Independent Submission
Internet-Draft
Intended status: Experimental
Expires: 19 April 2026

Y. Narvaneni
Independent Researcher

16 October 2025

The agent:// Protocol -- A URI-Based Framework for Interoperable Agents
draft-narvaneni-agent-uri-02

Abstract

This document defines the agent:// protocol, a URI template-based framework as described in RFC 6570 for addressing, invoking, and interoperating with autonomous and semi-autonomous software agents. It introduces a layered architecture that supports minimal implementations (addressing and transport) and extensible features (capability discovery, contracts, orchestration). The protocol aims to foster interoperability among agents across ecosystems, platforms, and modalities, enabling composable and collaborative intelligent systems.

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 19 April 2026.

Table of Contents

1. Introduction

The rise of intelligent software agents necessitates a standardized way to identify, invoke, and coordinate them across diverse platforms. While protocols like HTTP [RFC9110] provide a transport mechanism for static APIs, agents differ significantly in behavior, output variability, and interaction patterns. The agent:// proposes a URI scheme and resolution model designed to complement existing

agent communication protocols and libraries like [Agent2Agent], [FIPA-ACL], Contract Net Protocol [FIPA-CNP], [LangChain], Model Context Protocol [MCP], [AutoGen], [SemanticKernel] etc. This document defines a resolution algorithm that maps agent:// URIs to transport endpoints using HTTPS, WebSocket, or local bindings as defined later. It serves as an addressing and discovery layer that works alongside these communication protocols.

The agent:// protocol supports diverse agent deployment models through a unified addressing scheme:

* Cloud-based agents accessible via standard web protocols

* Local agents running on the user’s device through the agent+local:// scheme

* On-premises agents within organizational boundaries

* Decentralized agents operating across distributed networks

This flexibility addresses a critical gap in current agent ecosystems, enabling applications (including browsers) to discover and invoke agents consistently regardless of where they’re hosted. By providing standardized URI[RFC6570] patterns for both remote and local agents, the protocol simplifies previously complex integration scenarios like browser-to-local-agent delegation for privileged operations.

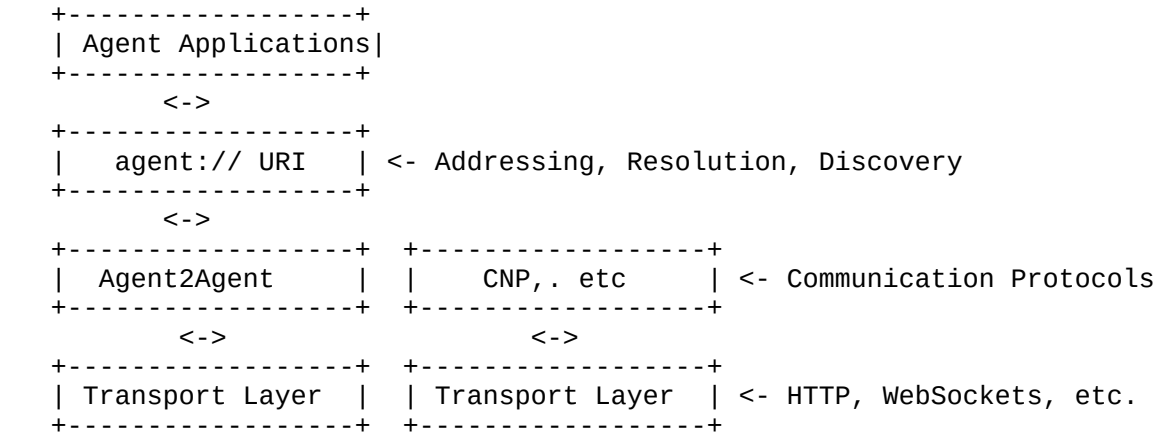

Figure 1: Agent Protocol Stack Architecture

The agent:// protocol supports:

* Unique and resolvable addressing of agents

* Optional self-describing capabilities

* Consistent invocation semantics over existing transports

* Progressive support for advanced patterns like delegation, collaboration, and orchestration

This document outlines the specification for the agent:// protocol, beginning with its URI scheme and extending through capability description, transport bindings, and extensibility patterns.

A reference implementation of the agent:// protocol is available to demonstrate resolution, transport bindings, capability discovery, and orchestration patterns. Implementers and adopters can find this example implementation at: [AGENT-URI-REPO]

2. Terminology

* *Agent*: An autonomous or semi-autonomous software entity that can receive instructions and perform actions.

* *Agent Descriptor (agent.json)*: A machine-readable document that describes an agent’s identity, capabilities, and behavior.

* *Capability*: A self-contained function or behavior an agent offers.

* *Resolver*: A service or mechanism that maps a URI to a network endpoint or metadata.

* *Invocation*: The act of calling a capability on an agent with input parameters.

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 RFC 2119, BCP 14 [RFC2119], [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Protocol Scope and Layering

The agent:// protocol is designed as a layered framework:

Layer	Purpose	Mandatory
URI Scheme	Unique addressing	Yes
Transport Binding	Mechanism for invocation (e.g., HTTP, WSS, Matrix, IPC)	Yes
Agent Descriptor	Self-describing agent interface	Optional
Resolution Framework	Maps agent URIs to endpoints	Optional
Application Semantics	Shared vocab for capability naming	Optional

Table 1: Protocol Layering Structure

This layering allows implementations to adopt minimal or full-featured configurations, depending on their needs.

