Network Management Research Group
Internet-Draft
Intended status: Informational
Expires: 18 April 2026

Y. Cui
Tsinghua University
Y. Chao
C. Du
Zhongguancun Laboratory

15 October 2025

HTTP-Based AI Agent Discovery and Invocation Protocol
draft-cui-ai-agent-discovery-invocation-00

Abstract

This document proposes a standardized protocol for *discovery and invocation of AI agents* over HTTP. It defines a common metadata format for describing AI agents (including capabilities, I/O specifications, supported languages, tags, authentication methods, etc.), a semantic search mechanism to discover and match agents based on capabilities, and a unified RESTful invocation interface for calling those agents. The goal is to enable cross-platform interoperability among AI agents by providing a * “discover and match” * mechanism and a *unified invocation entry point* via standard HTTP. 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.

This Internet-Draft will expire on 18 April 2026.

Copyright Notice

Table of Contents

1. Introduction

AI agents (autonomous services powered by AI models) are proliferating rapidly across industries. Multi-agent collaboration scenarios are becoming common, where different AI agents need to work together. However, *technology fragmentation has led to “capability islands” agents from different vendors and platforms often cannot easily interconnect or form coordinated teams*. Many enterprises have deployed AI agents (a PwC survey indicates 79% of companies have done so, with 66% reporting significant productivity gains), yet they face challenges in integrating these diverse agents into unified workflows.

One key challenge is *discoverability*. Currently, finding an appropriate agent for a given task often relies on manual searches or browsing through catalogs, which is inefficient. Agents are distributed across many platforms and repositories, and each may have its own way of categorizing or exposing them, making it complicated for a developer or system to find the best match. For example, some platforms host thousands of AI tools but require users to navigate by predefined categories, and others provide basic search services but still depend on manual addition and configuration of agents by the user. Moreover, different agents might advertise overlapping or unclear capabilities, leading to confusion and the risk of misuse if an agent’s claimed capabilities are not trustworthy.

Another challenge is the *lack of a unified invocation interface*. Once an agent is identified, calling it may involve platform-specific APIs, custom input/output formats, or unique authentication schemes. Integrators must write custom integration code for each new agent service. This heterogeneity hinders the seamless composition of agents from different sources. A unified calling mechanism would allow clients to invoke any compliant agent through a common protocol, hiding underlying platform differences a concept similar to services like OpenRouter, which provide a single API to access models from multiple providers.

Scope of this Document: This document addresses the above challenges by specifying:

* A standard *agent metadata schema* to describe an AI agent’s capabilities, interfaces, and requirements.

* A *discovery and matching API* (RESTful) that allows clients to find suitable agents via semantic search of agent metadata.

* A *unified invocation API* that defines how to call any compliant AI agent and receive results in a consistent format.

* *Security and authentication guidelines* to ensure only authorized and safe interactions with agents.

* An *example interaction flow* illustrating how a client can discover and invoke an agent using this protocol.

By defining these elements, we aim to foster an open ecosystem where AI agents from different vendors can be registered in a common directory, searched by capability, and invoked in a uniform way. This will reduce integration effort and enable dynamic "agent teams" to be composed on the fly, as all agents adhere to a common invocation standard.

2. Terminology

For clarity, this draft uses the following terms:

* *AI Agent (or simply Agent):* An AI-driven service or tool that performs a specific task or set of tasks. An agent typically exposes a network API (in this context, an HTTP-based API) [RFC9110] to accept requests and return results. It could be a large language model interface, an image analysis service, an autonomous task-solving agent, etc.

* *Agent Metadata (Agent Description or *Agent Card*):* A machine-readable description of an agent's capabilities and interfaces. This is often a JSON document containing the agent's identity, functionality, input/output specifications, supported languages, tags for search, and details on how to invoke and authenticate to the agent. The metadata acts as the agent's "profile" in the discovery system.

* *Client:* An entity (application, service, or another agent) that searches for and invokes agents via this protocol.

* *Registry / Discovery Service:* A service implementing the discovery and matching API as defined in this document. It indexes agent metadata (from one or many providers) and allows clients to search for agents. In some deployments, this could be a centralized agent directory, whereas in others it could be a distributed or federated index. The registry may also facilitate the unified invocation (as a gateway) or simply provide the information for direct invocation.

* *Unified Invocation Endpoint:* The standardized interface (HTTP method, path, and data format) through which a client sends requests to an agent and receives responses, abstracting away the agent’ s internal implementation details. This could be implemented by each agent (each agent hosting a conformant API) or by a gateway that routes requests to agents.

