to. You can use a custom agent as shown in this tutorial, or one written with a framework like LangGraph.

> **Note**
>
> If you do not have an AI agent that you can extend with the OAuth 2.0 flows, you can try out the flows in cURL or Postman.

Tasks

Create demo identities

1. In the Advanced Identity Cloud admin console, go to **Identities**, and then click **Manage**.
2. Click **Alpha realm - users**.
3. Create two identities: one for the end user and one for the app owner.

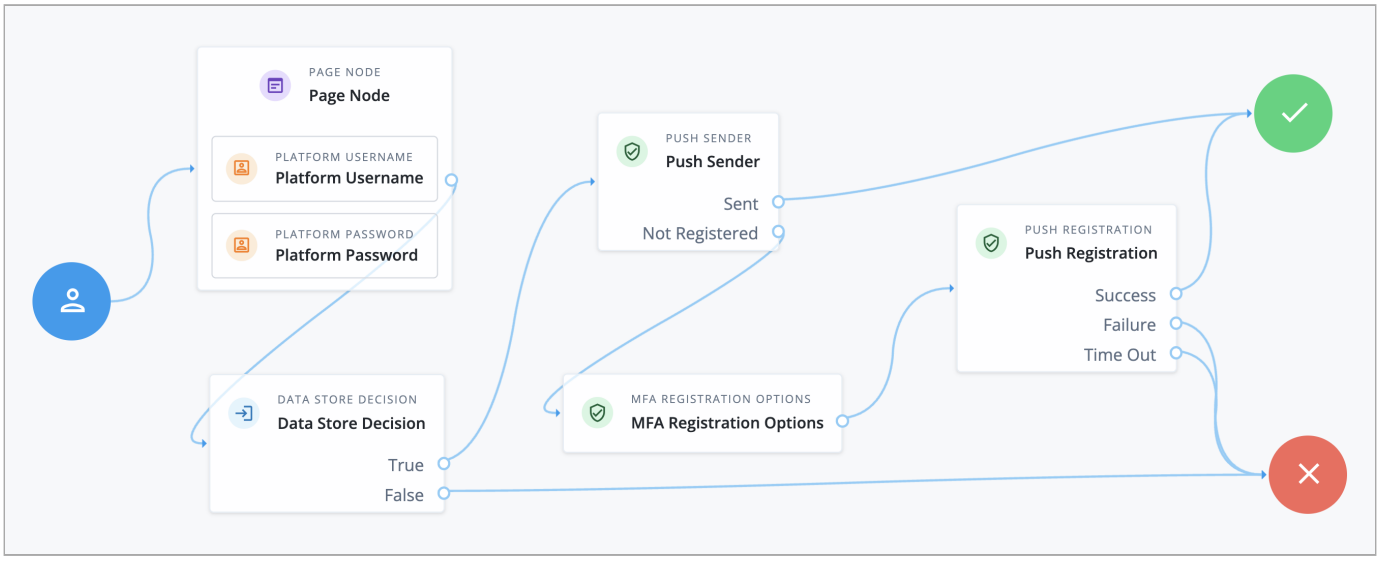
> **Note**
>
> You'll use the end user to register for push notifications with a device. You'll use the app owner when setting up the OAuth 2.0 client.

   1. Click **+ New Alpha realm - User** and create the end user identity using the form:
      - Username: end.user
      - First Name: End
      - Last Name: User
      - Email Address: end.user@example.com
   2. Click **Save**.
   3. Click **+ New Alpha realm - User** and create the app owner identity using the form:
      - Username: app.owner
      - First Name: App
      - Last Name: Owner
      - Email Address: app.owner@example.com
   4. Click **Save**.

   Learn more in Create test users and roles in the Advanced Identity Cloud documentation.

4. Create a Push Registration journey to allow the end user to register for push notifications, as the CIBA consent message will be delivered through the PingID mobile app.

   The following is a minimum viable journey:



Learn more in Journeys and Push authentication journeys in the Advanced Identity Cloud documentation.

5. Test the journey by running the journey's **Preview URL** in a new incognito browser window as an end user.

Create an application

In this task, you'll create an application that provisions an OAuth 2.0 client used by AI agent to request authorization.

1. In the Advanced Identity Cloud admin console, go to **Applications** and click **+ Custom Application**.
2. When prompted for **Sign-in Method**, click **OIDC - OpenID Connect** and then click **Next**.
3. When prompted for **Application Type**, click **Web** and then click **Next**.
4. Enter a human-readable **Name** for your application, such as **AI Agent**. In the **Owners** list, select the app owner identity. Click **Next**.
5. Enter a **Client ID**, such as `ai-agent`, and **Client Secret** and save them for later.

> **Note**
>
> The AI agent app will use the client secret to invoke OAuth 2.0 flows.

