Independent Submission
Internet-Draft
Intended status: Experimental
Expires: 20 October 2026

Y. Narvaneni
S. P. Ravi
Independent Researcher

18 April 2026

The agent:// Protocol -- A URI-Based Framework for Interoperable Agents
draft-narvaneni-agent-uri-03

Abstract

This document defines the agent:// protocol, a URI-based addressing scheme for identifying, discovering, and invoking autonomous and semi-autonomous software agents. The agent:// scheme provides a semantic layer that signals "this resource is an agent" and enables agent-specific discovery via standardized descriptors. It introduces a layered architecture with four conformance levels, allowing implementations to adopt minimal addressing (Level 0) or full descriptor-based discovery with authentication and versioning (Level 3). The protocol complements existing agent communication protocols such as Agent2Agent (A2A), Model Context Protocol (MCP), and Agent Communication Protocol (ACP) by providing the addressing and discovery layer they lack.

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 20 October 2026.

Copyright Notice

Table of Contents

1. Introduction

The agent ecosystem is fragmented. Protocols like [Agent2Agent] define how agents communicate. Model Context Protocol [MCP] defines how models access tools. [FIPA-ACL] and Contract Net Protocol [FIPA-CNP] define negotiation semantics. Frameworks like [LangChain], [AutoGen], and [SemanticKernel] provide runtime environments. Yet none of these define a universal way to *address* an agent -- to say "this is an agent, here is how to find it, here is what it can do."

The agent:// protocol fills this gap. It is an *addressing scheme* that identifies a resource as an agent and enables agent-specific discovery and metadata. DNS, HTTPS, and existing transport protocols remain as-is -- agent:// adds a semantic layer on top that signals agent identity and unlocks standardized discovery.

In a single sentence: *agent:// is to A2A, MCP, and ACP what https:// is to REST, GraphQL, and gRPC -- the URI scheme, separate from the application protocol.* This document defines addressing and discovery only. Communication semantics are delegated to the chosen application protocol.

The following table illustrates how agent:// complements rather than competes with existing protocols:

Concern	agent://	HTTPS + OpenAPI	A2A	MCP
Agent addressing and identity	Yes	No (URL only)	Partial	No
Transport-agnostic discovery	Yes	No	No	No
Capability description	Yes	Yes (OpenAPI)	Yes (AgentCard)	Yes (tools)
Communication semantics	No (delegates)	Yes	Yes	Yes
Local agent invocation	Yes	No	No	Yes

Table 1: Comparison of agent:// with Related 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

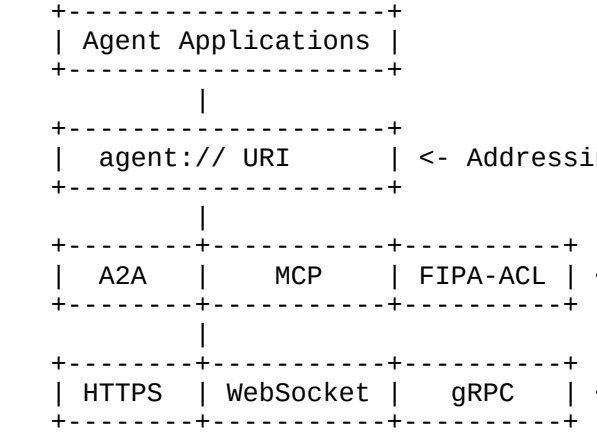

Figure 1: Agent Protocol Stack Architecture

This document defines four conformance levels (Section 3.1) to support incremental adoption:

* *Level 0*: Use agent+https:// as a direct invocation scheme (zero infrastructure)

* *Level 1*: Publish /.well-known/agents.json for discovery

* *Level 2*: Support resolution, caching, and multiple transports

* *Level 3*: Full capability descriptors with authentication, versioning, and composition

1.1. Quick Start

The simplest adoption path requires no infrastructure changes:

Level 0 -- Direct Invocation:

agent+https://example.com/my-agent?message=hello

This is semantically equivalent to an HTTPS request to https://example.com/my-agent?message=hello, but identifies the target as an agent.

Level 1 -- Discoverable Agent:

Publish a /.well-known/agents.json file on your domain:

```
{
  "agents": {
    "my-agent": "https://example.com/my-agent/agent.json"
  }
}
```

Clients can now resolve agent://example.com/my-agent by fetching this registry and following the descriptor URL.

A reference implementation of the agent:// protocol is available at agent-uri reference implementation [AGENT-URI-REPO] to demonstrate URI parsing, resolution, transport bindings, and descriptor handling.

2. Terminology

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

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

* *Agent Registry*: A domain-level directory (/.well-known/agents.json) mapping agent names to descriptor URLs.

* *Skill*: A discrete, named function or behavior offered by an agent, declared in the descriptor under the skills field with at minimum a name and description, plus optional input/output schema and metadata. This term aligns with the [AgentCard] skills concept. "Capability" is used as the general English term where context requires it.

* *Conformance Level*: One of four progressive levels of protocol adoption (Level 0 through Level 3), as defined in Section 3.1.

* *Delegation Chain*: An ordered list of agent identifiers representing the chain of agents that have delegated a task, expressed as signed JWT claims.

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

* *Resolution Endpoint*: The /.well-known/agents.json URL used for agent discovery.

* *Resolver*: A service or mechanism that maps an agent URI to a network endpoint or descriptor.

* *Transport Binding*: A mapping from the agent+<protocol>:// scheme to a concrete communication mechanism (e.g., HTTPS, WebSocket, gRPC, local IPC).

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.

3. Protocol Scope and Layering

The agent:// protocol is an addressing and discovery scheme. It defines how to identify, locate, and describe agents. It does *not* define communication semantics -- how agents exchange messages, manage tasks, or handle state. Those concerns belong to the communication protocol layer (e.g., [Agent2Agent], [MCP], [FIPA-ACL]) operating above agent://.

The protocol is designed as a layered framework:

Layer	Purpose	Conformance Level
URI Scheme	Unique addressing and agent identity	Level 0+
Transport Binding	Mechanism for invocation (e.g., HTTPS, WSS, gRPC, local IPC)	Level 0+
Agent Descriptor	Self-describing agent interface and skills	Level 1+
Resolution Framework	Maps agent URIs to endpoints via /.well-known/agents.json	Level 2+
Application Semantics	Shared vocabulary for skill naming and versioning	Level 3

Table 2: Protocol Layering Structure

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

3.1. Conformance Levels

This specification defines four conformance levels to support incremental adoption:

Level	Name	Requirements
0	Direct Invocation	Support agent+<protocol>:// URI parsing and direct transport invocation
1	Discoverable	Level 0 + publish /.well-known/agents.json with agent entries pointing to descriptor URLs
2	Resolvable	Level 1 + support the resolution algorithm (Section 5), caching, and multiple transports
3	Full	Level 2 + descriptors with full skills, authentication, optional signing, and version aliasing

Table 3: Conformance Levels

Implementations SHOULD declare their conformance level in documentation. Clients SHOULD gracefully degrade when interacting with agents at a lower conformance level.

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+grpc://inference.example.com/model/predict
* agent+local://examplelocalagent
* agent://did%3Aweb%3Aexample.com%3Aagent%3Aresearcher/get-article?doi=10.1234/example (canonical, ABNF-conformant)

* agent://did:web:example.com:agent:researcher/get-article?doi=10.1234/example (convenience form; see Section 4.3)

To resolve agent://<authority>/<path>?<query>: 1. Fetch https://<authority>/.well-known/agents.json 2. Locate <path> mapping -> agent descriptor URL 3. Fetch descriptor 4. Extract transport.endpoint or transport metadata 5. Invoke using indicated method or default (GET for read-only, POST for state-changing) 6. If agents.json is not found and an explicit transport binding is present -> invoke directly via that transport

4.1. Components

* *Authority*: Uniquely identifies the agent or agent namespace (e.g., DNS hostname or Decentralized Identifier [DID-CORE]). DNS authorities follow RFC 3986 syntax directly. DID authorities require special handling because DIDs contain unreserved colons that conflict with the RFC 3986 host [":" port] rule; see Section 4.3.

