

Independent Submission
Internet-Draft
Intended status: Informational
Expires: 13 November 2026

C. Hood
Nomotic, Inc.
12 May 2026

AGTP-API: Verbs, Paths, Endpoints, and Synthesis
draft-hood-agtp-api-00

Abstract

This document specifies AGTP-API: the contract layer that the Agent Transfer Protocol (AGTP) [AGTP] relies on to govern interactions between autonomous agents and AGTP servers. AGTP-API defines a curated approved method catalog (with versioned evolution and graceful deprecation), path grammar rules that prevent method-name leakage into paths, the endpoint primitive (the structural unit a server exposes to agents), the semantic block carried by every endpoint, schema validation requirements, the server manifest format that exposes a server's endpoint catalog, the per-server method policy carried as a sub-block of the manifest, the PROPOSE-and-synthesis runtime contract negotiation mechanism, the three handler binding kinds (composition, `registered_function`, `external_service`), and the structural rejection status codes (404, 405, 459, 460) that together cover the contract-level failure surface. This document supersedes the AGIS Internet-Draft (draft-hood-independent-agis-01) and the previously-proposed AGTP-Methods Internet-Draft, both of which are deprecated. AGTP-API is the unified companion specification they were splitting concerns across.

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 13 November 2026.

Copyright Notice

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

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

Table of Contents

1. Introduction	4
1.1. Why the Contract Belongs in the Protocol	4
1.2. Document Lineage	5
1.3. Requirements Language	6
2. Terminology	6
3. Method Catalog	7
3.1. Catalog Structure	7
3.2. Method Rules	8
3.3. Custom Methods	8
3.4. Catalog Evolution Overview	9
3.5. Catalog Version Exposure	9
3.6. Legacy Verb Support	9
4. Catalog Evolution	10
4.1. Versioning Scheme	10
4.2. Deprecation Metadata	10
4.3. Manifest Catalog Declaration	11
4.4. Runtime Deprecation Signaling	11
4.5. Removal Behavior	12
4.6. Cross-Server Version Negotiation	12
4.7. Tooling	13
4.8. Multi-Version Catalog Support	13
4.9. Catalog Publishing	14
5. Path Grammar	14
5.1. Path Rules	14
5.2. Path Grammar ABNF	14
5.3. Permissiveness	15
5.4. Path Pattern Matching	16
5.5. Query Strings	16
5.6. Fragments	17
5.7. No Reserved Path Prefixes	17
5.8. Server Metadata via AGTP Methods	17
6. The Endpoint Primitive	18
6.1. Endpoint Definition	18
6.2. Required Fields	19
6.3. Field Semantics	19

6.4. Endpoint Deprecation	21
7. The Semantic Block	22
7.1. Required Semantic Block Fields	22
8. Server Manifest	23
8.1. Purpose	23
8.2. Manifest Structure	23
8.3. Server Block	25
8.4. Methods Inventory	26
8.5. Agents Disclosure	26
8.6. Protocol and API Surface	26
8.7. Policies Block	27
8.8. Manifest Signature	27
8.9. Endpoint Projection	28
8.10. Manifest Retrieval	29
8.11. Manifest Caching	29
9. Method Policy	29
9.1. Purpose	29
9.2. Policy Shape	29
9.3. Example	30
9.4. Catalog-Graceful Skip Semantics	31
9.5. Discovery	31
10. PROPOSE and Synthesis	31
10.1. Purpose	32
10.2. Negotiation Flow	32
10.3. Server-Side Synthesis Evaluation	32
10.4. Synthesis-Composed Endpoints	33
10.5. Session-Scoped vs Persistent Synthesis	33
11. Status Codes	34
12. Handler Binding	35
12.1. Composition	35
12.2. Registered Function	36
12.3. External Service	37
13. Conformance	39
13.1. Single Tier	39
13.2. Validation Timing	40
13.3. Schema Strictness	42
13.4. Conformance Test Suite	42
14. Relationship to OpenAPI	43
14.1. OpenAPI as an Authoring Source	43
14.2. OpenAPI as an Export Target	43
14.3. OpenAPI as a Coexistence Layer	44
15. What AGTP-API Does Not Specify	44
16. Security Considerations	44
16.1. Verb-Path Tampering	45
16.2. Synthesis Authority Preservation	45
16.3. Method Policy Tampering	45
16.4. Wildcard Abuse	45
16.5. Verb List Trust	45

17. IANA Considerations	46
17.1. AGTP-API Status Code Assignments	46
17.2. Media Type Registrations	46
17.3. AGTP-API Method Catalog Reference	47
17.4. AGTP-API Response Headers	47
18. Open Items	47
19. Acknowledgments	49
20. References	49
20.1. Normative References	49
20.2. Informative References	50
Appendix A. Implementation Guidance (Informative)	50
A.1. Endpoint Authoring	50
A.2. Runtime Dispatch Pattern	51
A.3. Tooling	52
A.4. Operator Notes on Catalog Versioning	52
Appendix B. Migration Paths (Informative)	53
B.1. Wrap-and-Expose	53
B.2. Translate-and-Rehost	53
B.3. Coexist-Permanently	54
B.4. Path Selection	54
Author's Address	54

1. Introduction

AGTP [AGTP] is a transport protocol designed for AI agent traffic. It establishes identity, authority, attribution, and a twelve-method protocol-level floor. Beyond that floor, the question of what an AGTP server actually exposes -- what endpoints it presents, what verbs are valid, what paths are permitted, what schemas govern input and output, what authority is required for invocation -- is the contract layer. This document specifies the contract layer.

