

Network Working Group
Internet-Draft
Intended status: Informational
Expires: 16 August 2026

Y. Cui
Tsinghua University
Y. Chao
C. Du
Zhongguancun Laboratory
12 February 2026

AI Agent Discovery and Invocation Protocol
draft-cui-ai-agent-discovery-invocation-01

Abstract

This document proposes a standardized protocol for discovery and invocation of AI agents. It defines a common metadata format for describing AI agents (including capabilities, I/O specifications, supported languages, tags, authentication methods, etc.), a capability-based discovery mechanism, and a unified RESTful invocation interface.

This revision additionally specifies an optional extension that enables intent-based agent selection prior to discovery and invocation, without changing existing discovery or invocation semantics.

The goal is to enable cross-platform interoperability among AI agents by providing a discover-and-match mechanism and a unified invocation entry point. Security considerations, including authentication and trust measures, are also discussed. This specification aims to facilitate the formation of multi-agent systems by making it easy to find the right agent for a task and invoke it in a consistent manner across different vendors and platforms.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at <https://example.com/LATEST>. Status information for this document may be found at <https://datatracker.ietf.org/doc/draft-cui-ai-agent-discovery-invocation/>.

Discussion of this document takes place on the WG Working Group mailing list (<mailto:WG@example.com>), which is archived at <https://example.com/WG>.

Source for this draft and an issue tracker can be found at <https://github.com/USER/REPO>.

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 16 August 2026.

Copyright Notice

Copyright (c) 2026 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	3
2. Conventions and Definitions	4
3. Agent Metadata Specification	5
3.1. Core Fields	5
3.2. Operations and I/O Schema	7
3.3. Example Agent Metadata	8
4. Agent Discovery Mechanism	8
4.1. Registry Overview	8
4.2. Agent Registration	9
4.3. Querying Agents	9
4.3.1. Attribute-Based Query (Filter)	10
4.3.2. Semantic Query (Natural Language Search)	10
4.3.3. Retrieve Single Agent	11
5. Agent Invocation	11
5.1. Invocation Request	11

5.2. Invocation Response	12
5.3. Additional Considerations for Invocation	14
6. Agent Semantic Resolution	15
6.1. Non-Goals	15
7. Semantic Routing Platform	15
8. Backward Compatibility	16
9. Security Considerations	16
10. Example Interaction Flow	16
11. IANA Considerations	17
12. Normative References	17
Acknowledgments	17
Authors' Addresses	17

1. Introduction

As artificial intelligence technologies advance rapidly, AI agents—autonomous software components capable of perceiving their environment, reasoning, and taking actions to achieve goals—have emerged as a powerful paradigm for task execution. Today, many organizations develop specialized AI agents for various purposes: from text translation and summarization, to code generation, to data analysis and beyond. These agents are often offered as services, accessible over the network and may be integrated into larger systems. However, despite the proliferation of AI agents, there is currently no standard protocol for discovering available agents and invoking their capabilities in a uniform way.

Existing agent frameworks and platforms facilitate building agents but typically operate in isolated ecosystems, making cross-platform or cross-organization agent interoperability difficult. Each platform tends to define its own APIs for agent description and invocation, which means a client wishing to use agents from multiple sources must adapt to disparate interfaces. This lack of standardization creates friction, increases integration costs, and hampers the development of multi-agent collaborative systems.

This document addresses these issues by proposing a standardized AI Agent Discovery and Invocation Protocol. The protocol provides:

1. ***Agent Metadata Specification:** A structured JSON Schema for describing an agent's identity, capabilities, inputs, outputs, authentication requirements, and other attributes. This enables agents to publish their specifications in a machine-readable form.

2. ***Discovery Mechanism:** A registry-based approach where agents register themselves and clients can search for agents by capability, tags, or semantic queries. The registry is language and platform agnostic, facilitating cross-platform discovery.
3. ***Invocation Interface:** A RESTful API that enables a client (which could be a human user application, another agent, or an orchestration system) to invoke an agent's capabilities through a standard endpoint and JSON payloads.
4. ***Security Considerations:** Guidelines for authentication, authorization, encrypted transport (TLS), and trust establishment, ensuring that discovery and invocation happen securely.
5. ***Interoperability with Existing Standards:** This specification references existing standards such as JSON Schema, OAuth 2.0 [RFC6749], and OpenAPI concepts, and leverages established web technologies for broad compatibility.