4. URI Scheme Specification

The format of agent:// URIs is:

```
agent://[authority]/[path]?[query]#[fragment]
agent+<protocol>://[authority]/[path]
```

Figure 2: Agent URI Format

Examples:

* agent://example.com/planning/gen-iti?city=Paris
* agent://planner.example.com/claude?text=Hello
* agent+https://example.com/assistants/chatgpt?query=hello
* agent+local://examplelocalagent
* agent:///did:web:example.com:agent:researcher/get-article?doi=...

To resolve agent://<authority>/<path>?<query>: 1. Fetch https://<authority>/.well-known/agents.json (if present) 2. Locate <path> mapping -> agent descriptor URL 3. Fetch descriptor (agent.json) 4. Extract transport.endpoint or transport metadata 5. Invoke using indicated method or default: - GET for read-only, POST for state-changing 6. If none found -> agent+https:// fallback

4.1. Components

* *Authority*: Uniquely identifies the agent or agent namespace (e.g., DNS or DID).

* *Path*: Specifies the agent being invoked. The [path] is opaque to agent:// and can represent either a namespace or direct capability.

* *Query*: Contains serialized parameters. Query parameters SHOULD be URL-encoded as key=value pairs. If more complex structures are needed, clients SHOULD use HTTP POST requests with application/json bodies rather than base64-encoding payloads into query parameters.

* *Fragment*: Optional reference for context or sub-capability.

* The optional +<protocol> indicates explicit transport binding.

* If not specified, clients use resolution or fall back to HTTPS-based invocation.

4.2. ABNF for agent:// URI

```
agent-uri      = "agent" ["+" protocol] ":" hier-part
                 [ "?" query ] [ "#" fragment ]
; hier-part from RFC3986, allowing both //authority and path-rootless

protocol       = 1*( ALPHA / DIGIT / "-" )
authority      = [ userinfo "@" ] host [ ":" port ]
                 ; <authority, defined in RFC3986, Section 3.2>
path           = path-abempty
                 ; begins with "/" or is empty.
                 ; Defined in RFC3986, Section 3.3
query          = *( pchar / "/" / "?" )
                 ; <query, defined in RFC3986, Section 3.4>
fragment       = *( pchar / "/" / "?" )
                 ; <fragment, defined in RFC3986, Section 3.5>

pchar          = unreserved / pct-encoded / sub-delims / ":" / "@"
unreserved     = ALPHA / DIGIT / "-" / "." / "_" / "~"
pct-encoded    = "%" HEXDIG HEXDIG
sub-delims     = "!" / "$" / "&" / "'" / "(" / ")" /
                 "*" / "+" / "," / ";" / "="

; Character sets like pchar, unreserved, etc. are defined in RFC3986
```

Figure 3: ABNF Grammar for agent:// URI Scheme

5. Resolution Framework

Every agent MAY expose a self-describing document at:

<scheme>://<domain>/<path-to-agent>/agent.json

Agents reachable over the network SHOULD publish /.well-known/agent.json

If a single agent is deployed at the top level then it should be under /.well-known.

* /.well-known/agent.json -- For single-agent deployments (compatible with AgentCard)

Multi-agent domains MAY additionally expose /.well-known/agents.json.

* /.well-known/agents.json -- For multi-agent domains (maps agent names -> descriptors)

This descriptor is OPTIONAL but RECOMMENDED. It enables capability discovery, transport resolution, and compatibility with ecosystem tools.

When present, the descriptor MAY use the [AgentCard] (as defined by Agent2Agent protocol by Google as of April 2025) schema as one possible format, or any equivalent [JSON-LD11] based structure that expresses the agent's identity, capabilities, and behavioral metadata.

If the agent is deployed at a subdomain (e.g., planner.example.org), the agent descriptor SHOULD be published at /.well-known/agent.json on that domain.

Resolvers MUST restrict fetches to HTTPS schemes and MUST NOT resolve to private or loopback IP ranges. Resolvers SHOULD verify TLS certificates and may require signed descriptors.

5.1. Ecosystem Registries

Domains MAY publish:

https://<domain>/.well-known/agents.json

This file should map agent names to their agent.json URLs for simplified enumeration. It is OPTIONAL but RECOMMENDED for better ecosystem interoperability.

Implementations MAY support resolution of agent URIs via:

* Static resolution maps

* DID resolution

* WebFinger or custom resolvers

Resolvers SHOULD support caching and capability introspection where applicable.