1.1. Why the Contract Belongs in the Protocol

The architectural goal AGTP is designed around is runtime contract negotiation between autonomous agents (RCNS). An agent encountering an AGTP server must be able to:

- * Discover what endpoints the server exposes, with full structural detail, in a single call.
- * Reason about whether an endpoint matches its intent based on the endpoint's declared semantic block.
- * Verify that its identity carries the authority required to invoke the endpoint.
- * Propose new endpoints when its needs are not met by existing ones.

- * Compose endpoints across servers while preserving authority through the composition.

None of this works if the contract lives in middleware. A protocol that does not know what an interaction means cannot govern it. When an agent invokes a method on a path, the protocol must verify identity, confirm permissions cover the method-on-path combination, confirm server policy allows the combination, validate the request body against the endpoint's input schema, apply scope checks, log the invocation, and -- if the request triggers synthesis -- ensure each composed step preserves authority. Each of these checks requires the protocol to know what the method-on-path combination means.

This is the load-bearing claim of AGTP-API. The contract is a first-class protocol concept rather than something inferred from documentation, OpenAPI specs, or framework conventions. Operational concerns -- rate limiting, observability, caching, custom authentication strategies, request transformation, deployment topology -- remain middleware territory because they are operational decisions specific to a deployment. The protocol governs contracts; middleware governs operations. Neither layer pretends to be the other.

1.2. Document Lineage

AGTP-API supersedes two previously-separate drafts:

- * AGIS (the Agentive Grammar and Interface Specification, draft-hood-independent-agis-01) -- deprecated. Retained as a historical citation source for the empirical research informing the verb-class semantics. ("Verb" here is the linguistic category — the catalog is a vocabulary of action-intent verbs. See Section 2.)
- * AGTP-Methods (the proposed Standard Extended Method Vocabulary, draft-hood-agtp-standard-methods-01) -- folded into this document. There is no separate method-catalog specification.

The previously-proposed AMG (Agent Method Grammar) draft is also deprecated; it never reached publication. The grammar framing is replaced by the contract framing throughout. This document assumes the reader understands AGTP [AGTP] v07 or later.

1.3. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD 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.

2. Terminology

Method: The action component of an AGTP request: an uppercase ASCII token appearing in the AGTP-API approved catalog or in a server's the server's policies.methods as a custom method. The method field on every endpoint declaration carries this token. The protocol-level term used throughout the wire format, manifest, and conformance language.

Verb: A descriptive synonym for "method" used when discussing the catalog as a vocabulary of action-intent verbs. The catalog at agtp.io/api/methods.json is a glossary of verbs; an operator picks a verb from the catalog and declares it as the method on an endpoint. Method is the field name; verb is the linguistic category that explains why the field carries imperative-form action-intent words.

Path: The resource portion of an AGTP request, beginning with a forward slash. Subject to the path grammar rules in Section 5.

Endpoint: A method-and-path pair exposed by an AGTP server, together with its semantic block, input schema, output schema, error declarations, and handler binding. The deployable unit of an AGTP service. Endpoints are first-class protocol concepts.

Semantic Block: Required structural metadata for every endpoint: intent, actor, outcome, capability classification, confidence guidance, impact tier, and idempotency. Encodes machine-readable intent so that agents can reason about endpoint applicability before invoking. Carried on the endpoint as the semantic field.

Server Manifest: The machine-readable document an AGTP server publishes to declare its endpoint catalog, hosted agents, hosted protocols, and policies. The authoritative source for what a server offers.

Method Policy: A per-server policy block declaring allowed methods, disallowed methods, legacy verb opt-in, and server-local method-to-method redirects. Carried as the policies.methods sub-block of the server manifest. See Section 9.

Synthesis: The runtime instantiation of an endpoint in response to a PROPOSE request, composing the requested behavior from existing endpoints while preserving authority through the composition.

Wildcards: An ad-hoc method invocation mode in which an agent invokes a method outside the server's declared endpoint catalog.
Subject to mutual consent: requires wildcards: true in the agent identity document and wildcards_accepted: true in server policy.

3. Method Catalog

AGTP-API defines a curated catalog of approved methods maintained at <https://agtp.io/api/methods.json>. The catalog is the canonical source for what method names are recognized. Method names not appearing in the catalog are rejected with status 459 Method Violation.

