Network Working Group
Internet-Draft
Intended status: Informational
Expires: 10 May 2026

J. Rosenberg
Five9
P. White
Bitwave

6 November 2025

AI Agent to Tool (A2T) Protocol
draft-rosenberg-aiproto-a2t-00

Abstract

This document defines a protocol that facilitates integration of tools into the design and run-time operations of AI Agents. The focus is on enterprise AI agents that need to make use of APIs exposed by third party providers. This protocol, called the Agent-to-Tool (A2T) Protocol defines an OpenAI spec that has two principle features - enumeration of tools and invocation of tools. The enumeration API enables a human - the AI Agent designer employed by the enterprise - to select and include tools from third-party vendors into operating procedures (also known as skills or instructions) which direct the behavior of AI Agents, including how and when to invoke those tools. The enumeration API can also be used (optionally) at run-time for the LLM to obtain tool descriptions. The second feature of the API - invocation - allows the AI Agent executor to perform the inter-domain invocation of the tool at run time. By standardizing these two API functions, the time and cost of integration of Internet APIs into AI Agents can be reduced.

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 10 May 2026.

Table of Contents

1. Background

AI Agents are applications built using Large Language Models (LLMs). AI Agents typically operate on behalf of some human or organization, and are capable of taking actions through the invocation of functions, also known as tools. [I-D.rosenberg-ai-protocols] provides an overview of protocols and use cases for AI Agents.

In enterprise applications of AI Agents - including customer support uses cases - it is very common for the AI Agent to invoke tools provided by other organizations. As an example, consider an enterprise that is a healthcare provider. They have an AI Agent that needs to facilitate appointment scheduling, appointment cancellation, prescription refills, store location and hours lookup, and vaccine scheduling. The actions are all supported by tools which are exposed to the AI Agent through its instructions. The tools, in this case, could be implemented by servers within the enterprise domain - built by their IT department for example. Or, they might be implemented by B2B SaaS providers that offer these capabilities to the entperrise. For example, in the healthcare space, Epic, Cerner, and Meditech are a few of the B2B SaaS providers that are often used by retail healthcare providers.

It is very common for enterprises to have dozens, if not hundreds, of different systems providing APIs that might need to be accessed by an AI Agent. This high cardinality is not new. Prior to the advent of large language model technologies, voice and chat bots - and even classic "press 1 for sales, and 2 for support" Interactive Voice Response Systems (IVR) - needed to access large numbers of APIs to provide customer support for end users. Integration of these APIs into IVRs, voice bots and chat bots often represented the most expensive and complex part of deploying those systems - not the training and optimization of the AI model which invokes them.

The advent of AI Agents - if anything - only increases the likely cardinality of the number of distinct API sysems that need to be integrated. This is due to increased capabilities of LLMs compared to prior generation AI models. Whereas, in the past, certain use cases where just too difficult to automate (for example, multi-room booking for hotels - is difficult to do on websites, and almost never attempted by voice or chat bots) - now, these may be within reach for automation using AI Agent technology. As a result, we will see more use cases, and therefore even more APIs that need to be integrated.

For an AI agent to be effective, it needs more direction than just the enumeration of tools with names and descriptions. Particularly in enterprise and customer support use cases, detailed instructions are required which direct when, how, if and why those tools should be invoked. These instructions guide the AI agent on how to converse with the customer, what kind of information to collect, how to use it to invoke the tools, how to handle the differing responses, and what to do with the data that is returned. We use the term operating procedure to describe an instruction that provides this detail to an AI Agent. An operating procedure is typically specific to a particular task (for example, booking an appointment for a healthcare visit). An operating procedure at the end of the day is a document,

containing natural language text with references to tools, tool inputs, and tool outputs. Others have referred to these as skills (see the recent Anthropic blog post here: https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills (https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills)), instructions or even prompts.