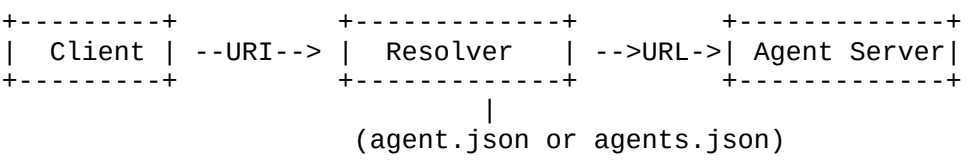

Figure 4: Agent URI Resolution Process

Example:

```
{
  "agents": {
    "planner": "https://planner.example.com/.well-known/agent.json",
    "translator": "https://example.com/translator/agent.json"
  }
}
```

Agents SHOULD use standard HTTP caching mechanisms (Cache-Control, ETag, Last-Modified) to enable efficient resolution and minimize unnecessary descriptor fetches. Clients SHOULD respect these headers and cache descriptors appropriately

5.2. Trust Anchors

Domains MAY use trust anchors (e.g., DNSSEC, HTTPS certificates, or DID-based verification) to enhance identity assurance.

A practical example of URI resolution, agent descriptor fetching, and caching strategies is included in the reference implementation available at: [AGENT-URI-REPO]

6. Transport Bindings

6.1. Explicit Transport Binding

Use the agent+<protocol>:// scheme for clarity:

Transport	Format	Method	Description
HTTPS	agent+https://	GET/POST	Secure HTTP-based invocation
WebSocket Secure	agent+wss://	NDJSON stream	Real-time streaming
Local	agent+local://	IPC/broker call	Runtime-registered local agents
Unix Socket	agent+unix://	Unix domain socket	IPC for co-located agents

Table 2: Transport Binding Formats

The agent+<transport>:// scheme allows explicit declaration of such bindings, enabling clarity, extensibility, and optimized routing. When no explicit transport is declared, clients MAY rely on resolution metadata (e.g., agent.json) or default to HTTPS-based invocation.

This flexibility ensures the protocol can adapt to different performance, privacy, or coordination requirements while remaining consistent at the addressing and invocation layer.

Local agents should be accessed using:

agent+local://<agent-name>

agent+local:// requires explicit user consent and origin binding. Implementations MUST prompt before first use.

This allows agent runtimes to register their presence using a local resolver (e.g., via IPC, sockets, or service registry). The transport mechanism is implementation-specific.

The agent+local:// scheme specifically addresses the current lack of standardized methods for browser-based applications to invoke locally installed agents. This enables web applications to delegate tasks to local agents that can perform privileged operations such as file system access, desktop automation, or hardware interaction - capabilities that are typically restricted in browser environments. Security considerations for such invocations are discussed in Section 10.

6.2. Default Fallback Behavior

If the protocol is omitted (i.e., agent:// is used), clients:

1. Check .well-known/agents.json (if available)
2. Retrieve the agent descriptor at agent.json for the specified path
3. Use the transport or endpoint hints from the descriptor

If nothing is found, clients MAY fall back to:

* HTTPS (default transport protocol)
* HTTP POST if payload present, otherwise GET

Note: This fallback behavior is provided for convenience and basic interoperability, but production systems SHOULD prefer explicit transport bindings or resolver-based discovery for robustness and clarity.

Clients SHOULD prefer explicit transport bindings (agent+https://) where available, and fall back to resolution-based discovery (agent://) when agent transport metadata is reliably available. Explicit binding reduces resolution ambiguity and improves latency.

6.3. Use Cases and Recommended Bindings

The following table outlines some use cases and recommended bindings

Use Case	Recommended Binding
Agent with known HTTPS endpoint	agent+https://
Local runtime agent	agent+local://
Dynamic/multi-transport agents	agent:// with agent.json
Inter-agent calls within a known context	agent:// or agent+matrix://

Table 3: Recommended Bindings for Common Use Cases

7. Capability Framework

Agents SHOULD expose a descriptor document at:

<agent-base-path>/agent.json

This descriptor MAY follow:

* The AgentCard structure (as defined by Google’s Agent2Agent protocol as of April 2025), or another equivalent format

* Any format other than AgentCard SHOULD be expressed in [JSON-LD11] to enable semantic discovery

Agent descriptors SHOULD include:

* Agent name and version

  - MAY include supportedVersions indicating the list of older versions and their end-points.

  - Versioning should follow [SemVer] or later

  - Clients SHOULD verify compatibility based on documented major, minor, and patch versions

* Human-readable description

* Input/output schemas (e.g., JSON Schema)

* Capability list with IDs, descriptions, tags, version

* Optional behavioral metadata (e.g., isDeterministic, expectedOutputVariability, requiresContext: boolean, memoryEnabled: boolean, responseLatency: "low" | "medium" | "high", confidenceEstimation: boolean)

  - isDeterministic (boolean): Indicates whether repeated calls with identical inputs yield identical outputs.

  - expectedOutputVariability: indicates typical variability in outputs, similar to temperature setting

  - responseLatency: Expected response time.

  - requiresContext (boolean): Indicates whether the input needs context or the agent can work on its own

  - memoryEnabled (boolean): Indicates whether the agent will remember the interactions

* Optional transport or invocation hints

* Optional authentication or permission requirements

* Optional state management practices

* Optional interactionModel to indicate a way to interact (e.g. agent2agent, fipa-acl, kqml, contract-net, emergent etc). If mentioned, the message payload SHOULD follow the model’s defined parameters if any.

Agents MAY expose inputFormats and outputFormats per capability using standard MIME types (e.g., application/json, application/ld+json, application/fipa-acl).