The catalog is a vocabulary of action-intent verbs: imperative-form words that describe what an operation does (BOOK, RESERVE, AUDIT, QUERY). A verb in the catalog becomes a `_method_` when an operator picks it from the catalog and declares it as the method field on an endpoint. The two terms are related: verbs are the linguistic content of the catalog; methods are the protocol-level identifier on each endpoint and request.

3.1. Catalog Structure

The catalog is a versioned JSON document with five top-level keys:

version Semantic version of the catalog as a whole. See Section 4.

embedded The 12 protocol-floor verbs every AGTP server **MUST** support: six cognitive (QUERY, DISCOVER, DESCRIBE, SUMMARIZE, PLAN, PROPOSE) and six mechanics (EXECUTE, DELEGATE, ESCALATE, CONFIRM, SUSPEND, NOTIFY).

legacy The five HTTP-style verbs (GET, POST, PUT, DELETE, PATCH), each carrying a preferred mapping to its canonical AGTP replacement (FETCH, CREATE, REPLACE, REMOVE, MODIFY). Legacy verbs are opt-in per server through the manifest's `policies.methods` block. See Section 3.6.

categories The taxonomy used to classify verbs: discovery, retrieval, analysis, transaction, modification, creation, notification, mechanics, domain_spanning.

verbs The catalog of approved verbs, each with categories,

description, and optional deprecation metadata (deprecated_in, removed_in, successor).

The catalog is published in machine-readable form at the URL above and is intentionally maintained as an open living artifact rather than a closed IANA registry. Curation is performed by the AGTP-API maintainers in public; the version of the catalog in effect at any given moment is declared in the catalog_version field of every server manifest. A closed IANA registry would freeze the verb vocabulary at the cadence of IETF process; AGTP-API verbs need to evolve at the cadence of agent deployments. The IANA registries that AGTP-API does request (Section 17) cover the structural status codes, the media types, and the response headers — all of which benefit from IANA's conservative cadence. The catalog itself does not.

3.2. Method Rules

Method names **MUST** be uppercase ASCII single tokens matching the regex `^[A-Z]+$`, length 3 to 32 characters inclusive. Method names **MUST** appear in the catalog, or in the legacy set when the server's policies.methods.legacy opts into legacy verbs.

The catalog is curated to contain only imperative-form action-intent verbs that:

- * Express a command or action, not a state, an attribute, or a noun (RESERVE is valid; RESERVATION is not).
- * Carry a single intent (verbs with compound intent are decomposed into separate verbs).
- * Are not HTTP method names.

These are curatorial principles applied during verb addition, not runtime checks performed by AGTP servers. Servers validate verbs against the catalog membership and the lexical regex; semantic class adherence is the responsibility of the catalog maintainers.

3.3. Custom Methods

Servers **MAY** accept additional methods beyond the catalog by listing them in the manifest's policies.methods block. Custom methods follow the same lexical rules in Section 3.2 but are server-specific and are not interoperable across organizations without explicit acceptance. Custom methods are not added to the catalog; they exist only within the publishing server's surface.

3.4. Catalog Evolution Overview

The catalog evolves under semver discipline:

- * ***Patch*** (1.0.0 → 1.0.1) — description or category metadata changes only.
- * ***Minor*** (1.0.0 → 1.1.0) — verbs added; existing verbs unchanged. Minor revisions ***MAY*** introduce new deprecation flags on existing verbs.
- * ***Major*** (1.0.0 → 2.0.0) — verbs removed or renamed.

Per-verb deprecation uses three optional fields: `deprecated_in` (catalog version that flagged the verb), `removed_in` (version scheduled for removal), and `successor` (recommended replacement). A deprecated verb remains admitted by the dispatcher; the response carries an AGTP-Catalog-Warning advisory header. See Section 4 for the full lifecycle, runtime semantics, removal behavior, cross-server version negotiation, and the `agtp-catalog-diff` tooling.

3.5. Catalog Version Exposure

The catalog version is exposed on the server manifest under `catalog_version` and `catalog_versions_supported` so clients can detect mismatches between their local catalog and the server's. See Section 8.

3.6. Legacy Verb Support

AGTP-API recognizes GET, POST, PUT, DELETE, and PATCH as legacy verbs with reframed semantics for transitional deployments. Legacy verb support is opt-in per server through the manifest's `policies.methods` block:

```
legacy_verbs: enabled
```

When legacy verbs are enabled, the server ***MUST*** treat them as follows:

- * GET maps to QUERY for read operations against a known path.
- * POST maps to EXECUTE for state-changing operations.
- * PUT maps to EXECUTE with `idempotency_key`.
- * DELETE maps to EXECUTE with `action: "delete"`.

* PATCH maps to EXECUTE with action: "modify".