Here is an abbreviated example of an operating procedure for prescription refill that illustrates this:

```
##Goal

Your goal is to help the user with prescription refill inquiries.

## Prescription Refill Status
Follow this flow if the user:
- is asking about the status of a prescription
 refill request they've made;
- is asking when their prescription refill(s)
  will be ready. For example:
- What's the status of my prescription request?
- I just want an update regarding my prescriptions.
- When will my prescription refill be ready?
### Prescription Lookup
- Use look_up_prescriptions() to look up the user's
  active prescriptions. You must have previously
  authenticated the user and have their account ID.
  If you don't have their accountID yet, perform the
  authentication flow.
### Present Status
- Find the prescriptions which have a status other
  than null and present them to the user along with
  their status.
### Sending Confirmation
- Ask the user if they want to receive a confirmation
  when the refill has been sent to the pharmacy.
- If you have the user's preferred contact method
 (which the you would have gotten earlier from
  look_up_ani_user_by_ani(), use that information
  and send them the confirmation of the appointment
  status along with the appointment details.
```

Figure 1: Example Operating Procedure

As you can see from the example, the operating procedure is a detailed recipe-book - a training manual of sorts - for the AI Agent. It describes API sequencing (for example, that the lookup of

prescriptions can only be done after authentication), usage of parameters (that accountID must be included from prior authentication), handling of results (show the list to the user), and so on.

Operating procedures are analagous to training manuals for human agents, which similarly instruct them in the sequencing and handling of a call. Instead of invoking tools, humans use web applications to do a similar job. In traditional contact centers, entire teams of people are accountable for creating and maintaining these training manuals. They are essential for ensuring quality of customer support.

At the end of the day - operating procedures are what make the AI Agent work. While their generation may be assisted by an AI to facilitate authoring, operating procedures are crafted by human beings using traditional software. The person whose role is to create the operating procedure is called the AI Agent designer - or just designer for short. The software used by the designer to create an AI Agent, including the authoring of the operating procedure, is called the AI Agent platform. To properly create an operating procedure, the designer must be a domain expert - familiar with the appropriate process for performing the task the AI Agent is to perform.

Part of task of creating an AI Agent is the selection of the tools to be used. Typically, this selection happens through the authoring of the operating procedure, which makes explicit reference to the tool as in the example above. It is also possible for the designer to select a tool for usage by the AI Agent but not provide any guidance on its usage in the operating procedure (most of the current APIs for AI Agents separate the tool list from the instruction, thus making this possible). While technically this can work, it often results in higher rates of hallucinated tool usage.

The choice by a designer on the set of tools that are shown to the LLM (we use the verb show as a synonym for inclusion of information into the model’s context window) is referred to as tool selection. The process of tool selection and authorship of the operating procedures happen ahead of the deployment and actual usage of the AI Agent. We refer to this as design time, whereas the usage of the AI Agent by end users is called run time.

At run time, a user will begin a session with an AI Agent. At the beginning of the session, the AI Agent is shown the operating procedure and tools, and begins operation.

It is becoming increasingly common for AI Agent design systems to make use of multiple distinct AI Agents (called sub-agents) which are each equipped to handle specific tasks. These are called multi-agent systems. Once again using our health care example, one sub-agent might handle prescription refills, while another handles appointment scheduling and cancellation. Each sub-agent would have its own distinct set of operating procedures and tools, appropriate for those tasks. This further reduces the size of the context window at any point in time, reducing hallucination rates. There are various techniques for how these sub-agents are strung together. At the end of the day, it is often done through tool calling, asking the LLM to invoke a tool that transfers the session to a different AI Agent. This is (interestingly) analagous to live human agents in contact centers, which typically have human agents with differing skills. When a customer asks for something which is outside of the expertise for the human agent they are talking to, the human agent transfers the call to a different human agent with the right skill. In essence - that is what is happening in these multi-agent systems.

The usage of a designer to select only those tools needed for an agent, and the usage of multi-agent systems, provides progressive tool disclosure. Progressive tool disclosure is a concept wherein an AI Agent is only shown just the set of tools it needs at a given time, and as the conversation flows, more tools are shown.

The tools selected by the designer, and incorporated into the operating procedure, are often implemented by the executor through the invocation of APIs on some server. The server exposing those APIs is called an API Server.

2. Terminology

Terminology is becoming foundationally important in this emerging AI Agent field. This section summarizes the key terminology and concepts from above.