Agent descriptors SHOULD include input/output schemas (e.g., JSON Schema) and MAY document content negotiation support via the contentTypes field per capability. This allows clients to understand and negotiate payload encoding, enabling interoperability across ecosystems that use JSON, [JSON-LD11], RDF/XML, [FIPA-ACL], or other formats.

Clients MAY use standard negotiation mechanisms such as Content-Type and Accept headers (in HTTP), or envelope metadata (in protocols like [JSON-RPC], [Matrix], etc.).

Implementations MAY advertise protocol compatility via metadata fields such as interactionModel, orchestration, or supported envelopeSchemas etc. These metadata fields enable clients and agent runtimes to interoperate across heterogeneous ecosystems and communication models.

This extensibility ensures agent:// can serve as a unifying addressing and invocation layer, bridging agents that follow established standards, platform-specific conventions, or learned behaviors in dynamic environments.

If an agent.json is provided, it SHOULD contain at least: name, version, and one or more capabilities.

Clients SHOULD explicitly specify the agent version either as a URI path segment, query parameter (?version=3.1.4), or HTTP header (X-Agent-Version). If omitted, servers SHOULD assume the latest version. Agents MUST document their preferred method for version negotiation clearly in their descriptor. Major version increments indicate breaking changes; clients default only within same major.

While .well-known/agents.json MAY be used to enumerate all available agents under a domain, the individual agent.json files serve as the canonical source of truth.

All published descriptors MUST use media type application/agent+json (or JSON-LD profile).

Expressing descriptors in [JSON-LD11] enables semantic interoperability and supports alignment with common web-based data models.

Implementers MAY choose to embed, proxy, or map to other protocols within the agent.json descriptor or transport bindings, allowing for seamless orchestration and hybrid deployments.

8. Interaction Patterns {# interaction-patterns}

Supported interaction types include:

* Request/Response (synchronous)

* Deferred response (polling or webhook) SHOULD include a taskId and polling interval hint.

* Streaming responses (e.g., Server-Sent Events, WebSocket). Streaming responses over agent+wss:// SHOULD use newline-delimited JSON (NDJSON)

* Delegated invocation (calling other agents on behalf of user)

* Asynchronous event notifications via HTTP webhooks or WebSockets. Event notifications if available SHOULD include event types, payloads, and identifiers.

All interaction patterns (e.g., streaming, event-driven, polling) are transport-agnostic but MAY impose format constraints (e.g., NDJSON over WebSockets).

Agents SHOULD include traceparent or X-Task-ID headers to correlate multi-agent workflows.

Agents SHOULD include status and confidence metadata in responses where applicable.

8.1. Stateful Interactions {# stateful-interactions}

The agent:// protocol leverages HTTP’s established mechanisms for state management. Clients and agents SHOULD use standard HTTP headers or query parameters to pass identifiers such as sessionId or taskId. Agents MAY maintain state across interactions using these identifiers. Clients and agents SHOULD agree on session semantics via capability descriptors or invocation headers.

Non-HTTP transports SHOULD include session or task identifiers within message envelopes (e.g., JSON-RPC headers, WebSocket message metadata, Matrix events). These fields SHOULD follow naming conventions similar to sessionId, taskId, etc.

When the transport lacks a native header mechanism, agents SHOULD extract session information from the body or envelope metadata.

When content negotiation fails or the requested format is not supported, agents SHOULD respond with a 406 Not Acceptable HTTP error or equivalent, and MAY include supported formats in the response metadata.

8.1.1. Recommended practices:

* Use HTTP headers (e.g., X-Session-ID, X-Task-ID) or query parameters for session and task identifiers.

* Clearly document state identifiers and their expected lifecycle in the agent’s descriptor (agent.json).

Example:

```
GET /tasks/1234 HTTP/1.1
Host: planner.example.com
X-Session-ID: abcde-12345
```

8.2. Orchestration Patterns

Agents MAY invoke other agents as part of delegated or composite tasks. Agents SHOULD explicitly provide orchestration workflows, delegation chains, or composite interactions either in their agent.json or in their response metadata.

8.3. Typical Interaction Flows

8.3.1. Client-to-Agent Interaction

A typical user-driven invocation of an agent using the agent:// protocol follows these steps:

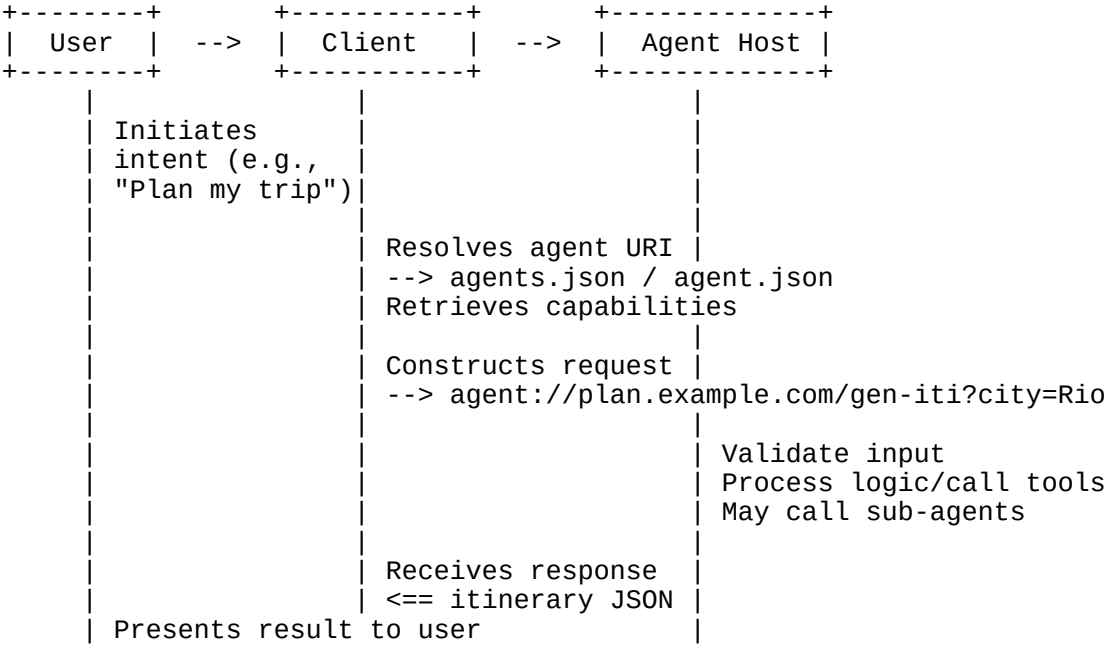

Figure 5: Client-to-Agent Interaction Flow

Notes:

* The client MAY handle fallback logic if the agent cannot be resolved initially.

* Authentication MAY be required before invocation.

* The invocation can be a simple GET or POST depending on input size and structure.

8.3.2. Agent-to-Agent Interaction

Agents MAY interact with each other using agent:// URIs to delegate tasks or compose workflows.

Example: A planning agent invoking a translation agent:

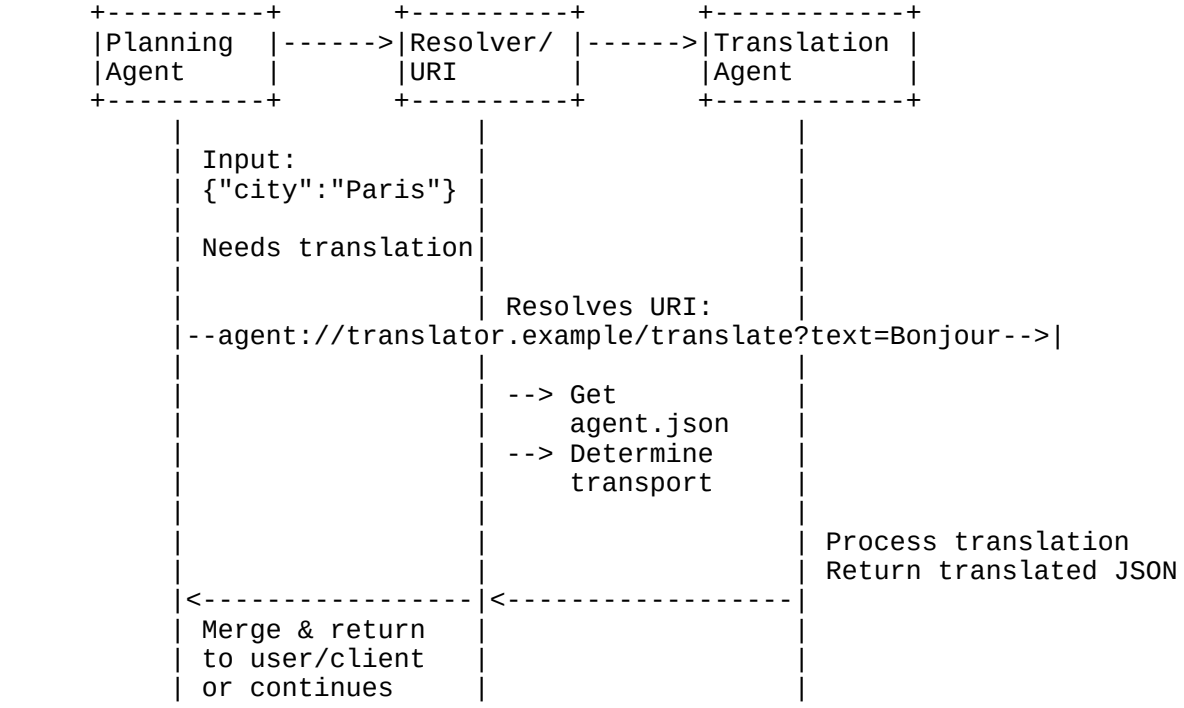

Figure 6: Agent-to-Agent Interaction Flow

Chaining Behavior:

* The invoking agent MAY include X-Task-ID, X-Delegation-Chain, or equivalent headers.

* The response MAY include intermediate metadata such as confidence, sourceAgent, taskTrace, or timeTaken.