These mappings are conveniences for migration. Legacy verbs **MUST NOT** be used in new endpoint definitions; servers **SHOULD** deprecate legacy support over time.

4. Catalog Evolution

The AGTP-API approved method catalog is versioned. This section specifies how catalog versioning works, how deprecation is signaled, and how servers and agents negotiate over catalog versions.

4.1. Versioning Scheme

The method catalog is versioned as semver MAJOR.MINOR.PATCH:

- * **PATCH** — editorial changes (clarifications, typo fixes, description rewordings). No verb additions, removals, or deprecations.
- * **MINOR** — additions and deprecations. Verbs may be added; verbs may be marked deprecated. No verbs are removed in a minor revision.
- * **MAJOR** — removals. Verbs marked deprecated in earlier minor revisions may be removed.

Servers and agents using a given catalog MAJOR version are guaranteed that no verb available in that MAJOR will disappear without warning; removal requires a MAJOR bump.

4.2. Deprecation Metadata

Each verb in the catalog **MAY** carry deprecation metadata:

```
{
  "name": "AUDIT_LEGACY",
  "category": "compute",
  "deprecated_in": "1.2.0",
  "removed_in": "2.0.0",
  "successor": "AUDIT"
}
```

`deprecated_in`: The catalog version at which the verb became deprecated.

`removed_in`: The anticipated catalog version at which the verb will

be removed. **MAY** be absent if removal is anticipated but the target version has not been chosen.

successor: A catalog verb name that callers should migrate to. **MAY** be absent if no single successor applies; clients should consult the verb's description for migration guidance.

4.3. Manifest Catalog Declaration

Every server manifest **MUST** declare:

catalog_version: The catalog version the server validates incoming method names against.

catalog_versions_supported: A list of catalog versions the server can validate against. The current catalog_version **MUST** appear in this list. Multi-version support is anticipated; v00 servers commonly list a single version.

Agents reading a manifest learn which catalog version they should use for outbound requests against this server. Mismatches are handled at request time per the runtime semantics below.

4.4. Runtime Deprecation Signaling

When a server processes a request whose method is deprecated in the catalog version the server is using, the server **MUST** include the AGTP-Catalog-Warning header on the response:

AGTP-Catalog-Warning: deprecated; successor=AUDIT; removed_in=2.0.0

Header fields:

- * deprecated — keyword token indicating the warning class.
- * successor=NAME — the recommended replacement verb. Omitted if the catalog metadata does not declare a successor.
- * removed_in=VERSION — the target removal version. Omitted if not declared.

The header is advisory, not a rejection. The request **MUST** still process. Agents and CLIs that surface the header to their users allow operators to schedule migration before forced removal.

A server that receives a request whose method is in the catalog at all (deprecated or not) **MUST NOT** return 459 Method Violation. 459 is reserved for verbs that are not in the catalog at all. Deprecated verbs are still in the catalog.

4.5. Removal Behavior

When a verb is removed in a MAJOR catalog revision, servers upgrading to that revision **MUST**:

1. Reject inbound requests for the removed verb with 459 Method Violation, citing the catalog version the server is now using.
2. Refuse to load endpoint TOMLs declaring removed verbs at startup, logging which endpoints are affected.
3. Invalidate any active synthesis plans whose steps reference removed verbs, returning a structured failure on the next synthesis tick.
4. Skip policies.methods entries (allow, disallow, redirects) that reference removed methods, logging each skip.

This is graceful degradation: the server starts, serves the endpoints that are still valid, and surfaces what was lost. The operator decides whether to roll back, fix the affected endpoints, or accept the reduced surface.

4.6. Cross-Server Version Negotiation

When an agent talks to a server whose catalog_version differs from the agent's own preferred version, the agent **SHOULD**:

1. Read catalog_versions_supported from the manifest.
2. If the agent's preferred version is in the list, use it; the server will validate against that version.
3. If not, downgrade to the highest mutually-supported version.
4. If no overlap exists, abandon the connection or escalate to the principal.

Servers **MAY** support multiple catalog versions concurrently; servers **MAY** support only the single version they declare in catalog_version. Agents adapt to the server's capabilities.

4.7. Tooling

A reference `agtp-catalog-diff` tool is published alongside the method catalog. The tool diffs two catalog versions, reports added / removed / newly-deprecated methods, and *MAY* scan a deployment directory (endpoint TOMLs, recipe registry, `agtp-server.toml`) to identify which deployment artifacts would break under a candidate catalog upgrade. See Appendix A.

Tool invocation:

```
agtp-catalog-diff old.json new.json [--against-deployment <directory>]
```

The tool produces a structured diff showing methods added in the new catalog, methods removed, methods newly deprecated, path-grammar conflicts (existing endpoint paths containing newly-added method names), endpoint conflicts (existing endpoints declaring removed methods), recipe conflicts (existing recipes referencing removed methods), and policy conflicts (existing policy directives referencing removed methods).