The primary audience for this specification includes developers of AI agent platforms, providers of AI agent services, and system architects building AI-enabled applications or multi-agent systems. By adopting this protocol, an AI agent developer can make their agent accessible to a wide ecosystem, and a client application can integrate AI agents from multiple vendors without custom integration for each.

This revision extends the base protocol with an optional Agent Semantic Resolution layer that enables intent-based agent selection. This extension allows a Host Agent or coordinator to describe a task intent and receive candidate agents without predetermining which agent to invoke. ASR does not replace discovery; it adds a semantic matching phase that can precede or augment the capability-based search defined in earlier sections.

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

- * ***AI Agent:** An autonomous software component that can perform tasks using artificial intelligence capabilities. Agents may wrap language models, specialized ML models, or reasoning engines, exposing their abilities via defined interfaces.

- * ***Agent Metadata:** A structured description of an agent, including name, description, capabilities, input/output schemas, authentication requirements, and endpoint information.
- * ***Agent Registry (Discovery Service):** A service that maintains a directory of registered agents and supports queries for discovering agents by attributes or semantic search.
- * ***Gateway:** (Optional) An intermediary service that routes client requests to appropriate agents. In some deployments, the registry or another service acts as a gateway to simplify client-to-agent connections.
- * ***Invocation Endpoint:** The URL provided by an agent (or gateway) where clients send requests to invoke the agent's capabilities.
- * ***Capability:** A high-level function an agent can perform, identified by a string (e.g., "translation", "summarization", "image_classification").
- * ***Operation:** A specific action supported by an agent. Some agents may have multiple operations; for example, an agent offering both translation and language detection could list these as separate operations, each with its own input/output schema.

3. Agent Metadata Specification

The Agent Metadata Specification defines a standard JSON document that describes an agent. All agents that wish to be discoverable and invocable through this protocol **MUST** provide a metadata document conforming to the schema below. This metadata is used for agent registration and returned to clients during discovery.

3.1. Core Fields

The following are the core fields of an agent metadata document:

- * ***id (string):** A globally unique identifier for the agent. This could be a UUID or a similarly unique value, assigned by the registry upon registration or by the agent provider in advance. This ID is used to refer to the agent in all subsequent operations (e.g., retrieval, invocation routing).
- * ***name (string):** A human-readable name for the agent (e.g., "Chinese-English Translator Agent"). Names need not be unique but should be descriptive.

- * `*description (string):*` A detailed description of the agent, its purpose, and capabilities in natural language. This helps both human users and semantic search algorithms understand what the agent does.
- * `*version (string):*` The version of the agent or its metadata (e.g., "1.0.0"). This allows tracking of agent updates over time.
- * `*publisher (string):*` The name or identifier of the entity publishing the agent (e.g., an organization name or developer name).
- * `*capabilities (array of strings):*` A list of capabilities the agent supports. Capabilities are high-level descriptors (like tags or categories) that clients can filter by. Examples: ["translation", "summarization", "text_generation"].
- * `*tags (array of strings):*` Additional tags for search and categorization (e.g., ["nlp", "chinese", "transformer_model", "cloud"]). Tags differ from capabilities in that they can include broader or orthogonal categories (like domain, language support, deployment model, etc.).
- * `*endpoint (string):*` The URL of the agent's invocation endpoint. If the agent is behind a gateway, this could be either the direct endpoint or, if direct access is not allowed, a gateway path (e.g., the gateway might provide a unified endpoint like /agents/{id}/invoke and internally route to the actual agent endpoint).
- * `*supported_languages (array of strings, optional):*` A list of languages the agent supports (e.g., ["en", "zh", "fr"]). For agents dealing with natural language tasks, this field indicates which languages are handled. If omitted, the agent is either language-agnostic or should not be filtered by language.
- * `*authentication (object, optional):*` Describes the authentication mechanism required to invoke the agent. This object may include:
 - `*type (string):*` e.g., "api_key", "oauth2_bearer", "mtls" (mutual TLS), "none".
 - `*instructions (string):*` Human-readable note or URL for obtaining credentials.
 - `*scopes (array of strings):*` If OAuth 2.0 is used, the OAuth scopes required.

If no authentication is required, this field can be omitted or set with type: "none".