* *Path*: Specifies the agent and optionally a skill. For agents exposing multiple skills, the path SHOULD follow the convention /<agent-name>/<skill-id>. For single-skill agents, the path MAY be the agent name alone.

* *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 a sub-skill.

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

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

Resolvers MUST preserve the port component during resolution. For example, agent://example.com:9090/my-agent MUST resolve against https://example.com:9090/, not https://example.com/.

4.2. ABNF for agent:// URI

The ABNF notation follows RFC 5234 [RFC5234]. The core rules ALPHA and DIGIT are defined in RFC 5234 [RFC5234] Appendix B.1. The productions authority, path-abempty, query, fragment, and their sub-rules (userinfo, host, port, segment, pchar, unreserved, pct-encoded, sub-delims) are imported from RFC 3986 [RFC3986] Section 3.

```
agent-uri      = "agent" ["+" protocol] "://" authority path-abempty
                 [ "?" query ] [ "#" fragment ]

protocol       = ALPHA *( ALPHA / DIGIT / "-" )
                 ; Registered transport binding identifier.
                 ; MUST start with a letter per URI scheme conventions.
```

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

The agent:// scheme always requires the :// delimiter and an authority component. The authority component MUST NOT be empty. The hier-part alternatives path-absolute, path-rootless, and path-empty from RFC 3986 [RFC3986] are not used; only the "//" authority path-abempty form is valid for agent:// URIs.

4.3. DID-based Authority

Decentralized Identifiers (DIDs) [DID-CORE] contain colons as internal separators, which conflict with the RFC 3986 host [":" port] rule in the authority production. Implementations MUST handle this as follows:

Canonical (ABNF-conformant) form. The DID MUST be percent-encoded in the authority component. Each : within the DID is replaced with %3A:

agent://did%3Aweb%3Aexample.com%3Aagent%3Aresearcher/get-article

Standard URI parsers will correctly treat the encoded string as a single reg-name host.

Convenience form (informative). Many existing tools and human-authored documents use the unencoded DID in the authority:

agent://did:web:example.com:agent:researcher/get-article

Implementations MAY accept this form on a best-effort basis by detecting the did: prefix and treating the entire authority as an opaque DID identifier (ignoring the RFC 3986 port rule). However, the unencoded form is not strictly ABNF-conformant and MAY be rejected by strict parsers. Serializers SHOULD emit the canonical percent-encoded form.

Resolution of DID authorities. When the authority is a DID, Step 1 of the resolution algorithm (Section 5.1) does NOT apply directly, because a DID is not a DNS host. Resolvers MUST instead perform DID resolution per DID Core [DID-CORE] to obtain the DID Document, then extract a service endpoint of type AgentDescriptor (or an equivalent type). If such an endpoint is present, its URL is treated as the descriptor URL (Step 3 onward). If no suitable service endpoint is found, resolution fails.

Not every DID method resolves to a DNS-addressable endpoint (e.g., did:key, did:pkh). Such DIDs are valid agent:// authorities only when coupled with an explicit transport binding (e.g., agent+local://did%3Akey%3Az6Mk.../capability) or when paired with out-of-band resolution infrastructure.

5. Resolution Framework

Agent discovery uses a single well-known endpoint: /.well-known/agents.json. This file serves as the agent registry for a domain, mapping agent names to their descriptor URLs. Both single-agent and multi-agent domains use the same format -- a single-agent domain simply has one entry.

5.1. Resolution Algorithm

To resolve agent://<authority>/<path>:

1. Construct the registry URL: https://<authority>/.well-known/agents.json (preserving port if present in the authority).

2. Fetch the registry via HTTPS GET.

3. Extract the first path segment as the agent name. Look up the agent name in the agents mapping to obtain the descriptor URL.

4. Fetch the agent descriptor from that URL.

5. Extract transport and endpoint metadata from the descriptor.

6. Invoke the agent using the indicated transport, or default to HTTPS with POST (if parameters present) or GET (otherwise).

If agents.json is not found (HTTP 404) and the URI has an explicit transport binding (e.g., agent+https://), clients MAY invoke directly via that transport without a descriptor.

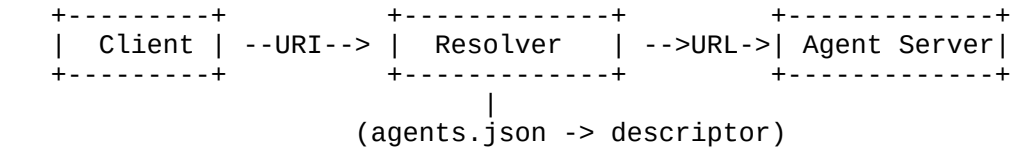

Figure 4: Agent URI Resolution Process

Example agents.json:

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

A single-agent domain simply has one entry in the registry.

Individual agent descriptors MUST conform to the normative minimum schema in Section 7. Additional application-layer metadata MAY be carried via the extension patterns of Section 7.2.

5.2. Resolver Security

Resolvers MUST restrict fetches to HTTPS schemes. Resolvers MUST NOT follow HTTP redirects to, or resolve agent URIs whose authorities resolve to, private, loopback, link-local, or otherwise non-routable IP address ranges:

* 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16 (IPv4 private)
* 127.0.0.0/8 (IPv4 loopback)
* 169.254.0.0/16 (IPv4 link-local, including cloud metadata endpoints)
* 0.0.0.0/8 (IPv4 "this network")
* ::1/128 (IPv6 loopback)
* fc00::/7 (IPv6 unique local)
* fe80::/10 (IPv6 link-local)

* IPv4-mapped IPv6 addresses (::ffff:0:0/96) MUST be unwrapped and re-checked against the IPv4 ranges above.

These checks MUST be applied at *every* stage of resolution:

1. When resolving the authority of the incoming agent:// URI.

2. When the descriptor URL extracted from /.well-known/agents.json is resolved (the agents.json content may be attacker-influenced).

3. When following any HTTP redirect during descriptor or registry fetch.

Resolvers SHOULD verify TLS certificates and MAY require signed descriptors. To mitigate DNS rebinding attacks, resolvers SHOULD pin the resolved IP address for the duration of a request or implement equivalent controls.

5.3. Caching

Resolvers that cache descriptors or registry responses MUST implement HTTP cache semantics per RFC 9111 [RFC9111]. Agents at Conformance Level 2 or above SHOULD include Cache-Control headers in descriptor responses and SHOULD include ETag or Last-Modified headers to enable conditional requests (If-None-Match, If-Modified-Since).

Resolvers SHOULD also apply negative caching to 404 responses from /.well-known/agents.json with a short TTL to avoid unbounded upstream traffic when clients repeatedly resolve non-existent authorities.

5.4. Relationship to WebFinger and Other Discovery Protocols

WebFinger [RFC7033] (/.well-known/webfinger) is the established IETF mechanism for account-centric resource discovery via a JRD. The /.well-known/agents.json registry defined by this document is similar in spirit but differs in important ways:

* WebFinger is account-scoped (acct:user@host) and returns a JRD with typed Link relations.

* agents.json is an authority-scoped directory of named agents resolving to descriptor URLs, optimized for the common case of a domain hosting a known set of agents.

Implementations MAY additionally support WebFinger-style resource-scoped queries, DID resolution (see Section 4.3), or proprietary resolvers. These alternative mechanisms are out of scope for this specification.