3. Architecture

The following ASCII diagram illustrates the high-level architecture of the discovery and invocation protocol:

‘‘‘text +------------------+ | Client | +------------------+ | ^ | Discovery & Search | | | Unified Invocation (HTTP API) v | | (POST /agents/{id}/invoke) +----------------------+ | Registry/Gateway | +----------------------+ ^ | Search Results (metadata, score) | Forwarded Invocation | v | +--------------+ | | Agent Host 1 | | +--------------+ | | | v | +--------------+ | | Agent Host 2 | | +--------------+ | | | v | +--------------+ | | Agent Host N | | +--------------+ | +------------------+-----------------------------+ | | | (Optional Direct Invocation, dashed line) | v v +--------------+ +--------------+ +--------------+ | Agent Host 1 | | Agent Host 2 | ... | Agent Host N | +--------------+ +--------------+ +--------------+

Figure 1: Client Registry/Gateway Agent Hosts (Direct and Gateway Invocation)

‘‘‘

This figure illustrates the overall architecture of the discovery and invocation protocol.

Note: The Client first uses the Registry/Gateway for discovery (search and metadata retrieval). For invocation, two modes are supported: (1) *Gateway Invocation*, where the Client calls the unified endpoint and the Registry forwards the request to the appropriate Agent Host; and (2) *Direct Invocation*, where the Client may optionally call an Agent Host directly. This dual-path design balances interoperability, centralized control, and low-latency flexibility.

4. Agent Metadata Format

* Each agent discoverable via this protocol must publish an *Agent Metadata* record (an “Agent Card”), typically as a JSON [RFC8259] object, optionally available at a standard URL (e.g., /.well-known/agent-card.json). Metadata enables both human understanding and machine processing for discovery and validation.

  Key fields include:

  - *id*: Unique identifier (UUID, slug, or URI).
  - *name / description*: Human-readable label and short description of the agent’ s purpose.
  - *capabilities*: Array of tags describing core functions (e.g., translation, vision).
  - *operations*: (Optional) Structured list of operations, each with name, description, input/output schema, and endpoint. For simple agents, inputs/outputs at the top level may be used instead.
  - *endpoint*: Base URL or path to invoke the agent.
  - *authentication*: Required auth method (none, API key, OAuth2, mTLS).
  - *Optional fields*: supported_languages, tags, protocols, version, provider, license, certification, last_update.

  This metadata format provides a consistent way to describe agent identity, capabilities, and invocation details, supporting secure discovery, filtering, and interoperability across registries.

Example Agent Metadata (JSON):

json { "id": "translator-001", "name": "MultiLang Text Translator", "description": "Translates text between English and French.", "capabilities": ["translation", "language"], "inputs": { "text": "string", "source_language": "string", "target_language": "string" }, "outputs": { "translated_text": "string" }, "supported_languages": ["en", "fr"], "tags": ["NLP", "translation", "English", "French"], "authentication": "API Key", "endpoint": "https://api.example.com/agents/translator-001", "version": "1.0", "provider": "ExampleAI", "certification": { "verified": true, "quality_score": 4.8 } }

Figure: An illustrative example of an agent's metadata. This agent can translate text between English and French. It requires an API Key for authentication and is marked as verified with a high quality score. In practice, the metadata format could be extended or serialized in different ways (JSON, JSON-LD, etc.), but a concise JSON structure as shown is the default.

The metadata format is crucial for discovery; it should provide enough information for a registry service to index the agent effectively (including semantic content from name, description, and tags) and for clients to know how to interact with the agent. Communities working on agent interoperability have stressed the importance of a comprehensive, machine-readable "agent card" schema that covers capabilities, interfaces, and security requirements. Formal schemas for inputs/outputs allow programmatic integration and even automated composition of agents, and including protocol/auth details ensures that invocation can be automated. Each agent should keep its metadata updated (preferably accessible at a stable URL) to reflect its current capabilities and interface. Caching and indexing by discovery services can be aided by standard HTTP cache headers on the metadata resource (e.g., ETag, Last-Modified). In the future, a dedicated media type for agent metadata could be registered (for example, application/agent-card+json) to make it easier to identify and validate these documents.

5. Discovery and Matching Mechanism

A registry exposes RESTful endpoints for listing, searching, and retrieving agent metadata. The discovery API supports both structured filters and semantic (natural language) search. Implementations may use embedding-based similarity or rule-based matching; the protocol only standardizes the interface and result format (including an optional relevance score).

5.1. Discovery Service API

At minimum, the discovery service provides the following endpoints:

* *List All (Public) Agents:* GET /agents (or /agents/public) returns a list of agent metadata entries that are publicly available for discovery. This endpoint allows unauthenticated browsing of agent descriptions (for agents that are open to all). It may support query parameters for basic filtering (e.g., by capability or tag) and pagination. The response would typically be a JSON object containing an array of agents (each possibly abridged to key fields like id, name, description, etc.) and perhaps a count or pagination links.