Operators *SHOULD* run this tool before upgrading a server's catalog version. The tool's exit code indicates whether breaking changes exist: 0 (no breakage), 1 (breakage in deployment context), 2 (parse errors).

The tool is implementation-provided; alternative implementations *MAY* ship equivalents. The diff format is specified by the implementation's documentation rather than normatively by this section.

4.8. Multi-Version Catalog Support

Future revisions of AGTP-API may specify how servers can validate against multiple catalog versions simultaneously, allowing graceful migration during transitions. The `catalog_versions_supported` manifest field is reserved for this purpose.

For v00 conforming servers, `catalog_versions_supported` *MUST* be a single-element array containing only the server's current `catalog_version`.

4.9. Catalog Publishing

The canonical AGTP method catalog is published at <https://agtp.io/api/methods.json>. The catalog is versioned independently of AGTP-API itself; a server may implement AGTP-API v00 against catalog version 1.0.0 today, and against catalog version 1.1.0 after a minor catalog update.

Servers **MAY** validate against alternative catalogs (industry-specific catalogs, internal-only catalogs, etc.) by configuring their build to use a different catalog document. The `catalog_version` exposed in the manifest reflects whichever catalog the server actually validates against.

Catalog updates follow a coordinated release process: the publisher announces the upcoming version, operators run the diff tool against their deployments, the new catalog is published, operators upgrade their servers. Specific governance for the canonical catalog (proposal process, approval cadence, etc.) is documented separately and is out of scope for this specification.

5. Path Grammar

AGTP path grammar prevents method-name leakage — the situation where a method token ends up in a path segment instead of on the request line. The grammar enforces that one rule plus structural minimums; everything else is operator judgment.

5.1. Path Rules

A request path **MUST** begin with /. A request path **MUST NOT** end with / unless the path is exactly / (the bare root). No path segment **MAY** match an approved AGTP method name (case-insensitive, after stripping - and _). Parameter segments wrapped in {...} are exempt from the method-token check.

Any violation **MUST** return status **460 Endpoint Violation** with the offending segment in the response body.

5.2. Path Grammar ABNF

The path component of the request line follows this grammar:

```

path          = "/" [ segment-nz *( "/" segment ) ]
segment       = *segment-char
segment-nz    = 1*segment-char
segment-char  = unreserved / pct-encoded / sub-delims
               / ":" / "@"
parameter     = "{" parameter-name "}"
parameter-name = 1*( ALPHA / DIGIT / "_" )
unreserved    = ALPHA / DIGIT / "-" / "." / "_" / "~"
pct-encoded    = "%" HEXDIG HEXDIG
sub-delims    = "!" / "$" / "&" / "'" / "(" / ")"
               / "*" / "+" / "," / ";" / "="
               ; per RFC 3986

```

A segment **MAY** be a literal segment (matching segment-nz with no { or } characters) or a parameter segment (matching the parameter production exactly). Mixing literal and parameter forms within a single segment (e.g., prefix-{id}) is **NOT** supported in v00; future revisions **MAY** specify mixed-form templates.

Parameter names within a path **MUST** be unique. A path containing two parameters with the same name **MUST** be rejected at endpoint registration time.

5.3. Permissiveness

Mixed case, underscores, hyphens, and any segment depth are valid. The protocol does not impose style preferences. Casing conventions, kebab-vs-snake-case, and parameter-naming style are operator decisions.

Path templates support typed parameters in curly braces:

```

/customers/{customer_id}
/orders/{order_id}/line-items/{line_item_id}
/products/category/{category_slug}

```

The {name} form is the only template syntax v00 admits. URI Template ([RFC6570]) syntax (e.g., {?query,page}, {+reserved}, {#fragment}) is **NOT** supported; servers encountering a path template that uses anything other than the basic {name} form **MUST** reject the endpoint at registration time. The conservative single-form rule keeps endpoint dispatch deterministic and predictable; future revisions **MAY** broaden the template grammar.

Template parameters **MUST** be declared in the endpoint's `input_schema` as JSON Schema properties of the same name. A template parameter that is not declared in `input_schema` **MUST** cause registration to fail. Parameter values arrive at the handler as part of the validated input alongside body parameters.

5.4. Path Pattern Matching

Servers match an incoming request path against registered endpoints in three passes:

1. **Exact match.** If the request path exactly equals a registered literal path (no template parameters), the matching endpoint is selected.
2. **Template match.** If no exact match, the request path is matched against registered templates. A template matches when the path has the same number of segments as the template and every literal segment matches exactly while every `{param}` segment captures the corresponding request segment as a parameter value.
3. **No match.** If neither pass matches, the server **MUST** return 404 Not Found.

When multiple templates would match the same request path, the server **MUST** select the template with the fewest parameter segments (most specific wins). If two templates have the same number of parameter segments and both match, registration **MUST** have failed at startup with a path-ambiguity error; the dispatcher **MUST NOT** be reached in this state.