- * `*status (string, optional):` Operational status of the agent (e.g., "active", "inactive", "deprecated"). The registry may use this to filter out agents that are not currently available.
- * `*additional fields:` Additional fields may include metadata about rate limits (e.g., max calls per minute), pricing info (if the agent charges per use), or links to documentation. These are not standardized here but can be included in agent metadata as needed.

3.2. Operations and I/O Schema

Each agent **MUST** describe its input and output formats. This is done using the `*operations` field:

- * `*operations (array of objects):` A list of operations the agent supports. Each operation object has:
 - `*name (string):` The operation name/identifier (e.g., "translateText", "summarize").
 - `*description (string):` A description of what the operation does.
 - `*inputs (object):` A JSON Schema describing the expected input. This allows clients to understand what data to send. The JSON Schema can specify required fields, types, enums, etc.
 - `*outputs (object):` A JSON Schema describing the output format.
 - `*examples (array of objects, optional):` Example input/output pairs. Each example is an object {"input": {...}, "output": {...}} showing a sample invocation.

If an agent has a single operation, this array will have one element. If it can do multiple distinct tasks, each is listed here. Some agents may not have structured operations (e.g., a general-purpose language model that just produces text). In that case, the `operations` field might include a generic operation like {"name": "generate", ...}.

If an agent has a single operation, this array will have one element. If it can do multiple distinct tasks, each is listed here. For simple agents with one primary function, an alternative is to use top-level inputs and outputs fields directly (instead of an operations array). In that case, the whole agent effectively has one implied operation. This spec allows both styles, but using operations is recommended for future extensibility.

3.3. Example Agent Metadata

Below is an example metadata JSON for a translation agent:

```
json { "id": "agent-12345", "name": "Chinese-English Translator",
  "description": "Translates text between Chinese and English with high
  accuracy using a fine-tuned model.", "version": "1.2.0", "publisher":
  "ExampleAI Inc.", "capabilities": ["translation"], "tags": ["nlp",
  "chinese", "english", "cloud"], "endpoint":
  "https://api.example.com/agents/translate", "supported_languages":
  ["en", "zh"], "authentication": { "type": "api_key", "instructions":
  "Include 'X-API-Key' header with your API key." }, "status":
  "active", "operations": [ { "name": "translateText", "description":
  "Translates text from source language to target language.", "inputs":
  { "type": "object", "properties": { "text": { "type": "string" },
  "source_language": { "type": "string", "enum": ["en", "zh"] },
  "target_language": { "type": "string", "enum": ["en", "zh"] } },
  "required": ["text", "source_language", "target_language"] },
  "outputs": { "type": "object", "properties": { "translated_text":
  { "type": "string" } } }, "examples": [ { "input": { "text": "好世界",
  "source_language": "zh", "target_language": "en" }, "output":
  { "translated_text": "Hello World" } } ] } ] }
```

This metadata tells us the agent is an active translation agent for Chinese and English, requires an API key for authentication, and has one operation `translateText` with a clear input/output schema.

4. Agent Discovery Mechanism

The discovery mechanism allows clients to find agents that meet certain criteria. Discovery is provided by an Agent Registry (or Discovery Service) that aggregates metadata from multiple agents.

4.1. Registry Overview

The Agent Registry is a network-accessible service that:

1. Allows agents (or their administrators) to register metadata about the agent.

2. Stores and indexes these metadata entries for efficient search.
3. Provides endpoints for clients to query and retrieve agent information.

A registry may be operated by an organization for its internal agents, or by a third party acting as a directory of agents across multiple providers. Multiple registries can coexist; interoperability between registries is facilitated by consistent metadata formats, though formal registry federation is out of scope for this draft.

4.2. Agent Registration

An agent (or its administrator) registers with the registry by sending its metadata to a registration endpoint:

- * ***Endpoint:** POST /agents
- * ***Request Body:** The agent metadata JSON document.
- * ***Response:** On success, the registry returns ***201 Created*** (if a new agent was added) or ***200 OK*** (if an existing agent was updated), with the stored agent metadata (including the assigned id if the agent did not provide one). If validation fails (e.g., missing required fields), the registry returns a ***400 Bad Request*** with error details.

The registry **MUST** validate the metadata against the schema. Registration may require authentication (for example, the registry only allows verified publishers to register agents).