* AI Agent: A software application that makes use of an LLM to perform tasks that require interaction with software systems. With an AI Agent, the LLM can output both text for the user, but it can also request the invocation of tools, which are an interface the LLM can use to access those software systems. AI Agents are composed of two elements: an LLM and an executor. The executor is built using traditional software (e.g., Java, node.js, python, etc) and interfaces with the LLM.

* Tool: An interface representing a software system. A tool has a name, a description, inputs and outputs. In AI Agent system, the LLM is responsible for collecting the inputs and requesting the

invocation of the tool. The executor actually executes the tool, often through the invocation of APIs, and feeds the results back to the LLM. By convention, tool names use underscores and have a terminating curly bracket, even though they may also take parameters. For example, when we say "the LLM invokes schedule_appointment()" this means that the LLM has made a decision to invoke the tool name schedule_appointment and has synthesized the needed input parameters to the tool

* Executor: Also called the orchestrator or orchestration engine. It represents the traditional software component of an AI Agent. The executor funnels messages to and from the end user into the LLM. It receives the outputs of the LLM, and if that output is a request to perform a tool call, the executor runs the tool and provides its output back to the LLM.

* Show: The process wherein the executor provides input to the LLM, inclusive of tool call results, by adding it to the context window of the LLM. For example, we say, "the tool call result is shown to the LLM'.

* Operating Procedure: A document authored by a human, and shown to an LLM during run-time operation, which instructs the AI Agent on the sequence of tools to invoke, how to interact with the user to collect the data needed for tool invocation, what to do with the results of the tool call, how to handle different conditions based on the output of the tools, and how to handle exceptional conditions. Operating procedures reference tools and their inputs by name.

* AI Agent Designer: Synonymous with designer

* Designer: A human being whose job is to create operating procedures and perform tool selection during the design phase of the AI Agent.

* Tool Selection: The process of selection of the right set of tools to show to an AI Agent for a particular task. By selecting only those tools which are needed for a task, the likelihood of tool hallucination is reduced. Tool selection in the A2T framework happens at design time (though the protocol allows this to happen at run time in an interoperable fashion, should that mode be selected by the AI Agent platform vendor).

* Design Time: A point in time during which the designer is interacting with the AI Agent platform in the producion of an operating procedure and selection of tools.

* Enterprise: The administrative entity on whose behalf an AI Agent operates. Typically the designer is an employee of the enterprise, though this can be outsourced to consulting firms or the AI Agent Platform vendor.

* Run Time: A point in time during which an end user is interacting with an AI Agent.

* Multi-Agent Systems: An AI Agent wherein its job is broken down into a smaller set of discrete tasks, each handled by a sub-agent that is equipped (via its operating procedure and tool list) to perform that specific task

* Sub-Agent: An AI Agent that performs a narrow set of tasks. Multiple Sub-Agents are woven together to form the AI Agent in a Multi-Agent System.

* Progressive Tool Disclosure: A technique for reduction of hallucination wherein, as an AI Agent session unfolds, the set of tools shown to the LLM evolves, offering it only the tools it needs to do its job at any point in time.

* API Server: A server that exposes APIs which can be invoked via a tool call by the AI Agent

* API Provider: The administrative entity that is offering APIs that are of interest to an AI Agent. Typically, the API Provider is different from the enterprise, and is also different in turn from the AI Agent Platorm vendor.

* AI Agent Session: A period of time from when an AI agent begins - by having the executor show the LLM the initial operating procedure and tool selection - estabishing initial context - and interacting with the end user - until the end user interaction completes.

* AI Agent Platform: A piece of software which includes a design time experience for designers, and a run-time capability for handling AI Agent sessions. Typically built as a cloud service by the AI Agent Platform vendor. Though the AI Agent Platform can be provided by the vendor of the LLM itself, these can be distinct vendors.

* AI Agent Platgorm Vendor: The administrative entity that has developed the AI Agent Platform, and made it available for usage by enterprises to author and execute AI Agents.

3. Problem Statement

There are multiple problems that are addressed by A2T, covering design time and run-time challenges.