5.5. Query Strings

Paths **MAY** carry a query string after a `?`:

AGTP/1.0 SCHEDULE /meeting?date=050526&attendees=alice%2Cbob

The wire layer splits the request-target at the first `?` per [RFC3986]: the path is everything from the leading `/` up to the first `?` or end-of-line; the query string is everything after the first `?`. Path-grammar enforcement (method-name leakage check, template matching) applies to the path component only; query strings carry no semantic constraints from this grammar.

Query parameters merge into the request input alongside body parameters before schema validation. On key conflicts, body parameters take precedence. Repeated query keys collapse to the last value; multi-valued shapes ride in the body.

Servers **MUST** percent-decode query parameter values per [RFC3986] Section 2.1 before schema validation.

5.6. Fragments

URI fragments (#anchor) are **REJECTED** at the wire layer. AGTP traffic carries no client-side anchoring use case; a # in the request line is always malformed. Servers **MUST** return 400 Bad Request with error code invalid-request-line for any request line containing #.

5.7. No Reserved Path Prefixes

AGTP reserves no path prefixes. There is no /.well-known/ discovery surface and no /_agtp/ namespace. AGTP is its own protocol; HTTP-style discovery conventions do not apply.

5.8. Server Metadata via AGTP Methods

Server-internal metadata is exposed via AGTP methods at AGTP-native paths, registered as built-in endpoints alongside operator-authorized endpoints. Servers **MUST** expose DISCOVER /methods, returning a compact inventory of every endpoint the server has registered:

```
[
  {
    "method": "QUERY",
    "path": "/catalog",
    "description": "Returns the current product catalog."
  },
  {
    "method": "BOOK",
    "path": "/room",
    "description": "Books a room for the named guest at the named property."
  },
  {
    "method": "DISCOVER",
    "path": "/methods",
    "description": "Lists all registered endpoints on this server."
  }
]
```

The response is a JSON array of objects, each with method, path, and description. The verb is DISCOVER because the response carries endpoints to talk to (the AGTP semantics for DISCOVER, see [AGTP]) rather than data to consume; agents use the response to drive follow-on invocations against the listed endpoints.

The DISCOVER /methods response is a lightweight inventory; agents that need full endpoint definitions (semantic blocks, schemas, deprecation metadata, handler types) **MUST** retrieve the full server manifest via target-less DISCOVER (Section 8). The built-in exists as a cheap, low-bandwidth probe for endpoint shape — useful for clients that need to know what to ask for without loading the full manifest.

Built-in endpoints appear in the server manifest like any other endpoint (including DISCOVER /methods itself, which is one of the entries returned by its own invocation), are subject to the same dispatcher gates (catalog, path grammar, policy), and **MAY** be overridden by operator-authored TOML at the same (method, path) pair. Operators that override a built-in are responsible for preserving the documented response contract.

Future revisions of this specification **MAY** define additional built-in endpoints (e.g., QUERY /catalog-version for catalog version negotiation). Built-ins are protocol-level conventions, not deployment-level extensions.

6. The Endpoint Primitive

6.1. Endpoint Definition

An endpoint is the structural unit an AGTP server exposes. Every endpoint is identified by the pair (method, path) and carries the fields specified below.

```
{
  "method": "BOOK",
  "path": "/room",
  "description": "Books a room for the named guest at the named property.",
  "namespace": "reservations",

  "semantic": {
    "intent": "Reserve a room for the named guest at the named property.",
    "actor": "agent",
    "outcome": "A confirmed reservation_id is returned for the guest.",
    "capability": "transaction",
    "confidence": 0.85,
    "impact": "irreversible",
    "is_idempotent": false
  },

  "input_schema": {
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "type": "object",
```

```

    "properties": {
      "guest_id": { "type": "string", "format": "uuid" },
      "room_id": { "type": "string" },
      "arrival": { "type": "string", "format": "date" },
      "departure": { "type": "string", "format": "date" }
    },
    "required": ["guest_id", "room_id", "arrival", "departure"],
    "additionalProperties": false
  },

  "output_schema": {
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "type": "object",
    "properties": {
      "reservation_id": { "type": "string", "format": "uuid" }
    },
    "required": ["reservation_id"],
    "additionalProperties": true
  },

  "errors": ["room_unavailable", "invalid_dates"],

  "handler": { "type": "registered_function" },

  "required_scopes": ["booking:room", "calendar:write"],

  "deprecated": {
    "deprecated_in": "2.1.0",
    "removed_in": "3.0.0",
    "successor": { "method": "RESERVE", "path": "/rooms" }
  }
}

```

6.2. Required Fields

Every endpoint *MUST* carry: method, path, description, semantic, input_schema, output_schema, errors, handler. The remaining fields (namespace, required_scopes, deprecated) are optional.