Positioning relative to emerging IETF work. Several related IETF efforts are in progress at the time of this writing: discussions around the "Discovery of Agents, Workloads, and Named Entities" (DAWN) problem space, the proposed AI-agent-protocols working group (AIPROTO), and various Agent Name Service / Agent Identity Protocol drafts. This specification addresses the URI scheme and authority-scoped discovery layer only. It is intentionally neutral to agent _naming_ and _identity_ semantics: a named entity defined by future IETF work (for example, via DAWN) can be used as an agent:// authority, and identity proofs (e.g., as proposed by Agent Identity Protocol drafts) can be layered via the descriptor's authentication and DID authority mechanisms. The resolution algorithm is agnostic to authority type.

A dynamic agent registry protocol for runtime registration and deregistration is planned as a companion specification (see Section 11.2).

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
gRPC	agent+grpc://	Protocol Buffers, bidi-stream	High-performance inter-agent calls
MQTT	agent+mqtt://	Pub/sub topics	IoT and edge agent messaging
Local	agent+local://	IPC/broker call	Runtime-registered local agents
Unix Socket	agent+unix://	Unix domain socket	IPC for co-located agents

Table 4: Transport Binding Formats

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

New transport bindings MAY be registered via the IANA registry defined in Section 12. Each registered binding MUST specify connection establishment, message framing, and error signaling conventions.

6.2. Transport-Specific Conventions

6.2.1. HTTPS (agent+https://)

Connection is established via standard HTTPS. Requests use GET (no parameters) or POST (with application/json body). Responses use standard HTTP status codes. Errors follow RFC 9457 [RFC9457] problem details.

6.2.2. WebSocket Secure (agent+wss://)

Connection is established via the WebSocket handshake per RFC 6455 [RFC6455]. Messages are framed as newline-delimited JSON (NDJSON). Errors are sent as JSON objects with type, title, detail, and status fields within the stream.

6.2.3. gRPC (agent+grpc://)

Connection is established via HTTP/2 with Protocol Buffer serialization. Agents SHOULD publish .proto service definitions alongside their descriptors. Supports unary, server-streaming, client-streaming, and bidirectional streaming patterns. Errors use standard [gRPC] status codes.

6.2.4. MQTT (agent+mqtt://)

Connection is established via MQTT 5.0 [MQTT] broker. The agent name maps to a topic prefix (e.g., agents/<agent-name>/invoke). Request/response uses MQTT 5.0 request-response pattern with correlation data. Messages are JSON-encoded. This binding is designed for constrained IoT and edge environments.

6.2.5. Local (agent+local://)

Local agents are accessed using:

agent+local://<agent-name>

agent+local:// requires explicit user consent and origin binding. Implementations MUST prompt the user before first invocation of a local agent. The consent dialog MUST identify the requesting origin and the target agent.

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 but common patterns include:

* Custom protocol handler registration in the operating system

* Service worker mediation in browser environments

* Browser extension APIs bridging web content to local processes

The agent+local:// scheme addresses the 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. Security considerations are discussed in Section 10.

Local discovery. A normative local-agent discovery mechanism (the local-runtime equivalent of /.well-known/agents.json) is intentionally out of scope for this document. Runtimes MAY expose a list of available local agents via an implementation-specific channel (e.g., an OS service registry, a browser extension API, or a UNIX domain socket at a well-known path). A standardized local discovery protocol is planned as a companion specification (see Section 11.2).

6.2.6. Unix Domain Socket (agent+unix://)

The agent+unix:// binding targets co-located agents addressable via a Unix domain socket. The authority component carries the agent name; the socket path is conveyed in the descriptor’s transport.unix value (e.g., /var/run/agent/planner.sock) and is NOT encoded into the URI. This keeps URIs portable across deployments.

Connection establishment: the client connects to the socket path referenced by the descriptor. Message framing: JSON messages delimited by newline (NDJSON) by default; implementations MAY negotiate alternative framing via the interactionModel selected. Error signaling: errors are JSON objects with the RFC 9457 problem-detail fields (type, title, detail, status) transported within the same NDJSON framing.

Access control is enforced by filesystem permissions on the socket path. agent+unix:// SHOULD NOT be exposed to untrusted local code without additional OS-level sandboxing.

6.3. Default Fallback Behavior

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

1. Fetch /.well-known/agents.json from the authority
2. Look up the agent and retrieve its descriptor
3. Use the transport or endpoint hints from the descriptor

If the registry is not 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.

6.4. 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://
High-performance inter-agent calls	agent+grpc://
Real-time streaming responses	agent+wss://
IoT / edge agent	agent+mqtt://
Local runtime agent	agent+local://
Dynamic/multi-transport agents	agent:// with agents.json

Table 5: Recommended Bindings for Common Use Cases

7. Descriptor Framework

Agents SHOULD expose a descriptor document (typically agent.json) at a URL referenced from the domain's /.well-known/agents.json registry.

The field naming in this specification aligns with [AgentCard]. An agent's declared functions are carried in a skills array. Where the English word "capability" is used in prose, it refers to the general concept; the JSON field name is always skills.

7.1. Normative Descriptor Schema

Agent descriptors MUST conform to the following minimum schema. A complete JSON Schema is provided in Section 16.

7.1.1. Required Fields

An agent descriptor MUST include:

* name (string): The agent's identifier.
* version (string): The agent's version, following [SemVer].
* skills (array): One or more skill objects, each with at minimum id (string), name (string), and description (string).

7.1.2. Recommended Fields

An agent descriptor SHOULD include:

* description (string): Human-readable description of the agent.

* url (string, URI): The agent's canonical agent:// URI.

* transport (object): Transport metadata. The object MUST contain at least one of the following keys: endpoint (a default URI for the agent), or one of the per-transport keys https, wss, grpc, mqtt, local, unix. Each value is a URI appropriate to that transport. The endpoint key carries the default transport when an agent:// URI has no explicit +protocol binding.

* authentication (object): Authentication metadata; see Section 10.1.

* provider (object): Organization information with organization (string) and optional url (string).

* conformanceLevel (integer): The conformance level (0-3) implemented by this agent, as defined in Section 3.1. Enables machine-readable feature detection.

7.1.3. Optional Fields

An agent descriptor MAY include:

* status (string): One of "active", "deprecated", "experimental". Defaults to "active" if omitted. For deprecation with a sunset date, also serve the Sunset HTTP response header per RFC 8594 [RFC8594].

* supportedVersions (object): OPTIONAL map of older version strings to their endpoint paths, useful for static discovery of historical versions. Live version signaling on the wire SHOULD use the HTTP Link response header with rel="predecessor-version"/rel="successor-version" and the Sunset header per RFC 8594 [RFC8594]; supportedVersions is purely informational and MUST NOT be the sole mechanism for deprecation signaling.

* documentationUrl (string, URI): Human-readable documentation URL for the agent.

* environment (string): Deployment environment hint. Values aligned with the OpenTelemetry semantic conventions [OTEL-GENAI] deployment.environment attribute are RECOMMENDED (e.g., "production", "staging", "development", "preview", "sandbox").

* interactionModel (array of strings): One or more communication protocol identifiers the agent speaks. Registered values include agent2agent, mcp, fipa-acl, openapi. Additional values MUST be registered via the IANA registry (Section 12.5). A client negotiates with the agent to select one.

7.1.4. Skill Fields

Each entry in the skills array MUST include id, name, and description. Each entry MAY additionally include:

* version (string): Skill-specific version (SemVer).

* tags (array of strings): Classification tags for discovery.

* input (object): JSON Schema describing the input format.

* output (object): JSON Schema describing the output format.

* contentTypes (object): With accepts and produces arrays of MIME types (mirroring HTTP Accept / Content-Type semantics).

* streaming (boolean): Whether the skill supports streaming responses. A skill MAY indicate the streaming format via the streamingFormat field with values such as "sse", "ndjson", or "grpc-stream".