6. Click **Create Application**.

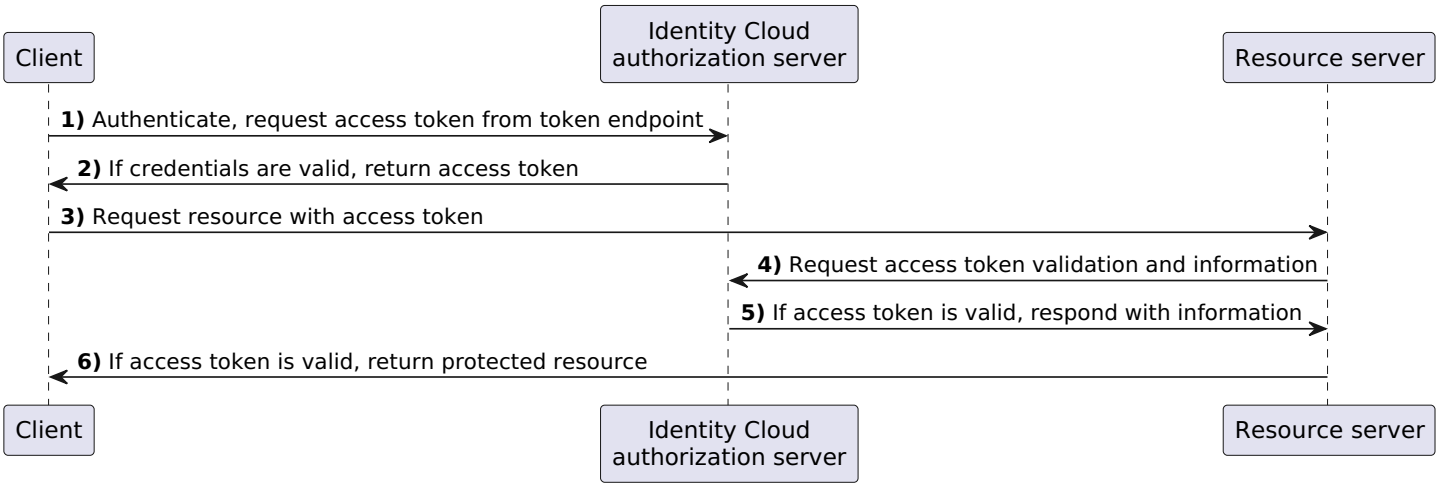
Learn more about applications in Register a custom or SSO application in the Advanced Identity Cloud documentation.

Next, you'll configure the client for the required OAuth 2.0 flows.

Authorize autonomous actions with the Client Credentials flow

In this task, you'll enable your agent to receive an access token on its own in cases where human authorization is not required. For example, accessing secured business APIs using tools like "Get available deals" that should be protected but don't require specific user authorization.

The following diagram illustrates the authorization flow:



Learn more about this flow in Client credentials grant in the Advanced Identity Cloud documentation.

1. Configure your application

    1. In the Advanced Identity Cloud admin console, go to **Applications** and select the application you created in Create an application.

    2. Click the **Sign On** tab and review the **Grant types** list. Add **Client Credentials** if it isn't already listed.

    > **Note**
    >
    > This allows the client to request authorization by providing the client ID and secret you created earlier.

    3. Review the **Scopes** list and add a new scope called `prices`.

    > **Note**
    >
    > The token will use this scope when requesting pricing information.

    4. Click **Save**.

2. Verify the authorization flow by making the following REST call to get an access token that your agent could use with a tool endpoint (not provided):

    ```
    curl \
    --request POST \
    --user '<client-id>:<client-secret>' \
    --data "grant_type=client_credentials" \
    --data "scope=prices" \
    "https://<tenant-env-fqdn>/am/oauth2/realms/root/realms/alpha/access_token"
    ```

    Result:

    Advanced Identity Cloud returns an access token for your agent to use:

```
{
  "access_token": "<access-token>",
  "scope": "write",
  "token_type": "Bearer",
  "expires_in": 3599
}
```

3. Integrate with your agent

    1. Implement the Advanced Identity Cloud token request by adding a function to your agent's authentication service.

        The following example shows how you can program an agent written in Node to fetch it's own access token.

        It uses the following environment variables for Advanced Identity Cloud integration.

Environment variable	Value
`PINGONE_TOKEN_ENDPOINT`	The token endpoint for your Advanced Identity Cloud environment, for example https://<your-tenant>.forgeblocks.com/am/oauth2/realms/alpha/access_token
`PINGONE_CLIENT_ID`	The client ID for the client created previously, for example `your-agent-client-id`
`PINGONE_CLIENT_SECRET`	The client secret for your client, for example `your-agent-client-secret`

