Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: 8 January 2026

J. Rosenberg
Five9
P. White
Bitwave

7 July 2025

AAuth - Agentic Authorization OAuth 2.1 Extension
draft-rosenberg-oauth-aauth-00

Abstract

This document defines the Agent Authorization Grant, an OAuth 2.1 extension allowing a class of Internet applications - called AI Agents - to obtain access tokens in order to invoke web-based APIs on behalf of their users. In the use cases envisaged here, users interact with AI Agents through communication channels - the Public Switched Telephone Network (PSTN) or texting - which do not permit traditional OAuth grant flows. Instead, AI agents collect Personally Identifying Information (PII) through natural language conversation, and then use that collected information to obtain an access token with appropriately constrained scopes. A primary considering is ensuring that the Large Language Model (LLM) powering the AI Agent cannot, through hallucination, result in impersonation attacks.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 8 January 2026.

Copyright Notice

Table of Contents

1. Introduction

AI Agents have recently generated a significant amount of industry interest. They are software applications that utilize Large Language Models (LLM)s to interact with humans (or other AI Agents) for purposes of performing tasks. Just a few examples of AI Agents are:

* A travel AI Agent which can help users search for travel destinations based on preferences, compare flight and hotel costs, make bookings, and adjust plans

* A loan handling agent that can help users take out a loan. The AI Agent can access a users salary information, credit history and then interact with the user to identify the right loan for the target use case the customer has in mind

* A shopping agent for clothing, that can listen to user preferences and interests, look at prior purchases, and show users different options, ultimately helping a user find the right sports coat for an event

Technically, AI agents are built using Large Language Models (LLMs) such as GPT4o-mini, Gemini, Anthropic, and so on. These models are given prompts, to which they can reply with completions. To creates an AI agent, complex prompts are constructed that contain relevant documentation, descriptions of tools they can use (such as APIs to read or write information), grounding rules and guidelines, along with input from the user. The AI Agent then makes decisions about

whether to interact further with the user for more information, or whether to invoke an API to retrieve information or perform an action.

Please refer to [I-D.rosenberg-ai-protocols] for a framework and description on the protocols relevant in the design of AI Agents.

Users interact with AI Agents through communication channels. These can be dedicated mobile or web applications that provide a chat or voice communications function.

This document focuses on a specific use case, which is the usage of AI Agents for customer support. In this use case, a business is offering the AI agent to its customers. In this use cases, it is common to access the AI Agent via a widget embedded in the business' web page. Or, users can access these AI Agents through the Public Switched Telephone Network (PSTN). In the United States, it is commonplace for businesses to have toll-free numbers that users can dial to reach customer support. There is significant interest in allowing those calls to be handled by AI Agents. Similarly, users can communicate with these AI Agents through text channels, or through third party applications like WhatsApp or Facebook Messenger. In all of these cases, the AI agent can only interact with the user through the exchange of real-time voice or messaging channels.

In due course, the AI Agent will need to invoke APIs to perform actions or retrieve information, in which case it will require an access token that is valid for the invocation of the APIs. Because the user can only interact with the AI Agent via real-time voice or SMS, there is no ability to perform the traditional OAuth [RFC6749] grant flows used in web or mobile applications.

This is not a new problem of course. Prior to the arrival of LLMs, voice and chat bots for customer support have been interacting with users and invoking APIs to retrieve information or take actions. The common practice was, at the time of development of the bot, to configure a service account into the various systems for which API access was required. These service accounts were granted access to data for all of the potential users of the system. The bot designer would then provide service account credentials to the bot, and the bot would perform an OAuth client credentials grant flow to obtain an access token - at design time. The bot would also be designed to identify users, typically by collecting some kind of personally identifiable information (PII). At run time, when the user interacts with the bot, the bot would collect PII information from the user, and then dip into databases that mapped the PII to a user identifer, such as an accountID, username or email address. The bot would then - using its service account - access enterprise APIs to perform actions, and provide the user identifier as one of the API arguments. This allowed the bot to operate on the user’s behalf.