3.1. Tool Curation

At design time, the AI Agent platform needs to enable the designer to author the operating procedure and, as a result, select the tools that will be made available to the AI Agent (or the sub-agent). To do this, the designer needs to know what the available tools are, from which they can select. In essence, the designer is performing a curation task, choosing amongst a large inventory of tools available from many possible API servers. Most often, the enterprise deploying the AI Agent, is different from the vendor of the AI Agent platform, which is different still from the vendors of the API servers.

As a result, the designer needs to know the API surface area of the various vendors of API servers, and know which API to select and use. Sometimes, API Servers offer OpenAPI specifications for their APIs, but these are tailored for usage with software developers, not consumption by AI Agent designers, and certainly not by LLMs. Consequently, they lack the required natural language descriptions to aid in the curation process by the designer. Designers need to spend time researching and studying APIs, learning what they do, and determining whether the right APIs exist, and if so, what they are. They have to determine whether the API can be used as is, or whether changes are needed to make the API more easily (and reliably) consumeable for the automation. This process is historically long and tedious, and is exacerbated by the fact that the APIs are spread across dozens of API servers from the many vendors used by a particular enterprise.

The lengthy time required for API integration is one of the most significant parts of the time required to build automation solutions (including AI Agents) within enterprises.

3.1.1. Tool Sculpting

Oftentimes existing APIs on API servers - usually REST these days - are complex and (clearly) not optimized for usage by an LLM. They have been designed to be consumed by traditional software authored by software developers. Traditional REST API signatures have complex inputs - URI parameters, header parameters, and JSON bodies. They also tend to mix programmatic elements and semantic elements. Programmatic elements are those parts of the API parameters that are meaningful only to the software which is invoking those APIs. These include resource identifiers like UUIDs, trace IDs, URLs, timestamps,

meta-data and so on. Semantic elements are those which are ultimately the input from the end user which drove the execution of the API. As an example, when doing something as simple as looking up the weather, the city and the date (today or the future) are the semantic elements. But the lookup API might need to provide programmatic elements like a weather station ID, which is not meaningful to the user request.

It is also the case that a particular end user request (e.g., give me the weather tomorrow in Manhattan), doesn't map to a single API call. Instead, a sequence of API calls need to be orhcestrated together to actually provide this simple ask. As an example, the APIs at the US National Weather service provide weather forecasting based on grids of 2.5km to 2.5km. Consequently, to provide the weather for Manhattan, the location must be first mapped to a coordinate, and then the coordinates fed into the forecast API.

There are two ways these problems are solved today.

A simple way to solve these problems is to just let the LLM figure it out. A naive implementation would - in essence, use the existing OpenAPI specs for an API as a tool, and request the LLM to synthesize inputs, process outputs, and chain API calls together. While this is possible - and indeed works for simple use cases - it introduces hallucination risk. It also complicates the design process, requiring the designer to understand the details of these APIs. The designer must - via the operating procedure - direct the AI Agent on how to use the APIs and navigate them.

This approach has many problems. First, it increases the time required to author the operating procedure, since it must now include the manually generated instructions on how to use the APIs. Second - and most importantly - it increases hallucination risk. The larger the gap between how the API works, and what the semantically meaningful operation is - the larger the surface area for mistakes by the LLM.

The lesson is - never ask an LLM to do what a normal piece of software can do better.

And thus, the second way this is solved today is through the development of automation ontop of existing APIs. Again using the US National weather service example, a piece of traditional software can be written which exposes an API called lookup_weather_by_city. This API would, under the hood, first convert the city to geo-coordinates, and then lookup the weather for those coordinates. The resulting simplified API can then be shown to the LLM as a tool.

The process of adding a layer of traditional software ontop of existing APIs in order to produce ones that more directly map to user input, in order to reduce tool call hallucination risk, we call as tool sculpting.

Specifically, tool sculpting is the process of crafting an API which: 1. Minimizes the inputs for each tool to only those which are semantically meaningful and must be synthesized by an AI Agent 2. Minimizes the outputs for each tool to only those which would be relevant to an AI Agent 3. Orchestrates multi-step backend API calls so that the AI Agent doesn’t need to do it

4. The A2T Protocol Goals

The A2T protocol (A2T) addresses the underlying problems of tool curation and tool minimization.