        Show example

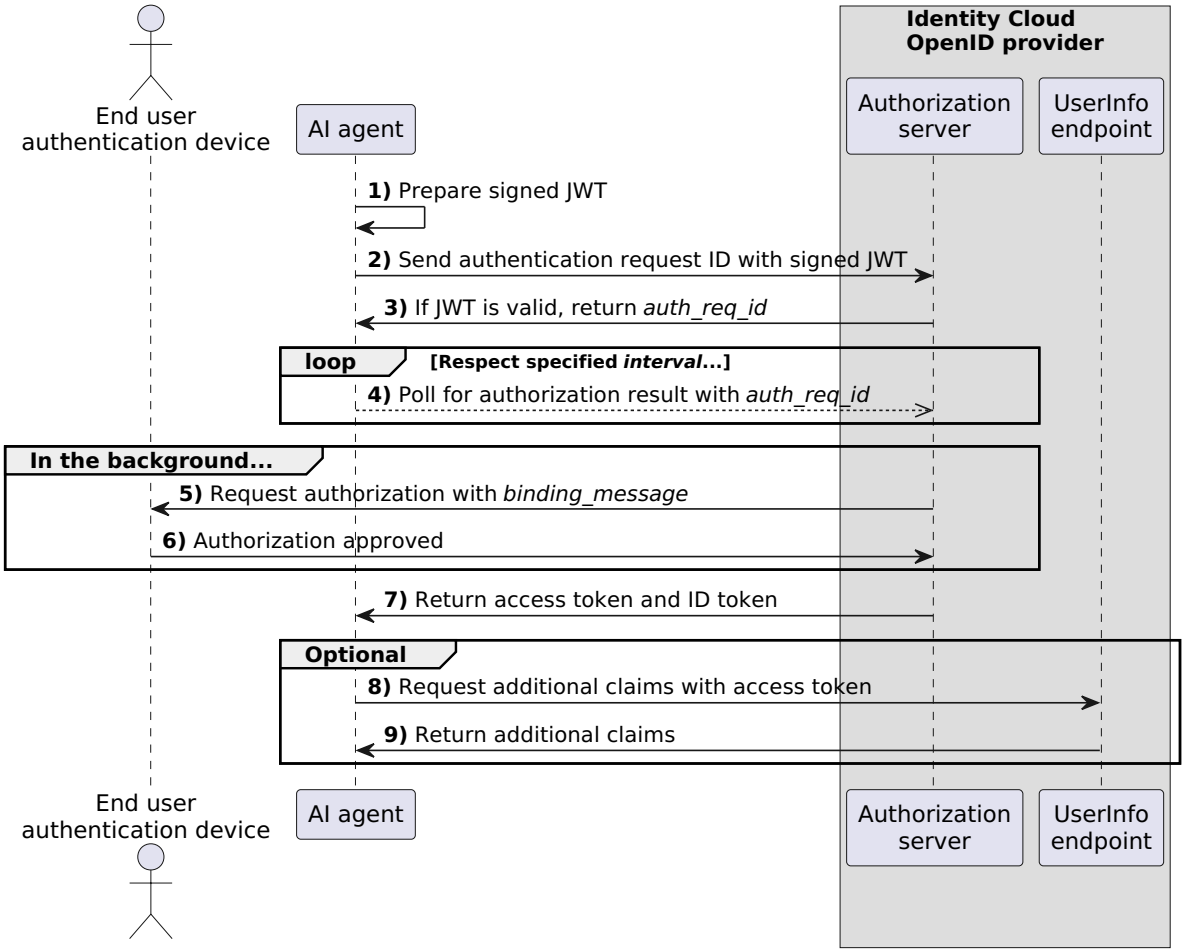
```
/**
 * Requests an access token using Client Credentials flow
 * This is used for autonomous actions that don't require user permission
 */
async function getClientCredentialsToken(scope: string): Promise<string> {
  console.log(`Requesting client credentials token for scope: ${scope}`);

  // Create Basic auth credentials from environment variables
  const credentials = Buffer.from(
    `${process.env.PINGONE_CLIENT_ID}:${process.env.PINGONE_CLIENT_SECRET}`
  ).toString('base64');

  // Make token request to P1AIC
  const response = await fetch(process.env.PINGONE_TOKEN_ENDPOINT, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/x-www-form-urlencoded',
      // Use Basic auth with base64 encoded client_id:client_secret
      'Authorization': `Basic ${credentials}`,
    },
    body: new URLSearchParams({
      'grant_type': 'client_credentials',
      scope, // Request specific scope (e.g., 'prices' for deals API)
    }),
  });

  const data = await response.json();
  console.log('Client credentials token obtained successfully');

  return data.access_token;
}
```

With this structure, the agent can now securely and efficiently call autonomous tools.

Authorize on-behalf-of actions with CIBA

In the previous task, you enabled an agent to request an access token for itself to use in a tool call which doesn't require human authorization.

Next, you'll implement a flow that allows your agent to call a tool on a human's behalf and with their explicit permission through the form of a push notification.



Learn more about this flow in Backchannel request grant in the Advanced Identity Cloud documentation.