Updates to an agent's metadata (e.g., a new version, changed endpoint, etc.) can be done via a PUT request to the agent's entry:

- * ***Endpoint:** PUT /agents/{id}
- * ***Request Body:** Updated metadata.
- * ***Response:** ***200 OK*** on success; ***404 Not Found*** if no agent with that ID exists; ***403 Forbidden*** if the requester is not authorized to update that agent.

4.3. Querying Agents

Clients query the registry using the search endpoint. The protocol supports two types of queries:

4.3.1. Attribute-Based Query (Filter)

Clients can specify criteria to filter agents by capabilities, tags, supported languages, etc.

* *Endpoint:* GET /agents?capabilities=X&tags=Y&language=Z

* or a structured query via *POST /agents/search* with a JSON body.

For simplicity, *POST /agents/search* is recommended for more complex queries. The body might look like:

```
json { "filters": { "capabilities": ["translation"],  
  "supported_languages": ["en", "zh"], "tags": ["nlp"] }, "top": 10 }
```

This returns up to 10 agents that match all the specified filters. Filters are combined with AND logic (the agent must satisfy all conditions). Capabilities and tags are matched by set intersection (the agent must have at least the ones listed).

The response is a JSON array of agent summary objects:

```
json [ { "id": "agent-12345", "name": "Chinese-English Translator",  
  "description": "...", "endpoint": "https://api.example.com/agents/  
  translate", "capabilities": ["translation"] }, ... ]
```

Summary objects include essential fields to help the client decide which agent to use, without returning the full detailed metadata. A client can retrieve full metadata via the single-agent endpoint.

4.3.2. Semantic Query (Natural Language Search)

In addition to attribute-based search, the registry MAY support semantic search where the client describes a need in natural language, and the registry uses AI techniques (embeddings, LLM-based matching) to find relevant agents. This is an optional feature for registries.

* *Endpoint:* POST /agents/search

* *Request Body:* json { "query": "I need an agent that can summarize long legal documents in Chinese.", "top": 5 }

The registry returns a ranked list of agents whose descriptions match the query semantically, possibly including a match score. For example:

```
json [ { "id": "agent-67890", "name": "Legal Document Summarizer",  
"description": "...", "score": 0.93 }, ... ]
```

Semantic search enables flexible discovery beyond exact filter matching, aligning with how users or orchestrating agents might reason about tasks. Registries not supporting semantic search simply ignore the query text and rely on provided filters.

4.3.3. Retrieve Single Agent

- * ***Endpoint:** GET /agents/{id}
- * ***Response:** Full metadata JSON for the specified agent, or ***404*** if not found.

5. Agent Invocation

Once a client discovers a suitable agent, it invokes the agent by sending a request to the agent's endpoint. This section defines the interface for invocation.

5.1. Invocation Request

To invoke an agent, the client sends an HTTP POST request to the agent's invocation endpoint with a JSON body containing the input data for the agent's task.

- * ***Method:** POST
- * ***URL:** The endpoint URL from the agent's metadata (e.g., `https://api.example.com/agents/translate`). If a gateway is used, the URL might be a gateway-provided path.
- * ***Headers:**
 - Content-Type: application/json
 - Authentication header as required (e.g., `Authorization: Bearer <token>` or `X-API-Key: <key>`).
- * ***Body:** A JSON object containing input data as per the agent's input schema.

For example, invoking the translation agent:

```
json { "text": "Hello, how are you?", "source_language": "en",  
"target_language": "fr" }
```

This corresponds to the agent's expected input fields. If the agent had multiple operations and a unified endpoint, there might be an additional field to specify which operation or capability to use. For instance, the JSON could include something like "operation": "translateText" if needed. Alternatively, different operations could be exposed at different URLs (e.g., /agents/xyz/translate vs /agents/xyz/summarize), in which case the operation is selected by the URL and no extra field is required.

The protocol does not fix a specific parameter naming; it defers to the agent's published schema. The only requirement is that the client's JSON must conform to what the agent expects. For interoperability, using clear field names and standard data types (strings, numbers, booleans, or structured objects) is encouraged. Binary data (like images for an image-processing agent) should be handled carefully: typically, binary inputs can be provided either as URLs (pointing to where the data is stored), or as base64-encoded strings within the JSON, or by using a multipart request. This specification suggests that if agents need to receive large binary payloads, they either use URL references or out-of-scope mechanisms (like a separate upload and then an ID in the JSON). The core invocation remains JSON-based for simplicity and consistency.