It has, at its core, the following goals:

* Reduce the time and costs for designers to create operating procedures and perform tool selection by simplifying the process of curation

* Enable development of AI Agents for enterprises which need access to tools across dozens or hundreds of API servers, typically sold to the enterprise by third party B2B SaaS vendors, by consolidating decision logic on tool selection in the hands of the designer

* Simplify adoption by tool vendors by building ontop of the existing API platforms and APIs they already have. The closer the current API surface area is to the ideal tool set for an AI Agent, the less effort is required to implement

* Standardize an API signature for tool calling

* Tools are just APIs that are optimized for an LLM, but can also be used by traditional software applications too

* Authentication and Auhtorization use the existing techniques used for REST APIs offered by the API Vendor

* Reduce hallucination risk through tool sculpting

* Separation of responsibility - API vendors perform tool sculpting, and the AI Agent designer crafts the operating procedure which instructs the AI Agent on how to use the many tools (across vendors even) that it can use. In other words - one vendor makes tools which independent of the agent, and the other vendor makes agents which make use of those tools.

5. Overview of Operation

A2T is a web API that defines a pair of REST endpoints - one for enumerating tools available on the server, and the other for invoking a tool on the server. Both of these make use of a tool signature, which is a standardized interface for what a tool looks like - how to describe a tool, and how to invoke it. The API client is implemented by the AI Agent platform, and the rest endpoints are implemented by vendor wishing to expose their APIs as A2T tools. It is envisioned that these just end up being new REST APIs added to the existing API surface area of the API Provider.

5.1. Tool Signatures

A2T standardizes the signature for tool calling. A tool can be thought of as a Java interface, and the implementation of the tool happens over the wire by having the executor take the tool call from the LLM and send it to the API server.

Every tool in A2T is comprised of the following descriptive signature:

* The tool ID
* The name of the tool
* The description of the tool
* The set of input parameters. For each input parameter:
  - The name of the input parameter, and whether it is required or optional
  - The type of the input parameter - one of a small set (string, int, enum)
  - The description of the input parameter
  - Any constraints on the input types
    - Strings: max length
    - Int: min and max values
    - Enum: list of enumerated values that are permitted, along with a description of each value.
* The output, of which there can be one or more. Each output is:
  - The name of the output
  - A description of the output
  - The type of the output (string, int, enum, json)

Invoking a tool requires the following to be provided:

* the name of the tool
* an input parameter list, which is an array of name/value pairs

Consequently, the A2T protocol defines two JSON objects - the signature, and the invocation. The A2T protocol carries these in its body.

The signature ensures that the executor can take the output of the LLM, validate it against the signature, and execute it by sending the invocation over the wire. For example, if the LLM synthesizes a tool call request with a missing parameter, or with an enum value that is not amongst the allowed values, the executor can reject it and ask the LLM to try again, or take other error handling logic.

5.2. Enumeration Endpoint

The enumeration endpoint - as the name implies - returns a list of tools available. This API endpoint returns an array, each element in the array is a signature as noted above. The API supports tagging and versioning, with filtering on those parameters, to facilitate design time interactions between the AI Agent platform and the API server.

This API facilitates the following design-time workflow:

1. The designer identifies the set of API vendors for which the AI Agent should be allowed to operate
2. For each such API vendor, the designer configures the AI Agent platform with the URI for the A2T tool enumeration endpoint. This specification suggests a common naming practice of using the root API endpoint followed by "tools" (e.g., https://api.weather.gov/tools (https://api.weather.gov/tools)) to simplify this when possible.
3. The administrator configures the appropriate credentials needed for AuthN and AUthZ (typically an OAuth grant flow) - this is not any different from any other endpoint from the API provider
4. The designer accesses the AI Agent platform, and the platform invokes the enumeration API and retrieves the tool signatures for all of the tools exposed on the API server for all of the configured API providers. This provides a list of tools to the platform. This list can be viewed, searched, filtering and sorted by the designer. The UI of the design time experience can help the designer find and select tools for placement into the operating procedure.