1. Configure your application:

    1. In the Advanced Identity Cloud admin console, go to **OAuth2 Clients** and select the application you created in Create an application.

    2. Click the **Sign-on** tab.

    3. Review the **Grant types** list and add **Back Channel Request**.

    > **Note**
    >
    > This will allow the client to request authorization from a human using the OIDC CIBA flow.

    4. Review the **Scopes** list and add a new scope called `address`.

    > **Note**
    >
    > The token will use this scope when requesting the end user's information.

    5. Configure access to the relying party's (RP's) public keys so that Advanced Identity Cloud can verify JWT signatures:

        1. In the **General settings** section, click **Show advanced settings** and then click **Signing and Encryption**.

2. In the **Public key selector** list, select **JWKs**.
3. In the **JSON Web Key** field, enter a JWK set similar to the following:

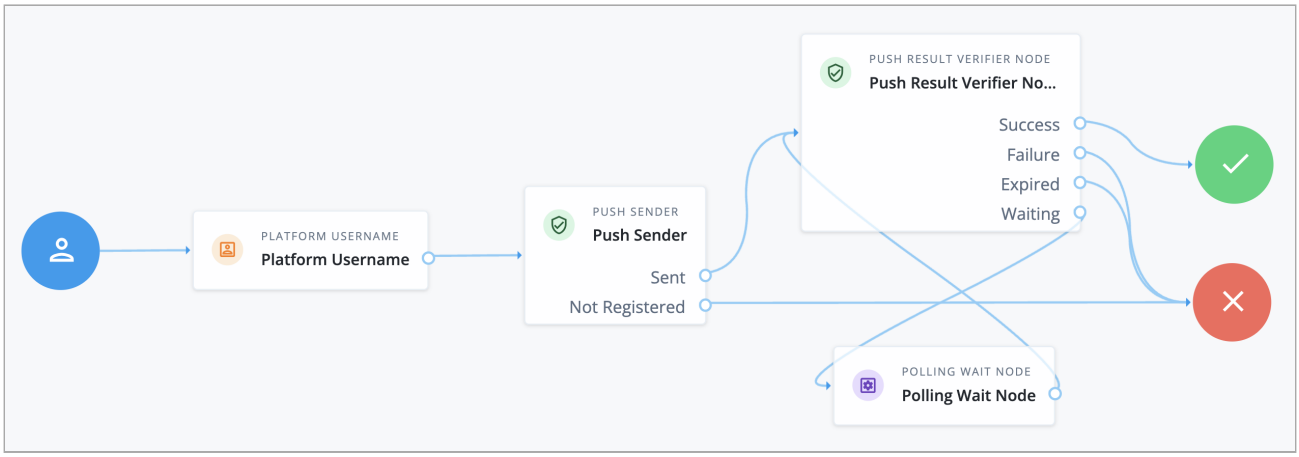
```
{
  "keys": [
    {
      "kty": "EC",
      "use": "sig",
      "crv": "P-256",
      "kid": "myCIBAKey",
      "x": "m0CkpWpZyGu-FLRLjCROVGC7Fwm5vGt8Lm3HhYU4ylg",
      "y": "U8NMtO-C2c3yhu2I_AzAELttmaitefPNPQaIJxvTCHk",
      "alg": "ES256"
    }
  ]
}
```

Learn more about Generating JWKs in the Advanced Identity Cloud documentation.

4. Click **Save**.

2. Configure journeys:

1. Create a journey named **Agent Push Sender** similar to the following:



This minimum viable journey uses the following nodes:

- Platform Username node
- Push Sender node
- Push Result Verifier node
- Polling Wait node

Learn more in Push authentication journeys in the Advanced Identity Cloud documentation.

> **Note**
>
> To verify your journey, you can test it in your browser. After entering the end user's credentials, you'll be sent a push notification.

2. In the Advanced Identity Cloud admin console, under **Native Consoles > Access Management**, go to **Realms > Realm Name > Services > OAuth2 Provider > Advanced**.
3. Make sure the **Grant Types** field includes **Back Channel Request**. Save any changes you make.
4. Associate the journey with incoming `acr_values`:
    1. Go to the **Advanced OpenID Connect** tab of the OAuth 2.0 provider configuration.
    2. In the **OpenID Connect acr_values to Auth Chain Mapping** box:
        1. Set **Key** to the value that will be passed in through the `acr_values` claim of the incoming CIBA request, such as `push`.
        2. Click **Add**.
        3. Set **Value** to the name of your journey, such as **Agent Push Sender**.
    3. Save your changes.

    Learn more about the acr claim in the Advanced Identity Cloud documentation.

3. Verify your integration:

You can now test the CIBA flow manually to ensure your Advanced Identity Cloud configuration is working before integrating with your agent. Execute the following REST calls to initiate the CIBA flow which should send a push notification to the end user. If approved, this will allow retrieval of an access token to call a tool on the user's behalf.