* *Search Agents:* POST /agents/search allows the client to submit a search query and criteria to find matching agents. A POST method is used (rather than GET) to allow complex query parameters and to support semantic query content in the request body. This endpoint requires the client to be authenticated if the search should include private or entitled agents (see Security and Authentication below), ensuring that each client sees only the agents they are allowed to access. The request body and response format are described below.

These endpoints can be part of a single "registry" service. In federated scenarios, multiple registries might exist (e.g., enterprise-specific ones, or vendor-specific ones), but all would adhere to the same API structure so that a client could query any of them in a standard way.

5.2. Search Request and Response

Search Request: The client sends a JSON request to POST /agents/search describing what it is looking for. The format may include:

* *query* (string): A text description of the desired capability or task. This can be natural language (e.g., "Find an agent that can summarize a document" or "image classification of medical images"). The discovery service will perform a semantic match of this text against agent metadata (such as agent descriptions, names, and tags) using NLP techniques (embedding-based similarity, etc.).

* *filters* (object, optional): Additional filtering criteria to narrow down results. For example:

  - capability or capabilities: specify one or more required capability tags (e.g., "translation").

  - language or supported_language: specify required language (e.g., "zh" to find agents that support Chinese).

  - authentication: specify if only certain auth types are acceptable (e.g., {"authentication": "none"} to find open/public agents).

  - provider or other metadata fields can also be used as filters.

* *top* (integer, optional): The maximum number of results to return (for pagination or limiting result set). For example, top: 10 for top 10 matches.

* *skip* (integer, optional): The number of results to skip from the start (for pagination offset). For instance, skip: 10 to get the next page after the top 10.

* *ranked* (boolean, optional): If true (default), results are returned with a relevance score. If false, the service may return unranked or alphabetical list (mainly useful for listing endpoints).

* *include_metadata* (boolean, optional): If true, include full metadata for each agent in the results. If false or omitted, the service might return only a summary (e.g., id, name, description, score) to reduce payload size, with an option to fetch full details separately (like via GET /agents/{id}).

A simple example of a search request JSON might look like:

json { "query": "translate English to Spanish", "filters": { "capabilities": ["translation"], "supported_language": "es" }, "top": 5 }

This asks for up to 5 agents that can translate into Spanish, using a semantic match with the query text.

Search Response: The discovery service responds with a JSON object containing the search results. The typical fields in the response could be:

* *results* (array): A list of agent results, each containing:

  - id (string): the agent's identifier.

  - name (string): the agent's name.

  - description (string): a short description (this helps the client understand why it was matched).

  - score (number, optional): a relevance score or confidence value for the match, typically between 0 and 1 (where 1 means a perfect match). This is based on semantic similarity of the query to the agent's metadata. The score can help the client rank or filter results if needed.

  - Other metadata fields: depending on the include_metadata parameter and the service's default behavior, additional fields like capabilities, supported languages, etc., might be included. If full metadata is requested, this could even embed the entire agent metadata object for each result.

* *count* (integer, optional): The total number of matching agents (if known), which can be useful for pagination UI.

* *top*, *skip* (integers, optional): Echoing back the request's paging parameters or indicating the current slice of results.

* *query* (string, optional): Echo of the query for reference.

* *search_time* (number, optional): Time taken to execute the search (in milliseconds, perhaps).

Example of a search response:

json { "results": [{ "id": "translator-001", "name": "MultiLang Text Translator", "description": "Translates text between English, French, and Spanish.", "score": 0.93 }, { "id": "spanish-gpt", "name": "SpanishGPT Agent", "description": "Large language model specialized in Spanish outputs.", "score": 0.89 }], "count": 2, "query": "translate English to Spanish", "top": 5, "skip": 0 }

In this example, two agents were found. The first has a high match score of 0.93, likely because it explicitly mentions translating to Spanish in its metadata. The second is a more general LLM that can output Spanish, with a slightly lower relevance. The client could decide to pick the top result (translator-001) for the task.

The discovery mechanism leverages *semantic search* internally. This means the registry service might transform agent descriptions and queries into vector embeddings and do a nearest-neighbor match, or apply NLP to understand the query's intent. The protocol, however, abstracts those details — it simply defines the interface (what clients send and receive). The scoring mechanism and algorithm are implementation-specific, but including a score in results allows transparency and further client-side logic (for example, choosing only agents above a certain confidence, or showing the score in a UI).

In addition to the free-text search, the registry could support structured queries (for instance, a client could search by specifying a capability tag and a required output format, etc.). Those can be expressed via the filters object or through specialized endpoints. For example, a GET request like GET /agents?capability=translation&language=es could be defined as a shorthand for certain filters (especially for public agents). This draft prioritizes the semantic POST search for flexibility, but does not preclude simpler query interfaces.