In this approach, the bot is highly trusted. It is trusted to properly verify the identity of the user, and it is trusted to properly invoke APIs on behalf of the user which was identified. This trust is earned because the developers of bots are employees of the IT departments (or vendors operating on their behalf) of the businesses that served the end users in question, and also own the resources and APIs that the bot was accessing.

The arrival of LLM-powered AI agents - next generation voice and chat bots - is now challenging both of these assumptions.

Firstly, though the AI Agents are still being designed by IT departments (or vendors operating on their behalf), they are now making use of LLMs for decision making. These LLMs are susceptible to hallucination, wherein the LLM asserts information that is not justified based on its inputs. If we apply the prior generation of design practices to AI Agents - we end up giving them "god-like" tokens which allow highly privileged access, and they may hallucinate information passed into those APIs, perhaps being induced to do so via malicious end users. This includes hallucination of the user identity. Consider the following examples of problems that could arise:

* A patient is interacting with a healthcare AI agent, and asks to fulfill a prescription. The LLM hallucinates the identity of the patient when invoking the prescription fulfillment API, assigning it to a different patient. Because the AI Agent has a privileged service account, the fulfillment proceeds. The patient never gets their prescription and suffers negative health consequences.

* A malicious user is attempting to steal money by tricking a financial services AI Agent into transferring funds from a target bank account into their own. The user, through prompt injection attack, convinces the AI Agent to perform the transfer. Because the AI Agent has a privileged service account, the transfer succeeds, and the malicious user is able to steal money

These are but two examples of the many problems that can arise.

The second assumption - that the APIs accessed by the AI agent were within the span of control of the enterprise IT department building the agent - is being challenged because these new AI agents are more highly capable and can be more effective in accessing a larger API surface area, including APIs that are outside of the ownership of the bot developer. This is discussed in Section 2.2 of [I-D.rosenberg-ai-protocols]. Of particular relevance is the Model Context Protocol (MCP) (https://modelcontextprotocol.io/introduction (https://modelcontextprotocol.io/introduction)) which is being driven by Anthropic, and allows AI Agents to easily access APIs across the Internet.

The solution proposed here is a new OAuth grant flow that removes the need for service-level accounts and their associated access tokens, and instead allows AI Agents to obtain low-privilege access tokens for the user's on whose behalf they are operating.

2. Framework and Requirements

The framework for the proposed grant flow is shown in the diagram below:

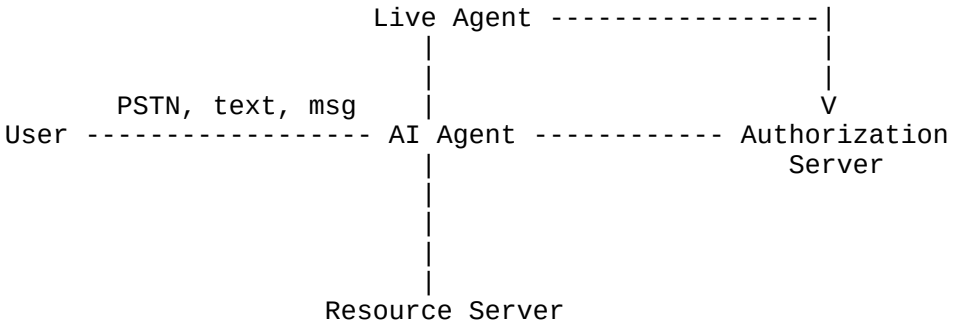

Figure 1: Framework for AAuth

The AI Agent is a software application running on a server, making use of an LLM to communicate with a user. The user communicates with the AI Agent over a communications channel that offers voice or chat

functionality. Examples include the PSTN, SMS or text messaging, or third party messaging channels such as WhatsApp or Facebook Messenger. Most notably, these channels do not provide a vehicle for typical OAuth grant flows - they only allow the exchange of real-time voice and text messages.