1. Prepare a signed JWT with the required claims in the payload.

    The example uses the following values:

Variable	Value
`exp`	A current timestamp plus 15 minutes. You can generate this in your terminal with `echo $(($(date -u +%s) + 899))`.
<rp-client-id>	Your client ID, such as `ai-agent`.
<end-user-id>	The username of the end user you created in Create demo identities, such as `end.user`.

```
{
  "aud": "https://<tenant-env-fqdn>:443/am/oauth2/realms/root/realms/alpha",
  "binding_message": "Allow tutorial agent to access your address data?",
  "acr_values": "push",
  "exp": 1759988402,
  "iss": "<rp-client-id>",
  "login_hint": "<end-user-id>",
  "scope": "openid address"
}
```

2. Send a POST request to the `oauth2/bc-authorize/` endpoint with the signed JWT in the payload. For example:

```
curl \
--request POST \
--user '<rp-client-id>:<rp-client-secret>' \
--data 'request=<signed-jwt>' \
'https://<tenant-env-fqdn>:443/am/oauth2/realms/root/realms/alpha/bc-authorize'
```

Learn more about generating JWTs in step 2 of Get an auth request ID in the Advanced Identity Cloud documentation.

Result:

- You'll receive a response similar to the following:

```
{
    "auth_req_id": "abc123-def456-ghi789",
    "expires_in": 120,
    "interval": 5
}
```

Make a note of the `auth_req_id` as your agent will use it to poll for the result of the flow.

- Advanced Identity Cloud sends a push notification with the `binding_message` to the end user.

3. Poll for the access token by making a request to the Advanced Identity Cloud environment to determine the outcome of the flow. For example:

```
curl \
--request POST \
--user '<rp-client-id>:<rp-client-secret>' \
--data 'grant_type=urn:openid:params:grant-type:ciba' \
--data 'auth_req_id=<auth-req-id>' \
'https://<tenant-env-fqdn>/am/oauth2/realms/root/realms/alpha/access_token'
```

Result:

After the end user has authorized the operation, Advanced Identity Cloud returns an ID token and an access token:

```
{
  "access_token": "<access-token>",
  "refresh_token": "<refresh-token>",
  "scope": "openid profile",
  "id_token": "<id-token>",
  "token_type": "Bearer",
  "expires_in": 3599
}
```

4. Integrate with your agent:

    1. Implement the CIBA flow within your agent. This involves creating the signed JWTs, initiating the CIBA request, and polling for user approval.

        The following example shows how you can program an agent written in Node to fetch it's own access token. It implements a new function, `getCibaToken`, to manage the CIBA flow.

        It uses the following environment variables for Advanced Identity Cloud integration:

Variable	Description
`PINGONE_CIBA_AUDIENCE`	Your Advanced Identity Cloud realm's OAuth 2.0 provider identifier, for example https://<your-tenant>.forgeblocks.com:443/am/oauth2/alpha
`PINGONE_CIBA_ACR_VALUES`	The authentication context reference, linking to the push notification journey you set up earlier, for example `push`
`PINGONE_CIBA_ISSUER`	The `client_id` of the client issuing the request
`PINGONE_CIBA_JWK`	The JWK for signing CIBA requests stored as JSON string
`PINGONE_CIBA_ENDPOINT`	The backchannel authorization request endpoint for your Advanced Identity Cloud environment, for example https://<your-tenant>.forgeblocks.com/am/oauth2/realms/alpha/bc-authorize

        Note

        This example requires libraries like jose for JWT signing. You must have a private key to use for signing.