9. Error Handling {# error-handling}

The agent:// protocol MAY leverage HTTP standard status codes for signaling errors. Implementations MAY return errors using standard HTTP status codes along with structured JSON error responses conforming to [RFC9457] ("Problem Details for HTTP APIs").

**Recommended HTTP status codes include (but are not limited to)

Status Code	Meaning
400	Bad Request (e.g., invalid parameters)
401	Unauthorized
403	Forbidden
404	Capability or resource not found
409	Conflict (e.g., state mismatch)
429	Too Many Requests (rate limiting)
500	Internal Server Error
503	Service Unavailable

Table 4: Recommended HTTP status codes

Example:

```
HTTP/1.1 404 Not Found
Content-Type: application/problem+json

{
  "type": "https://example.com/errors/capability-not-found",
  "title": "Capability Not Found",
  "status": 404,
  "detail": "The requested capability 'gen-iti' was not found.",
  "instance": "/planner/gen-iti"
}
{: #fig-error-response title="Example HTTP Error Response"}
```

This format is not prescriptive but aims to encourage consistency. Implementations MAY adapt the error schema based on their transport layer (e.g., JSON-RPC, HTTP status + body, WebSocket messages).

For non-HTTP transports (e.g., WebSockets, Matrix), agents SHOULD still return structured errors using similar JSON structures (type, title, detail, status), encapsulated within the transport's native message envelope (e.g., JSON-RPC error objects, Matrix event content fields). Implementers SHOULD document chosen structures clearly in their capability descriptors.

Where applicable, implementations SHOULD align with existing conventions such as:

* JSON-RPC error objects (code, message, data)
* [OpenAPI] or REST error payloads
* [GraphQL] errors array format

Recommended error categories:

* CapabilityNotFound
* InvalidInput
* AmbiguousResponse
* Timeout
* PermissionDenied

Clients SHOULD parse and utilize these structured responses to handle errors gracefully.

10. Security and Privacy Considerations

The agent:// protocol explicitly relies on widely-adopted HTTP authentication and authorization standards. Agents SHOULD support standard authentication and authorization schemes such as OAuth2 (Bearer tokens), API keys, or signed payloads. When using HTTPS, mutual TLS MAY be employed. JSON Web Tokens (JWT) are RECOMMENDED for conveying signed claims between agents. For agent+local://, browsers and runtimes MUST enforce same-origin policy, token handshake, and user mediation before invoking a local agent.

Security extensions MAY include:

* OAuth2 [RFC6750] Bearer Tokens
* JSON Web Tokens (JWT [RFC7519])
* Mutual TLS (mTLS [RFC8705]) authentication
* API Keys via HTTP headers (e.g., X-API-Key)
* Capability-based access control
* Delegation chains

For non-HTTP transports (e.g., WebSocket, Matrix), agents SHOULD leverage native authentication mechanisms, such as WebSocket protocol-level authentication tokens or Matrix homeserver authentication flows. Agents MUST clearly document supported security mechanisms per transport binding.

If OAuth 2.0 is used, authorization flows MUST conform to [RFC6749]; bearer usage per [RFC6750].

When using Decentralized Identifiers [DID-CORE] as authority, agent descriptors MAY be cryptographically signed. Clients SHOULD verify such signatures against the corresponding DID Document.

For agent-to-agent delegation, agents SHOULD include delegation metadata (e.g., X-Delegation-Chain) that identifies prior actors. These chains SHOULD be signed or verifiable via claims (e.g., using JWT, Verifiable Credentials, or DID-linked proofs).

Resolvers SHOULD use HTTPS with certificate validation and SHOULD validate integrity using ETag/Last-Modified for caching; if signature metadata is present, SHOULD verify signature per the descriptor’s signature scheme.

Agents MUST minimize data retention and expose revoke/delete interfaces for user data.

Privacy recommendations:

Agents SHOULD adhere to privacy best practices, including:

* Data minimization (collect only necessary data)

* Explicit consent and revocation mechanisms

* Clear logging/audit trails

* Ethical AI guidelines, including bias detection and fairness assessments as they evolve

10.1. Compliance and Regulatory Considerations

Implementers SHOULD ensure compliance with relevant legal frameworks (e.g., GDPR, CCPA) of the jurisdictions where the agent is hosted. Agents processing sensitive data SHOULD provide audit trails and explicit consent mechanisms clearly documented in capability descriptors.

11. Extensibility

The protocol supports extension via:

* Namespaced capability vocabularies
* Alternate transport bindings
* Extended agent descriptors
* Optional orchestration layers (task graphs, workflows)

Extension proposals SHOULD be documented clearly, and ideally reviewed through established processes such as community forums, dedicated working groups, or public registries to ensure transparency and interoperability.

12. IANA Considerations

This document requests the registration of the agent URI scheme in the IANA "Uniform Resource Identifier (URI) Schemes" registry.