The AI Agent needs to access an API hosted on a resource server (RS). The AI Agent needs to obtain an access token for that interaction, which they do through an authorization server.

There are also cases where the AI agent cannot do something without a human agent (live agent) approving something. For example, in our funds transfer case, a human agent needs to be pulled in to approve, and only then can the AI agent gain access to a higher privilege token on behalf of the user.

The following requirements apply:

* It must be possible for the AI Agent to obtain an access token that only allows them to perform operations for the user that is engaged with the AI Agent

* The AI Agent must be able to attain the access token using PII that the AI Agent has collected from the user over the communications channel

* It must not be possible, through any kind of hallucination that the AI Agent might perform, to obtain an access token for a different user than the one interacting with the AI Agent

* It must be possible for a human-in-the-loop approval from live agents for higher privilege tokens

3. Solution Overview

3.1. Basic Token Flow

The solution works in much the same way this problem is solved when users interact with live customer support agents.

In the case of a user interacting with a live customer support agent, the live customer support agent will identify the user by collecting multiple unique points of PII. For example, in healthcare solutions, HIPAA requires two unique pieces of information, including full name, date of birth, last four digits of the SSN, medical record number, or phone number on file.

The proposed solution here is similar. The AI Agent is configured to interact with the user over the communication channel to collect a set of PII. Once it collects this information, it passes the information to the Authorization Server (AS) using a new authorization grant type. The AS maps this information to matching user, and if it matches, issues an access token for the user in question. Though it is still possible that the LLM will hallucinate some of this information, it is highly unlikely that it would hallucinate a combination which represents a valid, but different user. Indeed, the security of the solution depends on the sparseness of the space of mappings from the PII tuples to valid users. If most PII tuples do not map to a valid user, then it is unlikely that the LLM would take PII data that does map to a specific user, and hallucinate a set of PII data that maps to a different, but valid user. This is shown pictorially below:

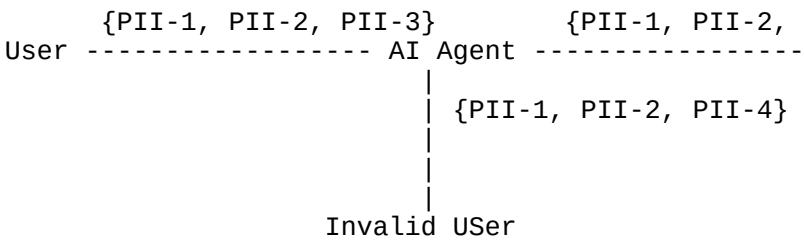

Figure 2: Hallucination Risk Mitigation with Sparse PII Mappings

Consider once again the healthcare AI Agent which is interacting with a patient. Through prompting, the AI Agent is configured to collect the user's last four digits of SSN, their full name, and their date of birth. The user provides their last-four digits of their SSN as 1234 (call this PII-1), their full name as "John Smith" (call this PII-2) and their date of birth as "April 3rd, 1975", clal this PII-3. Now, imagine that the LLM hallucinates the date of birth, and instead thinks it is "March 4th, 1975" - call this PII-4. When the AI Agent passes this information to the Authorization Server, the particular combo - {PII-1, PII-2, PII-4} does not correspond to any user in the system. The AS refuses to issue a token.

By altering the required set of PII elements, designers of AI Agents can control the likelihood of hallucination (or malicious prompt injection attacks) resulting in a token issuance for the wrong user. The selection of the information becomes even more important for AI Agents. Usage of PII elements which are known publically becomes a real risk. If these are included in the training data for the LLM, it is indeed possible (and likely) that the LLM would hallucinate it. For example, if the patient in question is a famous actor, and their date of birth is public record in the IMDB, it is likely that the LLM

would hallucinate the date of birth. Consequently, the Authorization Servers can be configured to require sufficient number and complexity of PII in order to provide the desired level of security.