        Show example

```
// Added around existing code from the Client Credentials section above

import { SignJWT, importJWK } from 'jose';

// Type definition for CIBA initiate responses
interface CibaInitiateResponse {
  auth_req_id: string;
  expires_in: number;
  interval: number;
}

/**
 * Creates a contextual binding message based on the scope and context.
 * This is displayed to users as part of the push notification
 */
function createBindingMessage(scope: string, context: string): string {
  switch (scope) {
    case 'payment':
      return context
        ? `Allow agent to place order for ${context}?`
        : 'Allow agent to place an order on your behalf?';
    case 'address':
      return 'Allow agent to access your address information?';
    default:
      throw new Error(`Unknown CIBA scope requested: ${scope}`);
  }
}

/**
 * Creates and signs a CIBA request JWT
 */
async function createCibaRequestJWT(scope: string, userId: string, context: string): Promise<string> {
  const bindingMessage = createBindingMessage(scope, context);
  const currentTime = Date.now();
  const iat = Math.floor(currentTime / 1000);
  const exp = iat + 600; // Have the request be valid for 10 minutes

  const payload = {
    aud: process.env.PINGONE_CIBA_AUDIENCE,
    binding_message: bindingMessage,
    acr_values: process.env.PINGONE_CIBA_ACR_VALUES, // e.g., "push"
    exp, // expiration time for the authentication request
    iss: process.env.PINGONE_CIBA_ISSUER,
    login_hint: userId,
    scope: `openid ${scope}`,
    iat, // time at which the request was created
    nbf, // time before which the request is unacceptable
  };

  // Import JWK and sign JWT
  const jwk = JSON.parse(process.env.PINGONE_CIBA_JWK);
  const privateKey = await importJWK(jwk, 'ES256');
  const jwt = await new SignJWT(payload)
    .setProtectedHeader({
      alg: 'ES256',
      typ: 'JWT',
      kid: jwk.kid,
    })
    .sign(privateKey);
  return jwt;
```

```
}

/**
 * Initiates CIBA flow with P1AIC
 */
async function initiateCibaFlow(requestJWT: string): Promise<CibaInitiateResponse> {
  console.log('Initiating CIBA flow with P1AIC');

  // Create Basic auth credentials
  const credentials = Buffer.from(
    `${process.env.PINGONE_CLIENT_ID}:${process.env.PINGONE_CLIENT_SECRET}`
  ).toString('base64');

  const response = await fetch(process.env.PINGONE_CIBA_ENDPOINT, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/x-www-form-urlencoded',
      'Authorization': `Basic ${credentials}`,
    },
    body: new URLSearchParams({
      'request': requestJWT,
    }),
  });

  const data = await response.json() as CibaInitiateResponse;
  return data;
}

/**
 * Polls for CIBA token after user approval
 */
async function pollForCibaToken(authReqId: string, interval: number): Promise<string> {
  console.log('Waiting for user approval...');

  const maxAttempts = Math.ceil(600 / initialInterval); // 10 minutes max
  let attempts = 0;

  // Create Basic auth credentials
  const credentials = Buffer.from(
    `${process.env.PINGONE_CLIENT_ID}:${process.env.PINGONE_CLIENT_SECRET}`
  ).toString('base64');

  while (attempts < maxAttempts) {
    // Wait before polling
    await new Promise(resolve => setTimeout(resolve, interval * 1000));
    attempts++;

    console.log(`Checking for approval (attempt ${attempts}/${maxAttempts})`);

    try {
      const response = await fetch(process.env.PINGONE_TOKEN_ENDPOINT, {
        method: 'POST',
        headers: {
          'Content-Type': 'application/x-www-form-urlencoded',
          'Authorization': `Basic ${credentials}`,
        },
        body: new URLSearchParams({
          'grant_type': 'urn:openid:params:grant-type:ciba',
          'auth_req_id': authReqId,
        }),
      });
```

```
      const data = await response.json();
      if (data.access_token) {
        // Success! The user approved
        return data.access_token;
      }

      if (data.error === 'authorization_pending') {
        // Continue polling
        continue;
      } else if (data.error === 'access_denied') {
        throw new Error('User denied the permission request');
      } else if (data.error) {
        throw new Error(`CIBA error: ${data.error}`);
      }
    }

    throw new Error('CIBA flow timed out - user did not respond');
  }
}

/**
 * Complete CIBA flow to get user permission token
 */
async function getCibaToken(scope: string, userId: string, context: string): Promise<string> {
  console.log(`Starting CIBA flow for scope: ${scope} and context ${context}`);

  // 1. Create signed JWT request
  const requestJWT = createCibaRequestJWT(scope, userId, context);

  // 2. Initiate CIBA flow
  const cibaResponse = await initiateCibaFlow(requestJWT);

  // 3. Poll for user approval
  const token = await pollForCibaToken(cibaResponse.auth_req_id, cibaResponse.interval);

  return token;
}

// Export the CIBA function along with client credentials
export { getClientCredentialsToken, getCibaToken };
```