* idempotent (boolean): Whether the skill is safe to retry without side effects. When set, clients SHOULD use the Idempotency-Key HTTP header per draft-ietf-httpapi-idempotency-key-header [I-D.ietf-httpapi-idempotency-key-header].

* status (string): Skill lifecycle status ("active", "deprecated", "experimental"), independent of the agent-level status.

* authentication (object): Per-skill authentication requirements overriding the agent-level authentication.

* depends (array of objects): Declarative references to other agents this skill may invoke. Each entry is an object {uri, relation?, versionConstraint?}. The uri is a canonical agent:// URI; relation is a typed relationship (e.g., "invokes", "enriches"); versionConstraint is a SemVer range string. The field is purely declarative -- clients MUST NOT pre-invoke or pre-resolve dependencies based solely on this field. Agents SHOULD publish acyclic dependency graphs. Tooling that transitively resolves depends chains (e.g., topology analyzers, cost estimators) MUST detect and reject cycles, and SHOULD bound traversal depth (a limit of 16 is RECOMMENDED).

Application-level behavioral metadata (quantitative latency, pricing, underlying model information, determinism/variability hints, multimodal I/O details, commerce terms, performance SLAs) is intentionally out of scope. See Section 7.2 for the recommended way to carry such metadata via JSON-LD @context referencing external vocabularies such as schema.org (https://schema.org) and the OpenTelemetry semantic conventions for Generative AI [OTEL-GENAI].

7.2. Extension Fields

The agent:// protocol intentionally limits its normative vocabulary to fields required for agent addressing, discovery, capability registration, and transport-level concerns. Commerce, performance analytics, multimodal content negotiation, organizational metadata, and other application-level concerns are *out of scope*. These are well-served by existing vocabularies.

Descriptors MAY carry extension fields from external vocabularies. Two patterns are RECOMMENDED:

7.2.1. Pattern 1: JSON-LD Context Layering

Descriptors MAY include a top-level @context field referencing one or more published JSON-LD contexts. Extension fields MAY then use terms defined in those contexts. Implementations that do not understand a given @context MUST ignore unrecognized fields (permitted by additionalProperties: true on the top-level schema).

```
{
  "@context": [
    "https://agent-uri.org/context/v1",
    "https://schema.org"
  ],
  "name": "planner.example.com",
  "version": "3.1.4",
  "skills": [{ "id": "gen-iti", "name": "Generate Itinerary", "description": "..." }],
  "offers": {
    "@type": "Offer",
    "priceSpecification": {
      "@type": "UnitPriceSpecification",
      "price": 0.01,
      "priceCurrency": "USD",
      "unitText": "per invocation"
    }
  }
}
```

Figure 5: Extension via schema.org

Pricing, product information, organization details, ratings, reviews, and similar commerce or metadata fields SHOULD use schema.org terms rather than agent://-specific fields.

7.2.2. Pattern 2: AgentCard Co-existence

This specification’s skills array is designed to align with [AgentCard]’s skills. Implementations that bridge to [Agent2Agent] SHOULD carry additional AgentCard-specific fields (such as defaultInputModes, defaultOutputModes) under an x-a2a- prefix rather than at the top level, to preserve clear namespace boundaries.

7.2.3. Pattern 3: Vendor Namespaces

Vendor-specific fields SHOULD be namespaced under a x-<vendor> prefix (note: the X- prefix deprecation in RFC 6648 [RFC6648] applies to HTTP headers, not to JSON field names; the x- convention is widely used in OpenAPI, schema.org, and similar JSON schemas to signal vendor extensions). Example: "x-acme-latency-p95-ms": 120.

7.2.4. What agent:// does NOT define

The following are NOT normatively defined by this specification and MUST be layered via the extension patterns above when needed:

* Pricing, billing, commerce terms (use schema.org Offer / PriceSpecification)

* Quantitative latency or SLA metrics (use OpenTelemetry semantic conventions [OTEL-GENAI] or schema.org QuantitativeValue)

* Underlying model or framework identifiers (use schema.org SoftwareApplication or vendor-specific contexts)

* Human-oriented ratings, reviews, endorsements (use schema.org Review / AggregateRating)

* Service-level quality assurances (use domain-specific SLA vocabularies)

* User-interface metadata (icons, colors, preferred languages) (use AgentCard extensions or vendor namespaces)

This sharper scope is deliberate: agent:// is an addressing and discovery layer. Application semantics belong at higher layers.

7.3. Content Negotiation

Agent descriptors SHOULD include input/output schemas (e.g., JSON Schema) and MAY document content negotiation support via the contentTypes field on each skill. This allows clients to understand and negotiate payload encoding, enabling interoperability across ecosystems that use JSON, JSON-LD 1.1 [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] etc.).

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

7.4. Version Negotiation

Clients SHOULD specify the requested descriptor version using the standard HTTP content-negotiation profile parameter on the Accept header (RFC 6906 [RFC6906]):

Accept: application/agent+json; profile="https://agent-uri.org/profile/v3"

This is the canonical negotiation mechanism. Servers MAY additionally accept version specification via URI path segment (e.g., /v3/) or query parameter (e.g., ?version=3.1.4), but these are conveniences; the Accept; profile= form takes precedence when

present. If the Accept; profile= value conflicts with a path-segment or query version, servers MUST honor the Accept; profile= value and SHOULD return 406 Not Acceptable when they cannot satisfy it.

If no version is specified, servers SHOULD return the latest version and SHOULD indicate the selected version via Content-Type: application/agent+json; profile="...". Major version increments indicate breaking changes; clients SHOULD only auto-upgrade within the same major version.

Deprecation of older versions SHOULD use the Sunset HTTP response header (RFC 8594 [RFC8594]) to signal the retirement date, and SHOULD use Link: <...>; rel="successor-version" to point at replacements.

7.5. Descriptor Format

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

Descriptors MAY use the [AgentCard] schema (as defined by the Agent2Agent protocol) as one possible format. Any format other than AgentCard SHOULD be expressed in JSON-LD 1.1 [JSON-LD11] to enable semantic discovery.

JSON-LD 1.1 [JSON-LD11] support is an extension point for semantic interoperability. Descriptors MAY include a @context field referencing a published JSON-LD context document that maps descriptor fields to IRIs. When a context is provided, the context URL SHOULD be dereferenceable. A standard JSON-LD context for agent:// descriptors is planned as a companion specification. Implementations SHOULD NOT rely on JSON-LD processing for core resolution or invocation.

The /.well-known/agents.json registry enumerates all available agents under a domain. The individual descriptor files serve as the canonical source of truth for each agent.

8. Cross-Cutting Conventions

This section covers transport-level conventions that apply regardless of the communication protocol in use. Interaction patterns (request/response, streaming, deferred response, delegated invocation, asynchronous events) and their semantics -- message formats, task lifecycles, state machines -- are defined by the communication-layer protocol (e.g., [Agent2Agent], [MCP]), not by agent://.

8.1. Correlation and Session Metadata

For cross-agent correlation and stateful interactions, clients and agents SHOULD use established W3C and IETF conventions rather than bespoke headers:

* *Distributed tracing*: use the W3C Trace Context [W3C-TRACE-CONTEXT] traceparent and tracestate headers.

* *Correlation baggage* (session IDs, user IDs, and other propagated key-value data): use the W3C Baggage [W3C-BAGGAGE] header. For cross-agent correlation, the session.id and enduser.id keys align with the OpenTelemetry semantic conventions [OTEL-GENAI]. Application-level task identifiers MAY be carried under a vendor-prefixed key (e.g., example.task.id) or under an x- prefixed header by mutual agreement; this specification does not register a normative task-id key.