The above steps can be done just once, and then apply to all AI Agents subsequently built on the platform. Alternatively, the enumeration APIs can be explored by the AI Agent platform at design time, as the designer works.

Once the designer has selected the tool, the tool signature can be retrieved by the AI Agent platform, stored, and then provided to the LLM at run-time. With A2T, it is also possible to use the enumeration API retrieve a fresh tool signature at run-time, rather than use the version retrieved and cached at design time. There are pros and cons to both approaches. Retrieving and caching the signature at design time ensures consistency. Designers seeking greater consistency - wherein testing of the agent performed before launch, matches run-time behavior, will prefer to cache the descriptions. Indeed, the usage of caching allows the designer to tune and tweak the natural language text in the signature to improve accuracy, without dependng on the API vendor to do so. This facilitates the separation of concerns built into the A2T protocol - that the AI agent designer can control all aspects of the behavior of the AI Agent, including the authorship of all natural language provided to the AI Agent. The API vendor provides the tool - which is purely programmatic.

Alternatively, if the AI Agent platform fetches the tool signature each time at run time, the latest-and-greatest can be used. A2T does not technically prohibit this, allowing this mode to be implemented in an interoperable fashion.

The key to efficacy of tool usage is the tool sculpting, a responsibility which sits with the API vendor in the A2T protocol. The API server must distill down its APIs into a set of tools, each of which do targeted, focused tasks. Tool sculpting has a double benefit. First, it reduces the surface area of information that the LLM needs to synthesize to invoke the tool, and reduces the amount it needs to comprehend to process its outputs. Second, it simplifies the job of the designer in selection of the tool, and in crafting the operating procedure to instruct the LLM on when and how to invoke it.

A2T also provides an API, part of the enumeration endpoint, for retrieving the signature for a specific tool. This is useful for the run-time fetching of the signature.

5.3. Invocation Endpoint

The invocation endpoint is used by the executor at run-time.

When the LLM creates a tool request, it provides the tool name and input values. Using the signature, the executor can validate the tool call request as being valid for the tool, and then use the wire protocol to invoke the tool on the API server. When the response comes, it can then be shown to the LLM.

The A2T invocation endpoint allows versioning, so that a specific version of the tool can be used.

5.4. Versioning

A common problem historically in the development of voice and chat bots - the precursor to modern AI Agents - is that API vendors sometimes change APIs, breaking operation of the bot. This problem remains with AI Agents, and A2T addresses it by natively adding version support.

The A2T protocol requires versioning for tools. It mandates that each version of a tool is backwards compatible. A new version of a tool can only add new optional input parameters, or add new output parameters. To ensure consistent behavior, A2T requires API servers to allow invocation of a tool against a specific version, allowing AI Agent designers to select when to upgrade when new versions become available.

6. Protocol Details

The following provides a detailed specification of the protocol.

6.1. Enumeration API

A2T servers MUST implement the enumeration API, which is structured as follows:

GET {platform-root}/tools

It is RECOMMENDED that {platform-root} be the same as the API root endpoint that the API vendor already provides.

The body returned in the result, which supports pagination (described below), includes an element called items, a JSON array that contains the array of signatures. Each element of the array is a JSON object of type ToolSignature. An example of a valid JSON object of this type is:

```
{
  "toolId" : "0479a45d-ad0a-49d4-94db-75edf00d2ca4",
  "name" : "Lookup Weather",
  "description" : "Invoke this tool to lookup the
                   weather for a given city.",
  "img" : "{optional URL for the tool}",
  "version" : "{versionNum},
  "currentVersion": "{versionNum}",
  "tags" : ["system", "retrievals"]
  "input_parameters" :
  [
    {
    "id" : "city"
    "name" : "City",
    "type" : "string",
    "description" : "The city for the weather
       lookup. For example, Boston or Los Angeles.",
    "required" : true
    }
  ]
  "output_parameters" :
  [
    {
      "id" : "temp-fh"
      "name" : "Temperature in Fahrenheit",
      "type" : "int",
      "description" : "The current temperature
         in the named city."
    }
  ]
}
```

Figure 2: ToolSignature Object Example

The toolID is always a UUID, ensuring tool uniqueness on the server. The name MUST be less than 255 characters, and MUST be unique across all other tools on the API server. Names SHOULD use snake case, and be sufficiently long to be usefully descriptive without the description. The longer names also reduce the likelhihood of tool name collisisons across API servers, in cases where an AI Agent is being shown tools from different API servers. However, it is ultimately the responsibility of the AI Agent platform to make sure tool names shown to the LLM are unique.