2. Update the tool orchestrator to obtain tokens using CIBA as needed.

   Show example

```
import { getClientCredentialsToken, getCibaToken } from './P1AICAuthService';
import { getAvailableDeals } from './tools/getAvailableDeals';
import { placeOrder } from './tools/placeOrder';

// Tool configuration mapping tools to their auth requirements
const TOOL_AUTH_CONFIG = {
  'getAvailableDeals': { type: 'client_credentials', scope: 'prices' }, // Business API access
  'placeOrder': { type: 'ciba', scope: 'payment' }, // Requires user permission
  //... other tool to authentication mappings
};

/**
 * Permission-aware tool execution that handles different auth types
 * This is called by the main agent when the LLM decides to use a tool
 */
async function executeToolWithPermissions(toolName: string, args: Record<string, unknown>, userId:
string) {
  // Look up what authorization this tool requires
  const authConfig = TOOL_AUTH_CONFIG[toolName];

  let token
  if (authConfig?.type === 'client_credentials') {
    // Use Client Credentials to get access token from P1AIC for this scope
    token = await getClientCredentialsToken(authConfig.scope);
  } else if (authConfig?.type === 'ciba') {
    // Use context from args to create meaningful permission message
    const context = args.productName;
    token = await getCibaToken(authConfig.scope, userId, context);
  }

  // Execute the tool with the token
  return executeTool(toolName, args, userId, token);
}

/**
 * Central tool execution function that routes to specific implementations
 * This is where you add new tools as your agent capabilities grow
 */
private async executeTool(toolName: string, args: any, userId: string, token?: string): Promise<string>
{
  switch (toolName) {
    case 'getAvailableDeals':
      return getAvailableDeals(token);
    case 'placeOrder':
      return placeOrder(args, userId, token);
    default:
      throw new Error(`Unknown tool: ${toolName}`);
  }
}

// Export for use by your main agent logic
export { executeToolWithPermissions };
Step 3: Add the placeOrder tool
The placeOrder tool is implemented similarly to the deals tool, but it will be calling a different and
more sensitive API endpoint.

// src/tools/placeOrder.ts

export async function placeOrder(args: { productId: string; quantity: number }, userId: string, token:
string): Promise<unknown> {
```

```
    try {
      const response = await fetch('https://api.example.com/purchase', {
        method: 'POST',
        headers: {
          'Authorization': `Bearer ${accessToken}`
        },
        body: JSON.stringify({
          productId: args.productId,
          userId,
          quantity: args.quantity,
        }),
      });
      const data = await response.json();
      return data;
    } catch(error) {
      return `Failed to process purchase: ${error.message}`;
    }
  }
```

Your agent can now make on-behalf-of tool calls for the end user using the access token that was returned after the user authorized the operation.

Result

You've successfully authorized your agent to access protected resources using a secure pattern for both autonomous and on-behalf-of actions in a modern AI agent architecture.

Your agent is now onboarded and has a token it can use to access tools and complete tasks.

Key Identity for AI Terminology

Learn about key terms related to artificial intelligence (AI) concepts and Identity for AI, a solution for securing, managing, and governing AI systems and data.

Agent types

Learn more in What Are AI Agents? and Agent Types.

Digital assistant

A reactive agent that responds to direct commands from a user to perform specific tasks based on explicit prompts. Digital assistants can be thought of as front-line workers managed by an organization that lie mainly or completely within the organization's trust boundary. They can be both customer-facing and deployed for the workforce:

- **Consumer digital assistants** interact directly with customers or end users to provide services, answer questions, and perform tasks. These agents lie mainly within the organization's trust boundary, interacting with consumers outside the boundary. Examples of this type of *our agent works for you* digital assistant include a digital mortgage advisor, travel agent, or chatbot.
- **Workforce digital assistants** assist employees by automating routine tasks and providing information that improves the efficiency of the organization. These agents lie completely within the organization's trust boundary. Examples of this type of *our agent works internally for employees* digital assistant include an HR assistant that helps with time-off requests or a payroll assistant that answers pay-related questions.

Digital worker

A managed agent that's more autonomous than a digital assistant and designed to perform structured, repetitive tasks to automate business processes and workflows. These agents lie completely within the organization's trust boundary. Examples of this type of *our agent solves tasks autonomously for our organization* digital worker include a finance assistant, Salesforce Agentforce, and Lindy.ai.

Personal agent

A proactive and autonomous agent that acts on behalf of a user to achieve the user's goals. This type of agent is highly personalized and acts as a proxy for the user on the user's device. Personal agents aren't managed by your organization and lie outside of your organization's trust boundary. Examples of this type of *your agent works for you* personal agent include agents that book travel, help with investments or shopping, and large language models (LLMs) such as ChatGPT, Claude, and Google Gemini.

AI and ML terms

Agent card

A JSON metadata document used in the agent2agent (A2A) protocol that describes an AI agent's purpose, skills, endpoint URL, and authentication requirements for discovery and collaboration.

Artificial intelligence (AI)

The simulation or replication of human-like intelligence by machines, especially computer systems, to perform tasks such as learning, reasoning, problem-solving, and decision-making.

Computer-using agent (CUA)

An agent that interacts directly with graphical user interfaces (GUIs) and command-line interfaces (CLIs), such as web browsers, buttons, menus, and text fields, by emulating human actions. Learn more in Computer-Using Agents (CUAs).

Context poisoning (prompt injection)

A security vulnerability where a malicious attacker manipulates the input context or prompts provided to an AI agent, subtly altering the agent's behavior and leading to unintended outputs.

Explainable AI (XAI)

The field of AI that focuses on making AI decisions and outputs understandable to humans, addressing the *black box* problem of complex models.

Generative artificial intelligence (GenAI)

AI systems that can create new content, such as text, images, audio, or video, based on learned patterns from existing data. Examples include LLMs, such as ChatGPT, Claude, or Google Gemini, and image generation models.

Large language model (LLM)

A type of deep learning AI model designed to understand and generate human-like text by processing vast amounts of textual data. LLMs, such as OpenAI's GPT series, are capable of performing various natural language processing tasks, including translation, summarization, and question answering.