3.2. Human-In-The-Loop

The AS is fully responsible for user (live agent) interaction:

1. _Initiate_ the consent review flow with the user. This spec does not specify how this occurs, it could be through an app, a chat client, or some other mechanism.
2. _Display_ the reason and each scope_descriptions entry to ensure the user has all the information necessary to assess the consent request.
3. _Authenticate_ the user (including MFA) and, upon consent, bind approval to request_code.

Note: The agent does _not_ handle redirects or UI rendering—it passively awaits token availability.

After user approval, the agent obtains its access token via HTTP POlling or SSE.

4. Detailed OAuth Grant Flow

Details to be filled in. New token endpoint which takes a series of PII parameters as input and produces an access token.

An agent requests user approval by authenticating and POSTing to /agent_authorization:

POST /agent_authorization HTTP/1.1 Host: auth.example.com Content-Type: application/x-www-form-urlencoded Authorization: Basic <base64(client_id:client_secret)>

grant_type=urn:ietf:params:oauth:grant-type:agent_authorization& scope=urn:example:resource.read urn:example:resource.write reason="<human-readable reason>"

* _grant_type_: MUST be urn:ietf:params:oauth:grant-type:agent_authorization.
* _scope_: Space-delimited list of scope URIs.
* _reason_: Concise, human-readable explanation provided by the agent; MUST be echoed verbatim by the AS.

Upon receipt, the AS:

1. _Validates_ client credentials.

2. _Fetches_ each scope's description from the target RS's https://{rs}/.well-known/aauth.json#scope_descriptions.
3. _Returns:_

{ "request_code": "GhiJkl-QRstuVwxyz", "token_endpoint": "https://auth.example.com/token" (https://auth.example.com/token"), "poll_interval": 5, "expires_in": 600, "poll_sse_endpoint": "https://auth.example.com/agent_authorization/sse" (https://auth.example.com/agent_authorization/sse"), "poll_ws_endpoint": "wss://auth.example.com/agent_authorization/ws" }

To obtain tokens for HITL:

1. _HTTP Polling_

    POST /token HTTP/1.1 Host: auth.example.com Content-Type: application/x-www-form-urlencoded Authorization: Basic <base64(client_id:client_secret)>

    grant_type=urn:ietf:params:oauth:grant-type:device_code& device_code=GhiJkl-QRstuVwxyz

Pending: {"error":"authorization_pending"} _Success_: Standard OAuth 2.x token response.

1. _Server-Sent Events (SSE)_

    GET /agent_authorization/sse?request_code=GhiJkl-QRstuVwxyz HTTP/1.1 Host: auth.example.com Authorization: Bearer <agent_jwt> Accept: text/event-stream

On approval:

event: token_response data: {"access_token":"<JWT>","expires_in":900, "issued_token_type":"urn:ietf:params:oauth:token-type:jwt"}

1. _WebSocket_

    GET /agent_authorization/ws?request_code=GhiJkl-QRstuVwxyz HTTP/1.1 Host: auth.example.com Upgrade: websocket Connection: Upgrade Sec-WebSocket-Protocol: aauth.agent-flow Authorization: Bearer <agent_jwt>

On open:

```
{
  "type": "token_response",
  "access_token": "<JWT>",
  "issued_token_type": "urn:ietf:params:oauth:token-type:jwt",
  "expires_in": 900
}
```

5. Informative References

[I-D.rosenberg-ai-protocols]
Rosenberg, J. and C. F. Jennings, "Framework, Use Cases and Requirements for AI Agent Protocols", Work in Progress, Internet-Draft, draft-rosenberg-ai-protocols-00, 5 May 2025, <https://datatracker.ietf.org/doc/html/draft-rosenberg-ai-protocols-00>.

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <https://www.rfc-editor.org/info/rfc6749>.

Authors' Addresses

Jonathan Rosenberg
Five9
Email: jdrosen@jdrosen.net

Pat White
Bitwave
Email: pat.white@traego.com