6.3. Field Semantics

method: An AGTP method drawn from the approved catalog at the server's declared catalog_version, or a custom method declared in the server's policies.methods. See Section 3.3. The method name satisfies the lexical rules of Section 3.2.

path: A path conforming to the path grammar of Section 5.

description: Operator-facing prose. A single sentence describing what the endpoint does. Distinct from the semantic block: description is for humans reading the manifest; the semantic block is for agents reasoning about applicability.

namespace: Optional free-form grouping label. Servers exposing many endpoints **MAY** organize them by namespace (e.g., "reservations", "billing", "inventory") for catalog navigation. Namespace has no protocol-level effect.

semantic: The structural metadata block defined in Section 7. Required by every endpoint with all required fields populated.

input_schema: A Draft 2020-12 JSON Schema [JSON-SCHEMA] describing the request body. **MUST** declare "type": "object" and **MUST** set "additionalProperties": false; see Section 13.3. Query-string parameters (Section 5.5) are merged into the body before validation; the same schema covers both. Schemas are inline in v00; cross-endpoint schema sharing is anticipated in a future revision.

output_schema: A Draft 2020-12 JSON Schema describing the successful response body. **SHOULD** set "additionalProperties": true so handlers can return forward-compat fields without bumping the contract; see Section 13.3.

errors: An array of named error condition strings (e.g., "room_unavailable", "invalid_dates"). The dispatcher returns these as 422 Unprocessable Entity responses with structured bodies identifying the condition. The error name is the protocol-level identifier; status-code mapping is a response-construction concern, not part of the contract. The list **MAY** be empty for endpoints with no business-level errors. Composition handlers **MUST** include "composition_failed"; external-service handlers **MUST** include the upstream-failure codes specified in Section 12.3.

handler: An object declaring the binding kind: {"type": "registered_function" | "composition" | "external_service"}. The handler's full reference (Python dotted path, recipe name, upstream URL, etc.) is server-internal implementation detail and **MUST NOT** appear in the manifest. See Section 8.9 and Section 12.

required_scopes: An array of scope identifiers the invoking agent

MUST declare in its Authority-Scope. Empty or omitted means no scope check; the endpoint inherits the server's default authority requirement. Insufficient scope returns 455 Scope Violation per [AGTP].

deprecated: Optional per-endpoint deprecation metadata. See Section 6.4.

6.4. Endpoint Deprecation

The optional deprecated block follows the same shape as catalog-level deprecation:

```
"deprecated": {  
  "deprecated_in": "2.1.0",  
  "removed_in": "3.0.0",  
  "successor": { "method": "RESERVE", "path": "/rooms" }  
}
```

Field semantics:

deprecated_in: The server's agtp_api_version (or operator-chosen version string) at which the endpoint became deprecated.

removed_in: Optional. The version at which the endpoint will be removed. ***MAY*** be absent if removal is anticipated but the target version has not been chosen.

successor: Optional. An object with method and/or path indicating the replacement endpoint callers should migrate to. Either field ***MAY*** be absent: successor.method alone signals "use the same path with a different method"; successor.path alone signals "use the same method with a different path"; both together signal a full replacement endpoint.

An endpoint ***MAY*** be deprecated even when its method and path remain valid in the catalog and grammar. Operators deprecate endpoints when they migrate callers from one (method, path) pair to another, independent of catalog evolution.

The dispatcher ***MUST*** stamp an AGTP-Endpoint-Warning advisory header on every response from a deprecated endpoint:

AGTP-Endpoint-Warning: deprecated; successor=RESERVE /rooms; removed_in=3.0.0

The header rides alongside AGTP-Catalog-Warning (Section 4) when both apply; both are advisory and the request still processes. The successor token renders as METHOD /path when both fields are present;

just one is emitted bare. `removed_in` and `successor` are omitted from the header when the endpoint's deprecated block does not declare them. The header fires on success and failure responses alike so callers always see the warning regardless of how the invocation resolves.

7. The Semantic Block

The semantic block is the part of the endpoint definition that encodes machine-readable intent. Agents reason about whether an endpoint matches their goal by interpreting the semantic block, not by parsing the verb and path alone. The empirical evidence for intent-expressing semantic metadata is documented in [HOOD2026].

7.1. Required Semantic Block Fields

intent: A natural-language statement of what the endpoint accomplishes. Single sentence. Imperative form. Subject is the actor; the verb reflects the endpoint's method; the object is the path resource.

actor: A short identifier of the agent class that should invoke this endpoint. Freeform string; servers **MUST NOT** reject endpoints whose actor is not in a fixed enum. Suggested values for interoperable authoring include `agent`, `human`, `system`, `customer`, `staff`, `admin`; domain-specific values (e.g., `merchant`, `auditor`, `patient`) are equally valid. The field helps agents recognize whether an endpoint applies to their role but carries no protocol-level enforcement.

outcome: A natural-language statement of the post-condition produced by successful invocation. Distinct from `intent`: `intent` is what the endpoint sets out to do; `outcome` is what is true after.