Machine learning (ML)

A subset of AI focused on algorithms and statistical models that enable computers to learn from data and make predictions or decisions without explicit programming for specific tasks.

Managed agents

AI agents that are developed, provisioned, centrally controlled, and governed by an organization, ensuring alignment with established security policies and compliance standards.

Natural language processing (NLP)

A branch of AI that focuses on the interaction between computers and human (natural) languages, enabling machines to understand, interpret, and generate human language in a meaningful and useful way.

Retrieval-augmented generation (RAG)

An AI technique that combines LLMs with external knowledge sources, such as databases or documents. This enhances the model's ability to generate accurate and contextually relevant responses by retrieving pertinent information during the generation process.

IAM terms

Agentic identity

The unique identity of an AI agent that acts autonomously and makes decisions. Agentic identity encompasses the agent's permissions, delegated authority, behavioral patterns, credentials, and roles in an identity and access management (IAM) framework.

AI for identity

The application of AI and ML techniques to enhance IAM processes such as user authentication, fraud detection, and access control.

Attribute-based access control (ABAC)

An authorization model that grants or denies access to resources based on a combination of attributes associated with users, AI agents, resources, and current environmental conditions (context).

Authenticated delegation

The process of a human user securely authorizing a specific AI agent to access digital services or interact with other agents on the human's behalf, with verifiable permissions and scope.

Bring your own agent (BYOA)

A security model in which employees or customers bring their own AI agents into an organization's environment. BYOAs aren't centrally managed or governed by the organization and might not adhere to the organization's policies and standards.

Client-Initiated Backchannel Authentication (CIBA)

An OAuth 2.0-based authentication protocol that allows an AI agent to get human approval for a sensitive action. CIBA allows the AI agent to initiate an authentication request to an identity provider (IdP) without direct, real-time user interaction, enabling human-in-the-loop (HITL) for critical tasks.

Delegated authority

A model that enables an AI agent to act on behalf of a user without using their personal credentials, allowing the agent to serve as a secure and limited proxy for the user.

Dynamic Client Registration (DCR)

An OAuth 2.0 protocol enabling AI agents to programmatically register themselves as OAuth 2.0 clients with an identity provider (IdP) or authorization server (AS) at runtime to obtain unique credentials and define initial scopes and permissions.

Human-in-the-loop (HITL)

A security and governance approach that requires human intervention and approval for critical actions performed by AI agents, ensuring oversight and accountability when full automation isn't feasible.

Identity for AI

A comprehensive solution for securely managing, governing, and auditing the identities and access of AI systems, including autonomous agents, within an organization's digital ecosystem. Learn more in What Is Identity for AI?.

Identity lifecycle management (ILM) for AI

The process of managing an AI's identity from creation to retirement to automate the provisioning, authentication, authorization, and deprovisioning of AI identities.

Least privilege

A fundamental security principle that restricts an entity's (human or AI agent) access rights and permissions to the minimum necessary to perform its legitimate functions, reducing the risk of unauthorized access or actions.

Mutual transport layer security (mTLS)

A security protocol that ensures both the client (AI agent) and server authenticate each other using digital certificates during communication, providing a higher level of trust and security.

Non-human identity (NHI)

Any digital identity assigned to an automated system, application, or device rather than a human user to manage and secure the access of non-human entities to digital resources.

Pushed Authorization Requests (PAR)

An OAuth 2.0 extension that allows an AI agent to securely send authorization requests directly to the authorization server (AS) instead of through a user agent such as a web browser. This separates the authorization request from the final authorization URL and is useful for AI agents that operate without direct human interaction.

Rich Authorization Requests (RAR)

An extension to OAuth 2.0 that enables AI agents to include rich, detailed information about their intended actions in authorization requests, such as specific permissions, constraints, and context, allowing for more granular and dynamic access control decisions.

Trust boundary

A conceptual line that separates an organization's digital assets into different levels of trust and security. Assets inside the boundary, such as employees, digital workers, and authorized applications, are managed and controlled by the organization's policies and operate with a higher baseline of implicit trust. Untrusted assets outside the boundary, such as external customers or personal agents, aren't managed by the organization and require additional authorization and verification.

Protocols

Model Context Protocol (MCP)

An open-source standard for how AI systems, particularly LLMs, securely and dynamically interact with external tools and data sources. MCP defines an MCP server as a standardized interface that exposes a specific tool, API, or data source to an AI agent. The MCP server acts as a *smart adapter* or *universal connector* for AI. Learn more in What is Model Context Protocol (MCP)? and MCP servers and OAuth 2.0.

Agent2Agent protocol (A2A)

An open-source protocol that enables secure and standardized communication between AI agents. A2A facilitates interoperability, allowing agents to share information, authenticate each other, delegate tasks, and coordinate actions while maintaining security and trust. Learn more in What is Agent2Agent Protocol (A2A)?.