An example of a good tool name is:

lookup_weather_by_city

The tool description MUST be less than 2000 characters. Tool names and description MUST be in English. Names and descriptions are consumed by both the designer (a human) and the LLM. Localization is only needed for the human, and is the responsibility of the AI Agent platform.

For a particular version of the tool, the signature is completely locked. Meaning, the API server MUST NOT change any part of the signature (with the notable exception of the value of currentVersion) without changing the version. The version number of a tool MUST be a positive integer, and MUST increase monotonically, starting at 1. The value of currentVersion reflects the most current version of the tool.

The version returned at the top level tool enumeration endpoint MUST be the latest version of the tool, in which case the values of version and currentVersion are always identical.

Input parameters are typed. The type value is optional to include in the JSON. If omitted, it is "string" by default. The other valid values are "int", "boolean" and "enum". More complex input types are not supported. This is to constrain the complexity of objects that the LLM is asked to synthesize. [OPEN ISSUE: lists maybe??]. If the type is "int", the optional "max" parameter can be included, which represents the maximum value for the integer. The default is 65535. If the type is enum, the input parameter must include the attribute "allowed-values", which contains an array of name-description pairs. As an example:

```
{
"id" : "flight_class"
"name" : "Flight Class",
"type" : "enum",
"description" : "The cabin class for the flight reservation",
"allowed-values" : {
    [
        {
            "name" : "ECONOMY",
            "description" : "Economy class, the least
            expensive cabin class. Also known as coach."
        },
        {
            "name" : "PREMIUM_ECONOMY",
            "description" : "Premium economy class, the
            second seat tier. More legroom."
        },
        {
            "name" : "BUSINESS",
            "description" : "Business class, the next to top
            seat tier. Offers lie-down seating."
        },
        {
            "name" : "FIRST",
            "description" : "The top tier. Lie down seating,
            luxury meal service, lounge access."
        },

    ]
}
"required" : true
}
```

Figure 3: ToolSignature Input Parameters Enum Support

For enumerated strings, names MUST be snake case but capitalized, with a maximum length of 255 characters. Descriptions have a maximum length of 2000 characters.

Input parameters have IDs and names. The IDs are meant to provide uniqueness across version, facilitating design time software processing of version changes. The IDs need only be unique within the tool.

Input parameter names must also be unique within the tool. The LLM is asked to synthesize the name, not the ID. Consequently, there is no reason to show the parameter ID to the LLM at run time.

Output parameters include the additional type of JSON. This is used for cases where a more complex structure is returned, and the expectation is that the LLM can parse the structure for comprehension. There is an explicit assumption in the signatures that it is more important to reduce hallucination on input synthesis, than it is to risk mis-comprehension of tool output. This is why the protocol is asymmetric - allowing JSON output, but not input.

For input parameters, the "required" field is optional; if absent, the default is "true". Optional inputs should be used sparingly. It is better to have more tools with fewer inputs, than a single tool with many optional inputs. This reduces the likelihood of the LLM hallucinating tool usage.

6.2. Details for a Specific tool

The API allows for a signature to be retrieved for a specific tool via:

GET {platform-root}/tools/{toolID}

The server MUST return the current version of the specific tool.

6.3. Tool Version Retrieval

The API allows all versions of a tool to be retrieved via:

GET {platform-root}/tools/{toolID}/versions

The result is paginated, and includes the full signature for each version. These MUST be sorted from most recent to oldest.

An individual version can then be retrieved via:

GET {platform-root}/tools/{toolID}/versions/{versionNum}

6.4. Pagination

Pagination is supported on the enumeration APIs (any of the above endpoints). All of the APIs support a pageCursor and pageLimit URI parameter. The page limit specifies the maximum number of pages to return.