12.1. URI Scheme Registration Template

* *Scheme Name*: agent
* *Status*: Provisional
* *Applications/Protocols That Use This Scheme*: The agent URI scheme identifies and invokes autonomous or semi-autonomous software agents across systems. It provides transport-agnostic addressing layer supporting discovery, invocation and orchestration. The scheme is compatible with existing schemes such as https, did and web+ schemes where appropriate.
* *Contact*: Yaswanth Narvaneni <yaswanth+ietf@gmail.com>
* *Change Controller*: The author or a relevant standards body such as the IETF if adopted.
* *References*: This document (Internet-Draft): _agent:// Protocol -- A URI-Based Framework for Interoperable Agents_ [RFC3986] - Uniform Resource Identifier (URI): Generic Syntax [RFC7595] - Guidelines and Registration Procedures for URI Schemes
* *URI Syntax*: The general form of an agent URI is:
* *Related Registrations:*

- Well-Known URIs (Section 12.2): /.well-known/agent.json, /.well-known/agents.json

- Media Type (Section 12.3): application/agent+json

agent:[+<protocol>]://<authority>/<path>[?<query>][#<fragment>]

Where: - authority is typically a domain name or Decentralized Identifier (DID) - path is an opaque agent-specific capability or namespace - query includes serialized key-value parameters - fragment MAY reference a sub-capability or context - The optional +<protocol> segment indicates an explicit transport binding (e.g., agent+https://)

Detailed ABNF is specified in Section 4.2 of this document.

* *Security Considerations*: The agent scheme does not introduce new transport-layer vulnerabilities but inherits risks from underlying protocols such as HTTP, WebSocket, or local execution environments. Implementers should apply standard authentication and authorization measures, such as OAuth2, JWTs, or mutual TLS. See Section 10 for security and privacy guidance.

12.2. Well-Known URI Registrations

This document registers the following Well-Known URIs under the "Well-Known URIs" registry established by RFC 8615:

1. *Name:* agent.json
   Purpose: Provides the self-describing metadata document (agent.json) for a single network-addressable agent.
   Expected Content: JSON object conforming to application/agent+json.
   Reference: This document.
   Security Considerations: Accessible only via HTTPS. Agents SHOULD sign or ETag-pin the descriptor.

2. *Name:* agents.json
   Purpose: Provides a registry mapping of agent names to descriptor URLs for multi-agent domains.
   Expected Content: JSON object mapping agent identifiers to agent.json URLs.
   Reference: This document.
   Security Considerations: Same as above.

12.3. Media Type Registration for application/agent+json

Type name: application
Subtype name: agent+json
Required parameters: none
Optional parameters: profile (for JSON-LD contexts)
Encoding considerations: 8bit; uses UTF-8 encoded JSON
Security considerations: Carries metadata that can affect network routing and authorization; publishers SHOULD serve only over HTTPS and validate signatures or ETags.
Interoperability considerations: Compatible with JSON-LD 1.1 and plain JSON processors.
Published specification: This document
Applications that use this media type: Agent resolvers and runtimes using the agent:// protocol.
Additional information: None
Person & email address to contact for further information: Yaswanth Narvaneni <yaswanth+ietf@gmail.com>
Intended usage: COMMON
Author/Change controller: IETF if standardized; author for independent submissions.

The profile parameter usage follows the concept in RFC 6906 [RFC6906] (Profiles), and media type registration procedures follow RFC 6838 [RFC6838].

13. Appendix A. Example Agent Descriptor

Following is an example of agent.json.

```
{
  "@context": "https://example.org/agent-context.jsonld",
  "name": "planner.example.com",
  "description": "Agent helps in researching & planning itineraries",
  "url": "agent://planner.example.com/",
  "provider": {
    "organization": "Example AI Org"
  },
  "documentationUrl": "https://planner.example.com/docs",
  "interactionModel": "agent2agent",
  "orchestration": "delegation",
  "envelopeSchemas": ["fipa-acl"],
  "version": "3.1.4",
  "supportedVersions": {
    "3.0.0": "/v3/",
    "2.1.2": "/olderversion/v2.1.2/",
    "1.0": "/version-1/"
  },
  "capabilities": [
    {
      "name": "gen-iti",
      "version": "2.1.5",
      "description": "Creates a travel itinerary for a given city.",
      "input": { "city": "string" },
      "output": { "itinerary": "array" },
      "isDeterministic": false,
      "expectedOutputVariability": "medium",
      "contentTypes": {
        "inputFormat": ["application/json", "application/ld+json"],
        "outputFormat": ["application/json"]
      }
    }
  ],
  "authentication": {
    "schemes": ["OAuth2"]
  },
  "skills": [
    {
        "id": "agent-skill-1",
        "name": "research-location"
    }
  ]
}
```

Figure 7: Example Agent Descriptor in JSON-LD

A JSON-LD context is added to support semantic querying and graph-based processing.

14. Appendix B. Use Cases

* Composing workflows with agents from different vendors
* Enabling discovery and invocation in agent marketplaces
* Facilitating human-in-the-loop workflows with agent transparency
* Building knowledge-based agents that invoke retrieval agents
* Real-time collaboration among specialized agents
* Browser-to-local-agent delegation for privileged operations and desktop automation
* Consistent addressing for agents across network boundaries and security contexts

15. Appendix C. Reference Implementation

A reference implementation of the agent:// protocol is available to guide implementers, demonstrating the following functionalities:

* URI parsing and resolution (agent.json, .well-known endpoints)
* Transport bindings including HTTPS, WebSocket, Matrix, and Local IPC
* Capability descriptor discovery, caching, and semantic processing
* Orchestration and delegation chaining examples
* Error handling, payload negotiation, and versioning patterns
* Security examples covering OAuth2, JWT, and mutual TLS (mTLS)

The implementation is open-source and maintained at:

[AGENT-URI-REPO]

Implementers are encouraged to use this as a starting point or reference during their implementation efforts.