* *Idempotency*: when a skill's idempotent field is true, clients SHOULD include the Idempotency-Key header per draft-ietf-httpapi-idempotency-key-header [I-D.ietf-httpapi-idempotency-key-header] to enable safe retry.

* *Asynchronous invocation*: use the standard HTTP idiom of 202 Accepted + Location: <poll-url> per RFC 9110 [RFC9110] Section 15.3.3. Richer task lifecycles (cancellation, progress, state machines) are defined by the communication protocol in use.

Example:

```
GET /tasks/1234 HTTP/1.1
Host: planner.example.com
traceparent: 00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-01
baggage: session.id=abcde-12345
```

This specification does not register new correlation headers. Deployments that require application-specific headers (beyond traceparent/tracestate/baggage/Idempotency-Key) SHOULD use namespaced headers per implementer-specific registries.

8.2. Multi-Tenancy

Multi-tenant deployments SHOULD follow the pattern established by RFC 8414 [RFC8414] (path insertion of tenant components after the well-known suffix): each tenant exposes its own registry at /.well-known/agents.json/<tenant>. Path-prefix namespacing inside a single shared agents.json document (e.g., "tenant-a/agent-1": "...") is permitted but is NOT RECOMMENDED for large multi-tenant domains because the

entire registry must be fetched to locate one tenant’s entry.

9. Error Handling

Implementations SHOULD return errors using standard HTTP status codes along with structured JSON error responses conforming to RFC 9457 [RFC9457] ("Problem Details for HTTP APIs").

9.1. Agent Errors

Recommended HTTP status codes for agent errors:

Status Code	Meaning
400	Bad Request (e.g., invalid parameters)
401	Unauthorized
403	Forbidden
404	Skill or resource not found
406	Not Acceptable (content negotiation failure)
409	Conflict (e.g., state mismatch)
410	Gone (agent deprecated or decommissioned)
429	Too Many Requests (rate limiting)
500	Internal Server Error
503	Service Unavailable

Table 6: Recommended HTTP status codes

When returning 429 (Too Many Requests) or 503 (Service Unavailable), agents SHOULD follow the runtime rate-limit signaling conventions defined by draft-ietf-httpapi-ratelimit-headers [I-D.ietf-httpapi-ratelimit-headers] (the RateLimit and RateLimit-Policy structured-field headers) and SHOULD include a Retry-After header per RFC 9110 [RFC9110].

Static advertisement of rate-limit policy at the descriptor level is out of scope; implementations needing to advertise limits before invocation SHOULD do so via an extension field referencing the IETF rate-limit-headers structured-field syntax.

Example:

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

{
  "type": "https://example.com/errors/skill-not-found",
  "title": "Skill Not Found",
  "status": 404,
  "detail": "The requested skill 'gen-iti' was not found.",
  "instance": "/planner/gen-iti"
}
```

Figure 6: Example HTTP Error Response

9.2. Resolution Errors

Resolvers SHOULD distinguish the following error conditions:

* *DNS resolution failure*: The authority cannot be resolved to an IP address.

* *Registry not found*: /.well-known/agents.json returns HTTP 404.

* *Agent not found*: The agent name is not present in the registry.

* *Descriptor fetch failure*: The descriptor URL from the registry is unreachable or returns an error.

* *SSRF violation*: The resolved IP address falls within a private or loopback range.

Resolvers SHOULD surface these errors with sufficient detail for clients to distinguish between misconfiguration and genuine unavailability.

9.3. Non-HTTP Transport Errors

For non-HTTP transports (e.g., WebSockets, gRPC), agents SHOULD return structured errors using JSON structures with type, title, detail, and status fields, encapsulated within the transport's native message envelope. Where applicable, implementations SHOULD align with existing conventions such as:

* JSON-RPC error objects (code, message, data)

* gRPC status codes

* OpenAPI [OPENAPI-3.1] or REST error payloads

Application-level error vocabularies are out of scope for this specification. Clients SHOULD parse and utilize structured error responses produced by the communication-layer protocol in use.

10. Security Considerations

10.1. Authentication

Agents SHOULD reuse existing IETF authentication and authorization infrastructure rather than defining new mechanisms. Descriptors advertise their authentication requirements through either:

* *OAuth 2.0 Authorization Server Metadata* per RFC 8414 [RFC8414] -- the descriptor carries an authorizationServer URI pointing at the issuer; clients dereference <issuer>/.well-known/oauth-authorization-server to discover the token endpoint, supported grant types, scopes, and keys.

* *OAuth 2.0 Protected Resource Metadata* per RFC 9728 [RFC9728] -- the descriptor carries a protectedResourceMetadata URI that points at the agent's own protected-resource metadata document.

* *HTTP Authentication Schemes* per RFC 9110 [RFC9110] Section 11 -- the descriptor's schemes array lists values from the IANA HTTP Authentication Scheme Registry (Bearer, Mutual, Digest, ...).

Token usage follows RFC 6750 [RFC6750] for bearer tokens and RFC 8705 [RFC8705] for mutual-TLS-bound tokens. The authentication field on a descriptor has the following shape:

```
{
  "authentication": {
    "schemes": ["Bearer"],
    "authorizationServer": "https://auth.example.com",
    "protectedResourceMetadata": "https://agent.example.com/.well-known/oauth-protecte
d-resource",
    "jwksUri": "https://agent.example.com/.well-known/jwks.json"
  }
}
```

Figure 7: Authentication Descriptor Example

Either authorizationServer or protectedResourceMetadata (or both) SHOULD be present when OAuth 2.0 is in use. Clients MUST prefer the metadata documents over any fields embedded directly in the descriptor. For deployments that do not use OAuth and do not use DID-based authorities, the descriptor MAY directly carry jwksUri (URL of a JWK Set per RFC 7517 [RFC7517]) or an inline jwks object; this is the third permitted path for signature-verification key discovery (see Section 10.3). Implementations MAY additionally carry an OpenAPI 3.1 [OPENAPI-3.1] securitySchemes object under the x-openapi-security extension key for tooling that consumes OpenAPI natively.

For non-HTTP transports (e.g., WebSocket, gRPC), agents SHOULD leverage native authentication mechanisms and document them in the descriptor via the extension-field patterns of Section 7.2.

10.2. Delegation

For agent-to-agent delegation, this specification does not define a new mechanism. Implementations SHOULD use *OAuth 2.0 Token Exchange* per RFC 8693 [RFC8693], which already supports multi-hop delegation via the nested act (actor) claim. A single signed token carries the full delegation chain compactly and verifiably:

* Each delegating principal appears as a nested act claim inside the token.

* The token’s iss, sub, aud, iat, exp, nbf, jti claims follow RFC 7519 [RFC7519] semantics.

* JWT validation follows RFC 8725 [RFC8725] best current practice, including algorithm restriction and replay protection via jti.

* Scope restriction uses the scope claim per RFC 9068 [RFC9068].

This specification adds one recommendation on top of RFC 8693:

Monotonic scope narrowing. When act claims are nested, the scope of each inner (downstream) actor SHOULD be a subset of the scope of the outer (upstream) actor. Verifiers SHOULD reject tokens where a nested actor appears to carry broader scope than its delegator. This aligns with the attenuation principle of capability-based security.

Scope comparison semantics are deployment-defined: most deployments treat scope as a space-delimited bag of tokens and apply set-inclusion, but some deployments encode hierarchical scopes (e.g., billing.read.*) or use opaque scope identifiers that require out-of-band comparison logic. Verifiers MUST document their scope-comparison policy, and SHOULD reject tokens whose scope structure they cannot compare.

Delegation tokens SHOULD be carried in the Authorization header using the Bearer scheme per RFC 6750 [RFC6750]. No separate Delegation-Chain header is defined by this specification.

10.3. Descriptor Signing

Agent descriptors served over HTTP SHOULD be integrity-protected using *HTTP Message Signatures* per RFC 9421 [RFC9421]. The signature covers at minimum the @status, content-type, content-digest, and the selected representation headers, using the Signature-Input and Signature response headers. The Content-Digest header (RFC 9530 [RFC9530]) carries the digest of the descriptor body.

The default signature algorithm is ecdsa-p256-sha256. Verifiers MUST refuse to accept unsigned descriptors when the resolver or client policy requires signed descriptors. Key material is discovered in order of precedence from: (1) the descriptor’s authentication.authorizationServer metadata (jwks_uri per RFC 8414 [RFC8414]); (2) the descriptor’s authentication.jwksUri or inline authentication.jwks; (3) for DID-based [DID-CORE] authorities, the corresponding DID Document’s verification methods.

In-document signing (detached JWS or JWS JSON Serialization per RFC 7515 [RFC7515]) is permitted for deployments that cannot use HTTP Message Signatures (e.g., descriptors distributed out of band). When used, the signed payload MUST be the descriptor canonicalized per JSON Canonicalization Scheme (RFC 8785 [RFC8785]) and the serialized JWS MUST be carried alongside the descriptor rather than inlined.

10.4. Local Agent Security

The agent+local:// scheme requires special security handling. The consent flow for local agent invocation:

1. A web application or client requests invocation of agent+local://<agent-name>/<skill>.

2. The browser or runtime MUST display a user consent dialog identifying the requesting origin and the target agent.

3. The user approves or denies the request.

4. If approved, the runtime establishes an IPC channel to the local agent.

5. The runtime mediates all subsequent invocations for this origin-agent pair until the user revokes consent.

Browsers and runtimes MUST enforce same-origin policy for local agent access. Consent grants SHOULD be scoped to specific capabilities and SHOULD be revocable.

10.5. Resolver Implementation Security

Resolvers MUST use HTTPS with certificate validation. Resolvers SHOULD validate descriptor integrity using ETag/Last-Modified for caching. If signature metadata is present, resolvers SHOULD verify signatures per the descriptor's signing scheme. See also Section 5.2 for SSRF protections.

10.6. Resolution Security Threats

Implementations SHOULD be aware of the following threats to the resolution mechanism:

* *Registry poisoning*: If /.well-known/agents.json is compromised, an attacker could redirect all agent resolution to malicious endpoints. Descriptor signing (Section 10.3) mitigates this but is currently OPTIONAL. Production deployments SHOULD sign descriptors.

* *Descriptor substitution*: A malicious descriptor URL could return a modified descriptor with altered endpoints or capabilities. Clients SHOULD validate descriptor integrity via ETag or signatures when available.

* *Open redirect*: A malicious agents.json could point to arbitrary URLs. Resolvers SHOULD validate that descriptor URLs use HTTPS and do not redirect to unexpected domains.

* *Confused deputy via depends*: An agent's depends field could reference malicious agents, causing legitimate agents to invoke attacker-controlled endpoints. Orchestrators SHOULD validate dependency chains and MAY restrict depends to trusted domains.

10.7. Privacy

Agents SHOULD minimize data retention and SHOULD expose revoke/delete interfaces for user data where applicable.

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

10.8. 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:

* New transport bindings (registered via Section 12.4)
* New interaction model identifiers (registered via Section 12.5)
* Extended agent descriptor fields via JSON-LD contexts and vendor namespaces (see Section 7.2)
* Companion specifications for functionality out of scope for the addressing layer

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.

11.1. Intentional Scope Limits

The following concerns are explicitly *out of scope* for this specification and are delegated to existing vocabularies or companion specifications:

* *Commerce and pricing* -- use schema.org Offer and PriceSpecification via Section 7.2.

* *Quantitative performance metrics* -- use domain-specific observability protocols (e.g., OpenTelemetry semantic conventions).

* *Underlying model or framework identification* -- use schema.org SoftwareApplication or vendor contexts.

* *Orchestration semantics* (task lifecycles beyond the minimal async pattern, state machines, compensation) -- use the communication-layer protocol in use (e.g., [Agent2Agent]).

* *UI metadata* (icons, colors, sample prompts, user-facing language preferences) -- use AgentCard extensions or vendor namespaces.

* *Commerce contracts, SLAs, legal terms* -- use domain-specific vocabularies.

This sharper scope is intentional. Keeping the addressing and discovery layer small allows it to interoperate cleanly with evolving application-layer protocols without forcing implementers to adopt opinionated vocabulary for concerns they do not have.

11.2. Planned Companion Specifications

The following are planned as separate companion specifications:

* *Dynamic Agent Registry Protocol*: A protocol for runtime agent registration and deregistration, enabling cloud-native environments with auto-scaling agents.

* *Local Agent Discovery Protocol*: A standardized local-runtime discovery mechanism (the local equivalent of /.well-known/agents.json) for browsers, desktop runtimes, and embedded environments.

* *Binary Encoding (CBOR)*: An application/agent+cbor media type for constrained IoT and edge environments requiring compact descriptor encoding.

* *JSON-LD Context for agent descriptors*: A published context document mapping descriptor fields to stable IRIs for semantic interoperability.

12. IANA Considerations

This document requests the following IANA registrations.

12.1. URI Scheme Registration Template

* *Scheme Name*: agent

* *Status*: Provisional

* *Applications/Protocols That Use This Scheme*: The agent URI scheme identifies and enables discovery of autonomous or semi-autonomous software agents across systems. It provides a transport-agnostic addressing and discovery layer that complements communication protocols such as A2A, MCP, and ACP. The scheme is compatible with existing schemes such as https and did where appropriate.

* *Contact*: agent:// Specification Maintainers iana@agent-uri.org (mailto:iana@agent-uri.org)

* *Change Controller*: The authors 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 RFC 7595 [RFC7595] - Guidelines and Registration Procedures for URI Schemes

* *URI Syntax*: The general form of an agent URI is:

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

* *Related Registrations:*

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

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

  - Transport Bindings Registry (Section 12.4)

  - Interaction Models Registry (Section 12.5)

Where: - authority is typically a domain name or Decentralized Identifier (DID) - path identifies the agent and optionally a skill - query includes serialized key-value parameters - fragment MAY reference a sub-skill 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, gRPC, or local execution environments. Implementers are advised to apply standard authentication and authorization measures. See Section 10 for guidance.

12.2. Well-Known URI Registrations

This document registers the following Well-Known URIs per RFC 8615 [RFC8615]:

1. *URI Suffix:* agents.json *Change Controller:* The authors or IETF if adopted. *Status:* provisional *Reference:* This document. *Related Information:* Provides a registry mapping of agent names to descriptor URLs. Both single-agent and multi-agent domains use this format. Accessible only via HTTPS.

Agent health and liveness signaling is intentionally NOT registered as a well-known URI. Deployments SHOULD use conventional health endpoints (/health, /healthz, /readyz), the gRPC Health Checking Protocol (https://github.com/grpc/grpc/blob/master/doc/health-checking.md) for gRPC transport, or application/health+json per draft-inadarei-api-health-check [I-D.inadarei-api-health-check]. Health-check output is dynamic and does not fit the static-discovery semantics of well-known URIs.

12.3. Media Type Registration for application/agent+json

Per RFC 6838 [RFC6838]:

Type name: application Subtype name: agent+json Required parameters: none Optional parameters: profile (a URI identifying a JSON-LD context document; follows RFC 6906 [RFC6906]) 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. The +json structured syntax suffix per RFC 6838 Section 4.2.8 allows generic JSON processors to parse the content. Fragment identifier considerations: Fragment identifiers for application/agent+json documents follow JSON Pointer [RFC6901] syntax per RFC 6901. This differs from the semantic fragment use in agent:// URIs (Section 4.1), which MAY identify a sub-skill or context; agent URI fragments are not evaluated against the descriptor document. Published specification:

This document Applications that use this media type: Agent resolvers and runtimes using the agent:// protocol. Magic number(s): N/A File extension(s): .agent.json Macintosh file type code(s): N/A Restrictions on usage: None Additional information: None Person & email address to contact for further information: agent:// Specification Maintainers iana@agent-uri.org (mailto:iana@agent-uri.org) 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].

12.4. Transport Bindings Registry

This document establishes a new IANA registry: "Agent URI Transport Bindings".

* *Registration Procedure*: Specification Required

* *Reference*: This document

Each entry MUST specify:

* *Protocol identifier* (e.g., https, grpc)

* *URI format* (the full agent+<protocol>:// scheme form)

* *Connection mechanism* (how a session is established)

* *Message framing* (the wire-level payload structure)

* *Error signaling* (how errors are communicated on this transport)

* *Specification reference* (a stable normative document)

Initial entries:

Protocol	Format	Connection	Framing	Error Signaling	Specification
https	agent+https://	HTTPS	HTTP req/resp	HTTP status + 9457	RFC 9110 [RFC9110]
wss	agent+wss://	WS handshake	NDJSON	JSON error	RFC 6455 [RFC6455]
grpc	agent+grpc://	HTTP/2 + TLS	Protobuf	gRPC status codes	[gRPC]
mqtt	agent+mqtt://	MQTT 5.0	JSON over topics	MQTT reason codes	MQTT 5.0 [MQTT]
local	agent+local://	IPC (impl.)	JSON	JSON error	This document
unix	agent+unix://	Unix socket	NDJSON	JSON error	This document

Table 7: Initial Transport Bindings Registry

12.5. Interaction Models Registry

This document establishes a new IANA registry: "Agent Interaction Models".

* *Registration Procedure*: Expert Review

* *Reference*: This document

Initial entries:

Model	Specification
agent2agent	[Agent2Agent]
mcp	[MCP]
fipa-acl	[FIPA-ACL]
openapi	OpenAPI [OPENAPI-3.1]

Table 8: Initial Interaction Models Registry

This specification does NOT register new HTTP header fields. Version negotiation uses the standard Accept; profile= parameter per RFC 6906 [RFC6906]. Trace correlation uses W3C Trace Context [W3C-TRACE-CONTEXT] (traceparent, tracestate). Session and task correlation use W3C Baggage [W3C-BAGGAGE]. Delegation tokens use the standard Authorization header per RFC 6750 [RFC6750]. Idempotency uses Idempotency-Key per draft-ietf-httpapi-idempotency-key-header [I-D.ietf-httpapi-idempotency-key-header]. Rate limits use RateLimit and RateLimit-Policy per draft-ietf-httpapi-ratelimit-headers [I-D.ietf-httpapi-ratelimit-headers].

13. Appendix A. Example Agent Descriptor

Following is an example of a complete agent descriptor. Note that input and output fields use JSON Schema format:

```
{
  "name": "planner.example.com",
  "description": "Agent helps in researching & planning itineraries",
  "url": "agent://planner.example.com/",
  "status": "active",
  "conformanceLevel": 3,
  "environment": "production",
  "provider": {
    "organization": "Example AI Org",
    "url": "https://example.com"
  },
  "documentationUrl": "https://planner.example.com/docs",
  "interactionModel": ["agent2agent", "mcp"],
  "version": "3.1.4",
  "supportedVersions": {
    "3.0.0": "/v3/",
    "2.1.2": "/olderversion/v2.1.2/"
```

```
    },
    "transport": {
      "endpoint": "https://planner.example.com/api",
      "https": "https://planner.example.com/api",
      "wss": "wss://planner.example.com/ws"
    },
    "skills": [
      {
        "id": "gen-iti",
        "name": "Generate Itinerary",
        "version": "2.1.5",
        "description": "Creates a travel itinerary for a given city.",
        "tags": ["travel", "planning"],
        "input": {
          "type": "object",
          "properties": { "city": { "type": "string" } },
          "required": ["city"]
        },
        "output": {
          "type": "object",
          "properties": { "itinerary": { "type": "array" } }
        },
        "streaming": true,
        "streamingFormat": "sse",
        "idempotent": true,
        "contentTypes": {
          "accepts": ["application/json", "application/ld+json"],
          "produces": ["application/json"]
        },
        "depends": [
          {
            "uri": "agent://translator.example.com/translate",
            "relation": "invokes",
            "versionConstraint": "^2.0.0"
          }
        ]
      }
    ],
    "authentication": {
      "schemes": ["Bearer"],
      "authorizationServer": "https://auth.example.com",
      "protectedResourceMetadata": "https://planner.example.com/.well-known/oauth-protec
ted-resource"
    }
  }
```

Figure 8: Example Agent Descriptor

Descriptors MAY additionally include a JSON-LD @context field to support semantic querying and graph-based processing. When present, a published JSON-LD context document SHOULD be available at the referenced URL.

14. Appendix B. Use Cases

* Coordinating tasks across 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
* High-performance agent-to-agent communication via gRPC transport
* IoT and edge agent networks using MQTT transport

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 (agents.json, .well-known endpoints)
* Transport bindings including HTTPS, WebSocket, gRPC, MQTT, and Local IPC
* Agent descriptor discovery, caching, and validation
* OAuth 2.0 Token Exchange (RFC 8693) act-claim delegation 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 reference implementation [AGENT-URI-REPO]

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

16. Appendix D. Normative JSON Schema

The normative JSON Schema for agent descriptors is available at:

https://agent-uri.org/schemas/agent-descriptor.schema.json

A copy is included in the reference implementation repository under docs/rfc/schemas/agent-descriptor.schema.json. See Section 7.1 for a prose description of required, recommended, and optional fields.

A JSON Schema for the agents.json registry format is also provided at docs/rfc/schemas/agents-registry.schema.json.

17. Appendix E. Conformance Level Requirements

Detailed requirements per conformance level:

Level 0 (Direct Invocation):

* MUST support parsing agent+<protocol>:// URIs
* MUST support at least one transport binding (typically HTTPS)
* No descriptor or resolution infrastructure required

Level 1 (Discoverable):

* All Level 0 requirements
* MUST publish /.well-known/agents.json with at least one agent entry
* Agent descriptors MUST include the required fields: name, version, skills

Level 2 (Resolvable):

* All Level 1 requirements
* MUST implement the resolution algorithm (Section 5.1)
* MUST respect Cache-Control headers on descriptor responses
* SHOULD support conditional requests (If-None-Match, If-Modified-Since)

* MAY support multiple transport bindings

Level 3 (Full):

* All Level 2 requirements

* Descriptors SHOULD include recommended fields (description, url, transport, authentication, provider)

* SHOULD support version negotiation via Accept: application/agent+json; profile="..." per RFC 6906 [RFC6906]

* MAY sign descriptors via HTTP Message Signatures (RFC 9421 [RFC9421])

* MAY carry delegation via OAuth 2.0 Token Exchange nested act claim per RFC 8693 [RFC8693]

18. Appendix F. Quick-Start Guide

18.1. Minimal Agent Setup (Level 1)

Step 1: Create an agent descriptor file (agent.json):

```
{
  "name": "my-agent",
  "version": "1.0.0",
  "skills": [
    {
      "id": "hello",
      "name": "Hello",
      "description": "Returns a greeting"
    }
  ]
}
```

Step 2: Create /.well-known/agents.json on your domain:

```
{
  "agents": {
    "my-agent": "https://example.com/my-agent/agent.json"
  }
}
```

Step 3: Serve the agent descriptor at the URL referenced above.

Step 4: Clients can now discover your agent:

```
# Resolve the agent
GET https://example.com/.well-known/agents.json

# Fetch the descriptor
GET https://example.com/my-agent/agent.json

# Invoke the skill
POST https://example.com/my-agent/hello
Content-Type: application/json

{"name": "World"}
```

Using the reference implementation:

```
from agent_uri import AgentClient

client = AgentClient()
result = client.invoke("agent://example.com/my-agent/hello", params={"name": "World"})
```

19. Appendix G. Comparison with Related Protocols

19.1. agent:// vs HTTPS + OpenAPI

OpenAPI [OPENAPI-3.1] describes REST APIs. agent:// adds agent identity signaling (the scheme itself says "this is an agent") and transport abstraction (the same URI can resolve to HTTPS, gRPC, or local IPC). The two layers are complementary: a skill's invocation surface MAY be described in OpenAPI and referenced from the descriptor.

19.2. agent:// vs Agent2Agent (A2A)

A2A [Agent2Agent] (by Google) defines a communication protocol for agent-to-agent interaction with task lifecycle management, streaming, and push notifications. agent:// complements A2A by providing the addressing and discovery layer: the descriptor's skills array aligns with AgentCard's skills, enabling bidirectional mapping; the interactionModel array signals A2A compatibility.

19.3. agent:// vs Model Context Protocol (MCP)

MCP (by Anthropic) defines how language models access tools, resources, and prompts. agent:// operates at a different layer: it addresses agents rather than tools. An agent may internally use MCP to access tools, while being addressable via agent:// and communicating with other agents via A2A.

19.4. agent:// vs Agent Communication Protocol (ACP)

[ACP] (by IBM/BeeAI) focuses on standardized agent-to-agent message exchange. Like A2A, it defines communication semantics. agent:// provides the addressing and discovery layer that enables clients and agents to find ACP-compatible agents in the first place.

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. The authors thank all contributors and reviewers who have provided feedback across previous drafts. It is intended as a foundation for future refinement and collaboration.

References

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>.

[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>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/rfc/rfc5234>.

[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>.

[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>.

[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>.

[RFC8414] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", RFC 8414, DOI 10.17487/RFC8414, June 2018, <https://www.rfc-editor.org/rfc/rfc8414>.

[RFC8615] Nottingham, M., "Well-Known Uniform Resource Identifiers (URIs)", RFC 8615, DOI 10.17487/RFC8615, May 2019, <https://www.rfc-editor.org/rfc/rfc8615>.

[RFC8725] Sheffer, Y., Hardt, D., and M. Jones, "JSON Web Token Best Current Practices", BCP 225, RFC 8725, DOI 10.17487/RFC8725, February 2020, <https://www.rfc-editor.org/rfc/rfc8725>.

[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>.

[RFC9111] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Caching", STD 98, RFC 9111, DOI 10.17487/RFC9111, June 2022, <https://www.rfc-editor.org/rfc/rfc9111>.

[RFC9421] Backman, A., Ed., Richer, J., Ed., and M. Sporny, "HTTP Message Signatures", RFC 9421, DOI 10.17487/RFC9421, February 2024, <https://www.rfc-editor.org/rfc/rfc9421>.

[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

[ACP] IBM / BeeAI, "Agent Communication Protocol", 2025, <https://github.com/i-am-bee/beeai-platform>.

[AGENT-URI-REPO] Narvaneni, Y. and S. P. Ravi, "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/>.

[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/>.

[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>.

[gRPC] Cloud Native Computing Foundation, "gRPC: A High Performance, Open Source Universal RPC Framework", 2024, <https://grpc.io/docs/>.

[I-D.ietf-httpapi-idempotency-key-header] Young, J., "The Idempotency-Key HTTP Header Field", 2024, <https://datatracker.ietf.org/doc/draft-ietf-httpapi-idempotency-key-header/>.

[I-D.ietf-httpapi-ratelimit-headers] Polli, R. and A. Martinez, "RateLimit header fields for HTTP", 2025, <https://datatracker.ietf.org/doc/draft-ietf-httpapi-ratelimit-headers/>.

[I-D.inadarei-api-health-check]
Nadareishvili, I., "Health Check Response Format for HTTP APIs", 2024, <https://datatracker.ietf.org/doc/draft-inadarei-api-health-check/>.

[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/>.

[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/>.

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

[MQTT] OASIS, "MQTT Version 5.0 - OASIS Standard", March 2019, <https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html>.

[OPENAPI-3.1]
OpenAPI Initiative, "OpenAPI Specification v3.1.0", October 2024, <https://spec.openapis.org/oas/latest.html>.

[OTEL-GENAI]
OpenTelemetry Authors, "OpenTelemetry Semantic Conventions for Generative AI Systems", 2025, <https://opentelemetry.io/docs/specs/semconv/gen-ai/>.

[RFC6455] Fette, I. and A. Melnikov, "The WebSocket Protocol", RFC 6455, DOI 10.17487/RFC6455, December 2011, <https://www.rfc-editor.org/rfc/rfc6455>.

[RFC6648] Saint-Andre, P., Crocker, D., and M. Nottingham, "Deprecating the "X-" Prefix and Similar Constructs in Application Protocols", BCP 178, RFC 6648, DOI 10.17487/RFC6648, June 2012, <https://www.rfc-editor.org/rfc/rfc6648>.

[RFC6901] Bryan, P., Ed., Zyp, K., and M. Nottingham, Ed., "JavaScript Object Notation (JSON) Pointer", RFC 6901, DOI 10.17487/RFC6901, April 2013, <https://www.rfc-editor.org/rfc/rfc6901>.

[RFC7033] Jones, P., Salgueiro, G., Jones, M., and J. Smarr, "WebFinger", RFC 7033, DOI 10.17487/RFC7033, September 2013, <https://www.rfc-editor.org/rfc/rfc7033>.

[RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <https://www.rfc-editor.org/rfc/rfc7515>.

[RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <https://www.rfc-editor.org/rfc/rfc7517>.

[RFC8594] Wilde, E., "The Sunset HTTP Header Field", RFC 8594, DOI 10.17487/RFC8594, May 2019, <https://www.rfc-editor.org/rfc/rfc8594>.

[RFC8693] Jones, M., Nadalin, A., Campbell, B., Ed., Bradley, J., and C. Mortimore, "OAuth 2.0 Token Exchange", RFC 8693, DOI 10.17487/RFC8693, January 2020, <https://www.rfc-editor.org/rfc/rfc8693>.

[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>.

[RFC8785] Rundgren, A., Jordan, B., and S. Erdtman, "JSON Canonicalization Scheme (JCS)", RFC 8785, DOI 10.17487/RFC8785, June 2020, <https://www.rfc-editor.org/rfc/rfc8785>.

[RFC9068] Bertocci, V., "JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens", RFC 9068, DOI 10.17487/RFC9068, October 2021, <https://www.rfc-editor.org/rfc/rfc9068>.

[RFC9530] Polli, R. and L. Pardue, "Digest Fields", RFC 9530, DOI 10.17487/RFC9530, February 2024, <https://www.rfc-editor.org/rfc/rfc9530>.

[RFC9728] Jones, M.B., Hunt, P., and A. Parecki, "OAuth 2.0 Protected Resource Metadata", RFC 9728, DOI 10.17487/RFC9728, April 2025, <https://www.rfc-editor.org/rfc/rfc9728>.

[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/>.

[W3C-BAGGAGE] W3C Distributed Tracing Working Group, "Baggage - W3C Candidate Recommendation", W3C Candidate Recommendation, 2024, <https://www.w3.org/TR/baggage/>.

[W3C-TRACE-CONTEXT] W3C Distributed Tracing Working Group, "Trace Context - W3C Recommendation", W3C Recommendation, April 2025, <https://www.w3.org/TR/trace-context/>.

Authors' Addresses

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

Sai Purnima Ravi
Independent Researcher
Email: r.sai.purnima+ietf@gmail.com