The array holding the enumeration is included in the JSON body using an attribute called items which contains the array. A peer element, called paging, holds the result of the query. It contains two values - pageLimit - containing the actual page limit the server is offering, and next. The next parameter contains the value of the cursor query parameter that the client should provide as a URI parameter to retrieve the next page.

7. Invocation API

At run-time, a tool is invoked with the invocation API:

POST {platform-root}/tools/{toolID}:invoke

The body of this takes the invocation object, an example of which is:

```
{
  "name" : "lookup_weather_by_city",
  "input_parameters" :
  [
    {
    "name" : "City",
    "value" : "Omaha, Nebraska"
    }
  ]
```

The response should contain the result of the invocation, an example of which looks like this:

```
{
  "output_parameters" :
  [
    {
    "name" : "Temperature in Fahrenheit",
    "value" : 80
    }
  ]
```

The output MUST include the name which matches to the output name in the signature.

It is also possible to request invocation of a specific version of the tool:

POST {platform-root}/tools/{toolID}/versions/{versionId}:invoke

Traditional HTTP response codes apply to the response, including 5xx for temporary failures and 4xx if the request is formatted correctly. Note that, a 4xx should never happen if the client follows the specification defined here.

If the request generates a 5xx, the executor SHOULD retry, rather than show the error to the LLM. However, this is up to the discretion of the AI Agent designer on how long to wait before showing it the error and asking it what to do.

7.1. OpenAPI Specification

To be added when spec is closer to complete.

8. Relationship to MCP

A2T is similar in many ways to the Model Context Protocol (MCP) (https://modelcontextprotocol.io/introduction (https://modelcontextprotocol.io/introduction)) which is being driven by Anthropic, but different in others. This section covers both similarities and differences. On the whole, the two protocols tackle a similar problem (facilitating tool calling and serving tool context to LLMs), but differ in their approach (MCP better suited for use cases where less designer control is appropriate, and A2T for cases where more designer control is appropriate).

8.1. Similarities

Both MCP and A2T support APIs for enumerating tools and invoking tools.

Both MCP and A2T can be used to feed context to the LLM at run-time.

8.2. Differences

The biggest difference is the assumption in control. A2T is very much targeted at use cases where there is a designer persona crafting an operating procedure. Consequently, the tool enumeration API is consumed by traditional software in A2T, enabling the designer to perform the process of tool selection and design of the operating procedure, both at design time. In MCP, the tool enumeration API is primarily consumed by the AI Agent at run-time. For this reason, MCP has the idea of progressive tool disclosure that happens on the server side, in order to reduce model context. A2T, on the other hand, assumes progressive tool disclosure happens as a consequnce of a design time operation, and thus, in simple terms, is client side.

In MCP, the logic for progressive tool disclosure is distributed in cases where there are multiple MCP servers, with each server executing its own, non-standardized logic for progressive tool disclosure (though the protocol does not require this behavior, it is purportedly one of the main reasons for the session oriented aspect of the protocol). In A2T, the logic for progressive tool disclosure lives in the AI Agent, with the logic being crafted at design-time by the designer. In this way, A2T is more compatible with Anthropic's skills concept, than it is with its MCP protocol. Indeed, A2T could be viewed as the offspring of skills and MCP, by adding over-the-wire tool calling to skills.

These foundational differences have consequences in the other parts of the protocols.

MCP is session oriented. A session oriented protocol is required when the server is responsible for progressive tool disclosure. A2T is stateless, because it is not performing server side progressive tool disclosure.

MCP uses JSON-RPC and runs over either stdio (for local usage on a PC) or streamable HTTP for network connections. A2T is a normal REST API and uses traditional HTTPS and OpenAPI specifications.

MCP typically requires a dedicated server that provides its functionality (termination of the persistent connection, session handling, and gateway to existing APIs). A2T is designed to be added to existing API servers as just another API - albeit ones optimized for LLMs.

MCP APIs are not easily consumeable by non-LLM applications. A2T APIs are just APIs and can be consumed by both traditional software and AI Agents.

9. 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>.

Authors' Addresses

Jonathan Rosenberg
Five9
Email: jdrosen@jdrosen.net

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