Acknowledgements

This draft reflects observations and aspirations drawn from emerging agent ecosystems. It builds on publicly available research, community discussions, and early experimentation with agent-oriented protocols. It is intended as a foundation for future refinement and collaboration.

References

Normative References

[DID-CORE] Sporny, M., Longley, D., Sabadello, M., Reed, D., Steele, O., and C. Allen, "Decentralized Identifiers (DIDs) v1.0", W3C Recommendation, July 2022, <https://www.w3.org/TR/did-core/>.

[JSON-LD11] Sporny, M., Longley, D., Kellogg, G., Lanthaler, M., Champin, P., and N. Lindstrom, "JSON-LD 1.1: A JSON-based Serialization for Linked Data", W3C Recommendation, July 2020, <https://www.w3.org/TR/json-ld11/>.

[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>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/rfc/rfc3986>.

[RFC6570] Gregorio, J., Fielding, R., Hadley, M., Nottingham, M., and D. Orchard, "URI Template", RFC 6570, DOI 10.17487/RFC6570, March 2012, <https://www.rfc-editor.org/rfc/rfc6570>.

[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>.

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <https://www.rfc-editor.org/rfc/rfc6750>.

[RFC7519] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token (JWT)", RFC 7519, DOI 10.17487/RFC7519, May 2015, <https://www.rfc-editor.org/rfc/rfc7519>.

[RFC7595] Thaler, D., Ed., Hansen, T., and T. Hardie, "Guidelines and Registration Procedures for URI Schemes", BCP 35, RFC 7595, DOI 10.17487/RFC7595, June 2015, <https://www.rfc-editor.org/rfc/rfc7595>.

[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>.

[RFC8705] Campbell, B., Bradley, J., Sakimura, N., and T. Lodderstedt, "OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens", RFC 8705, DOI 10.17487/RFC8705, February 2020, <https://www.rfc-editor.org/rfc/rfc8705>.

[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>.

[RFC9457] Nottingham, M., Wilde, E., and S. Dalal, "Problem Details for HTTP APIs", RFC 9457, DOI 10.17487/RFC9457, July 2023, <https://www.rfc-editor.org/rfc/rfc9457>.

Informative References

[AGENT-URI-REPO]
Narvaneni, Y., "Agent URI Protocol Reference Implementation", 2025, <https://github.com/agent-uri/agent-uri>.

[Agent2Agent]
Google LLC, "Agent2Agent Protocol", April 2025, <https://github.com/google/A2A>.

[AgentCard]
Google LLC, "Agent Card Schema from Agent2Agent Protocol", April 2025, <https://github.com/google/A2A/blob/main/specification/json/a2a.json>.

[AutoGen] Microsoft Research, "AutoGen: Enabling LLM Applications with Multi-Agent Conversations", 2024, <https://microsoft.github.io/autogen/>.

[FIPA-ACL] Foundation for Intelligent Physical Agents, "FIPA ACL Message Structure Specification", 2002, <http://www.fipa.org/specs/fipa00061/SC00061G.html>.

[FIPA-CNP] Foundation for Intelligent Physical Agents, "FIPA Contract Net Interaction Protocol Specification", 2002, <http://www.fipa.org/specs/fipa00029/SC00029H.html>.

[GraphQL] GraphQL Foundation, "GraphQL: A Query Language for APIs", October 2021, <https://spec.graphql.org/October2021/>.

[JSON-RPC] JSON-RPC Working Group, "JSON-RPC 2.0 Specification", 4 January 2013, <https://www.jsonrpc.org/specification>.

[LangChain] LangChain Team, "LangChain Documentation", 2024, <https://python.langchain.com/v0.3/docs/>.

[Matrix] The Matrix org Foundation, "Matrix Specification v1.14", 2014, <https://spec.matrix.org/>.

[MCP] Anthropic PBC, "Model Context Protocol (MCP)", March 2025, <https://modelcontextprotocol.io/specification/>.

[OpenAPI] OpenAPI Initiative, "OpenAPI Specification v3.1.0", October 2024, <https://spec.openapis.org/oas/latest.html>.

[RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <https://www.rfc-editor.org/rfc/rfc6838>.

[RFC6906] Wilde, E., "The ’profile’ Link Relation Type", RFC 6906, DOI 10.17487/RFC6906, March 2013, <https://www.rfc-editor.org/rfc/rfc6906>.

[SemanticKernel] Microsoft, "Semantic Kernel SDK", 2024, <https://github.com/microsoft/semantic-kernel>.

[SemVer] Preston-Werner, T., "Semantic Versioning 2.0.0", 2013, <https://semver.org/>.

Author’s Address

Yaswanth Narvaneni
Independent Researcher
London
United Kingdom
Email: yaswanth+ietf@gmail.com