It's important to note that access control can be enforced at search time: an authenticated client will only see agents it is allowed to invoke. For example, an enterprise might have private agents that only employees (with a token) can discover. The registry would then combine public agents and those private agents accessible to the client's identity when returning search results. If a client is unauthenticated and hits the search endpoint (and if the service allows it at all), it might either reject the request or return only publicly discoverable agents.

5.3. Matching and Ranking

The *matching* process uses the agent metadata fields defined earlier. Key fields that influence matching include:

* The agent's name and description: for semantic similarity to the query text.
* capabilities and tags: for direct filtering or boosting of relevance if the query contains those terms.
* supported_languages: for matching queries that specify language requirements.
* Possibly input_schema/output_schema: for advanced use-cases where the query might specify required input/output types (though this is more likely handled via filters than semantic match).
* Any other textual field in metadata could be indexed and matched (even provider or operations.description, etc., depending on implementation).

The *matching score* in the response is typically a normalized similarity measure. The exact scale and meaning might vary (cosine similarity of embeddings, for instance), but this specification assumes a 0.0 to 1.0 range where higher is better. The client should treat the score as a relative indicator. The registry may also rank results by other heuristics (popularity, performance, freshness of agent, etc.) in addition to pure semantic score, but such ranking policies are outside the scope of this protocol. If non-semantic factors are included, the score should still aim to reflect overall relevance.

5.4. Example of Discovery Process

1. *Agent Registration (pre-discovery):* Providers of agents register their agent metadata with the registry service. This could be done through an administration interface or by the registry crawling known agent metadata URLs. (The protocol for registration is not fully defined in this draft, but it could simply be an admin uploading the agent's JSON metadata or providing the URL of the agent's card. Some proposals suggest agents host their own metadata at a well-known URL, and registries periodically fetch and index them.)

2. *Client Query:* A client (e.g., an application looking for a translation service) sends a POST /agents/search request with a query, e.g., _ "translate product descriptions to Chinese" _.

3. *Semantic Matching:* The discovery service processes the query. It might detect the language requirement ("to Chinese") and capability ("translate") and use these to filter or boost candidates. It computes similarity between the query and each agent's metadata (perhaps using vector embeddings of the text). It filters out any agents the client is not authorized to see.

4. *Result Delivery:* The service responds with a list of matching agents, including their IDs, names, descriptions, and scores. The client inspects the results, maybe chooses the top one, or presents options to a user if this is an interactive scenario.

5. *Agent Selection:* The client picks an agent from the list that best fits the need (for example, the one with highest score that meets any other criteria like trust or cost). Now the client is ready to invoke the chosen agent.

The next section describes the invocation process once an agent has been discovered.

6. Unified Invocation Interface

Clients invoke agents via a standardized HTTP interface. Two deployment models are supported:

* *Direct Invocation:* Call the agent's endpoint (agent-hosted API).

* *Gateway/Proxy Invocation:* Call a registry/gateway with an agent identifier (e.g., POST /agents/{agent_id}/invoke), and the gateway routes/translates as necessary.

Invocation characteristics:

* Method: POST

* Content-Type: application/json (unless otherwise specified)

* Request body: conforms to the agent's input_schema or inputs definition (or includes an operation field if required)

* Authentication: as declared in authentication metadata (e.g., Authorization: Bearer <token>)

6.1. Invocation Request

The invocation is done via *HTTP POST* to the agent's invocation URL. The URL could be:

* In direct mode: the endpoint from the agent metadata (for example, https://agent-host.example.com/invoke or similar).

* In gateway mode: a path on the gateway that includes the agent's ID, for example: POST /agents/{agent_id}/invoke on the registry or router service.

In either case, the request method is POST (since we are typically sending a complex payload and possibly performing an action on the server side, which is not idempotent). The content type of the request *MUST* be application/json (unless the agent specifically expects binary input; see below). The body of the request contains the input data for the agent, formatted according to the agent's specified input schema or format.

For an agent with a single operation, the JSON body should include all the necessary parameters defined in the agent's inputs. For example, using the translator agent example above, a request might look like:

POST https://api.example.com/agents/translator-001/invoke *Content-Type:* application/json

Request body:

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 an HTTP 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.

6.2. Invocation Response

The agent (or gateway) will process the request and return a response. The HTTP 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 HTTP 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.

6.3. 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 HTTP 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 in either case, the invocation uses HTTP POST and a JSON body following the agent's schema.

7. 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.

8. 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.

9. IANA Considerations

This document has no IANA actions.

10. Normative References

[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>.

[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