***Headers:** If authentication is required (see Security section), the client must also include the appropriate headers (e.g., Authorization: Bearer <token> or an API key header) as dictated by the agent's metadata. The invocation request may also include optional headers for correlation or debugging, such as a request ID, but those are not standardized here.

5.2. Invocation Response

The agent (or gateway) will process the request and return a response. The status code and JSON body of the response follow these guidelines:

- *Success (2xx status):** If the agent successfully performed its task and produced a result, the status SHOULD be ***200 OK*** (or ***201 Created*** if a new resource was created as a result, though usually for these actions 200 is fine). The response body will contain the output data in JSON. Ideally, the output JSON conforms to the agent's advertised output schema.

For example, for the translation request above, a success response might be:

```
json { "translated_text": "Bonjour, comment tes-vous?" }
```

Here the JSON structure matches what was described in the agent metadata's outputs. If the output is complex (e.g., multiple fields or nested objects), those should appear accordingly. The response can include other informational fields if necessary (for example, some agents might return usage metrics, like tokens used or time taken, or a trace id for debugging, but these are optional and out of scope of the core spec).

- * ***Client Error (4xx status):*** If the request was malformed or invalid, the agent returns a ***4xx*** status code. The most common would be ***400 Bad Request*** for a JSON that doesn't conform to the expected schema or missing required fields. For example, if the client omitted a required field `target_language`, the agent might respond with 400. The response body **SHOULD** include an error object explaining what went wrong. We define a simple standard for error objects:

```
json { "error": { "code": "InvalidInput", "message": "Required  
field 'target_language' is missing." } }
```

Here, "code" is a short string identifier for the error type (e.g., `InvalidInput`, `Unauthorized`, `NotFound`), and "message" is a human-readable description. The agent can include additional details if available (e.g., a field name that is wrong, etc.). If the error is due to unauthorized access, ***401 Unauthorized*** or ***403 Forbidden*** should be used (with an appropriate error message indicating credentials are missing or insufficient). If the agent ID is not found (perhaps the client used an outdated reference), ***404 Not Found*** is appropriate.

- * ***Server/Agent Error (5xx status):*** If something goes wrong on the agent's side during processing (an exception, a timeout while executing the task, etc.), the agent (or gateway) returns a ***5xx*** status (most likely ***500 Internal Server Error*** or ***502/504*** if there are upstream issues). The response should again include an error object. For example:

```
json { "error": { "code": "AgentError", "message": "The agent  
encountered an unexpected error while processing the request." } }
```

The agent might log the detailed error internally, but only convey a generic message to the client for security. A ***503 Service Unavailable*** might be returned if the agent is temporarily overloaded or offline, indicating the client could retry later.

- * ***Status Codes Summary:*** In short, this protocol expects the use of standard status codes to reflect outcome (200 for success, 4xx for client-side issues, 5xx for server-side issues). Agents should

avoid using 2xx if the operation did not semantically succeed (even if technically a response was generated). For example, if an agent is a composite that calls other services and one of those calls fails, it should propagate an error rather than returning 200 with an error in the data.

5.3. Additional Considerations for Invocation

- * ***Streaming Responses:** Some agents (especially those wrapping large language models) may produce results that are streamed (for example, token-by-token outputs). While this base protocol assumes a request-response pattern with the full result delivered at once, it can be extended to support streaming by using chunked responses or WebSockets. For instance, an agent might accept a parameter like `stream: true` and then send partial outputs as they become available. This is an advanced use case and not elaborated in this draft, but implementers should consider compatibility with streaming if real-time responsiveness is needed.
- * ***Batch Requests:** If a client wants to send multiple independent requests to an agent in one go (for efficiency), the protocol can support that by allowing an array of input objects in the POST body instead of a single object. The response would then be an array of output results. This is optional and depends on agent support.
- * ***Idempotency and Retries:** Most agent invocations are not strictly idempotent (since an agent might perform an action or have side effects), but many are pure functions (e.g., translate text). Clients and gateways should design with retry logic carefully — if a network failure happens, a retry might re-run an operation. It's best to ensure that agents' operations are either idempotent or have safeguards (for example, an operation that sends an email might have an idempotency key).
- * ***Operation Metadata:** In cases where the agent defines multiple operations in its metadata, the invocation interface might allow a generic endpoint that accepts an operation name. Alternatively, each operation could be a sub-resource. This draft leaves the exact mechanism flexible: an implementation could choose one of these approaches. The key is that the invocation uses POST and a JSON body following the agent's schema.

6. Agent Semantic Resolution

Agent Semantic Resolution (ASR) is an optional extension to the discovery mechanism defined in this document. ASR enables a client to resolve a task intent into one or more candidate agents prior to invoking any specific agent.

ASR operates on the following conceptual model:

(Intent, Context, Policy) \rightarrow (Agent Endpoint(s), Invocation Metadata)

The intent represents the task to be performed, while context and policy may include domain constraints, trust requirements, or performance considerations.

6.1. Non-Goals

ASR explicitly does not provide:

- * Name-to-address resolution
- * Global or persistent agent identifiers
- * Replacement for DNS, ANS, or URI-based registries

ASR answers the question "Which agent(s) should handle this task now?" rather than "Where is agent X located?".

7. Semantic Routing Platform

A Semantic Routing Platform (SRP) is a control-plane service that implements ASR. An SRP assists a Host Agent in selecting appropriate agents before standard discovery and invocation procedures are used.

An SRP MAY perform semantic matching, ranking, and policy-based filtering of candidate agents. The SRP does not participate in task execution and does not alter the invocation semantics defined in this document.

Interaction with an SRP is OPTIONAL. Clients that do not support ASR continue to operate using the discovery and invocation mechanisms defined in earlier sections.

8. Backward Compatibility

All discovery and invocation mechanisms defined in previous revisions of this document remain valid and unchanged.

Agent Semantic Resolution is an optional extension. Implementations MAY support ASR incrementally, and registries MAY provide semantic resolution capabilities without affecting existing clients.

9. Security Considerations

Security is a critical aspect of this protocol. All discovery and invocation traffic MUST be protected with TLS [RFC8446], and authentication mechanisms such as OAuth 2.0 [RFC6749] bearer tokens, API keys, or mutual TLS are required except for public discovery endpoints. Registries MUST enforce per-client entitlements, ensuring that both search results and invocation access respect permissions and scopes. Gateways forwarding requests should authenticate themselves to agents, and agents should maintain stable identifiers and use signed responses when integrity is essential. All communication MUST be encrypted, and agents are encouraged to disclose data-retention or logging practices, while sensitive data is best handled by on-premises or certified agents. To mitigate abuse, registries and agents MUST implement rate limiting and quotas, particularly in semantic search scenarios. Trust mechanisms such as certification, test harnesses, or reputation systems may be used to validate agent claims, and metadata fields like "certification" or "quality_score" can inform client trust decisions. Systems SHOULD also provide audit and logging with privacy-aware retention, while clients must treat agent outputs as untrusted until verified, using sandboxing and validation before executing code or commands.

When Agent Semantic Resolution is used, security considerations extend to the pre-invocation phase. Resolution services SHOULD validate agent capability claims, apply policy constraints, and exclude agents that do not meet trust or reputation requirements. Agents deemed unsafe SHOULD NOT be returned as resolution candidates.

10. Example Interaction Flow

1. ***Search:** Client POST /agents/search with {"query":"summarize an English document","filters":{"capabilities":["summarization"],"supported_language":"en"},"top":3}.
2. ***Select:** Registry returns candidate agents with id, name, description, and score. Client retrieves full metadata via GET /agents/{id} if needed.

3. **Invoke:* Client POST to the agent's endpoint (or gateway path) with inputs conforming to agent schema and required auth header.
4. **Handle Response:* Client processes success or error response; may log usage and optionally rate/feedback the agent.

11. IANA Considerations

This document has no IANA actions.

12. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/rfc/rfc2119>>.
- [RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <<https://www.rfc-editor.org/rfc/rfc6749>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/rfc/rfc8174>>.
- [RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <<https://www.rfc-editor.org/rfc/rfc8259>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/rfc/rfc8446>>.
- [RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <<https://www.rfc-editor.org/rfc/rfc9110>>.

Acknowledgments

TODO acknowledge.

Authors' Addresses

Yong Cui
Tsinghua University
Beijing, 100084
China
Email: cuiyong@tsinghua.edu.cn
URI: <http://www.cuiyong.net/>

Yihan Chao
Zhongguancun Laboratory
Beijing, 100094
China
Email: chaoyh@zgclab.edu.cn

Chenguang Du
Zhongguancun Laboratory
Beijing, 100094
China
Email: ducg@zgclab.edu.cn