capability: The high-level effect class of the endpoint. One of the catalog category values: `discovery`, `retrieval`, `analysis`, `transaction`, `modification`, `creation`, `notification`, `mechanics`, `domain_spanning`. The same taxonomy used to classify methods in the catalog (Section 3); an endpoint inherits its capability class from the category of the method it exposes, though operators **MAY** override when the endpoint's specific behavior fits a different category.

confidence: A scalar on the closed interval 0.0 to 1.0 expressing how confident the server is that an agent invoking this endpoint with satisfying input will produce the declared outcome. Endpoints that are deterministic and well-tested score near 1.0; endpoints that involve nondeterministic resources or partial

failures score lower. Agents **MAY** use the value to decide whether to ESCALATE for clarification before invoking. The scale is advisory and not normatively defined beyond the endpoint bounds.

impact: The reversibility of the endpoint's effect. One of informational, reversible, irreversible. informational endpoints return data without changing server state. reversible endpoints change state in ways that can be undone via a compensating method (e.g., CANCEL, REFUND). irreversible endpoints commit changes that cannot be programmatically undone (e.g., sending an email, charging a payment, shipping goods). Governance frameworks **MAY** use this field to apply differential authorization, gating irreversible endpoints behind stricter scope checks.

is_idempotent: Boolean. Whether repeated invocation with identical input produces identical effect. Affects retry safety: idempotent endpoints **MAY** be retried freely; non-idempotent endpoints require an Idempotency-Key header per [AGTP] to be safely retried.

8. Server Manifest

8.1. Purpose

The server manifest is the machine-readable document an AGTP server publishes to declare its identity, version posture, endpoint catalog, hosted agents, hosted protocols, and operational policies. It is the authoritative answer to "what does this server offer."

8.2. Manifest Structure

```
{
  "agtp_version": "1.0",
  "agtp_api_version": "1.0",
  "document_version": "v2",
  "catalog_version": "1.0.0",
  "catalog_versions_supported": ["1.0.0"],

  "server": {
    "server_id": "agents.acme.com",
    "domain": null,
    "operator": "Acme Retail",
    "contact": "ops@acme.com",
    "supported_features": ["endpoint-registry", "synthesis"],
    "issued": "2026-01-15T09:00:00Z",
    "updated": "2026-04-15T09:00:00Z"
  },

  "embedded_methods": [ /* twelve floor methods */ ],
  "custom_methods": [ /* server-specific methods declared in policies.methods */ ],
  "endpoints": [ /* array of endpoint definitions, projected */ ],

  "agent_disclosure": "public",
  "hosted_agents": [ /* agent identity references */ ],
  "agent_disclosure_notice": null,

  "apis": [ /* legacy HTTP APIs this server fronts, if any */ ],
  "hosted_protocols": [ /* MCP, A2A, ACP, etc. */ ],

  "policies": {
    "wildcards_accepted": false,
    "anonymous_discovery": true,
    "scope_required_for_invocation": true,
    "synthesis_enabled": true,
    "max_synthesis_depth": 10
  },

  "manifest_signature": null
}
```

The manifest carries three distinct version fields at the top level:

agtp_version: The AGTP wire-protocol version the server speaks (currently "1.0", corresponding to AGTP v07 of [AGTP]). Pinned to the base protocol, separate from contract-layer evolution.

agtp_api_version: The AGTP-API contract-layer version the server

implements (currently "1.0" for v00 of this document). Carries the endpoint primitive, semantic block, status code, and synthesis semantics the server adheres to.

`document_version`: This specific manifest's revision string. Operator-controlled; bumped on every meaningful change to the manifest content. Permits agents to detect cache staleness without comparing the full body.

`catalog_version`: The method catalog version the server validates incoming method names against. See Section 4.

`catalog_versions_supported`: List of catalog versions the server can validate against. The current `catalog_version` **MUST** appear in this list. Multi-version support is anticipated in a future revision but is not normatively specified in v00.

8.3. Server Block

The server block carries operator-facing identity and timestamps:

`server_id`: The canonical identifier for this server. **MAY** be a hostname, an `agtp://` URI, or an operator-defined string. The value **MUST** be stable across manifest revisions.

`domain`: Optional operational host the server is reachable at, if different from `server_id`. **MAY** be null for servers whose `server_id` is itself a resolvable host.

`operator`: The human-readable name of the organization operating the server.

`contact`: An operator email address for security and operational issues.

`supported_features`: A non-normative array of feature tokens describing optional capabilities the server exposes (e.g., "endpoint-registry", "synthesis", "openapi-export"). Agents and operators **MAY** use these tokens for capability negotiation. AGTP-API does not define a canonical feature-token registry; tokens are operator-defined.

`issued`: ISO 8601 timestamp of when the manifest was first issued. Preserved across manifest revisions; identifies the manifest's origin in time.

`updated`: ISO 8601 timestamp of the most recent manifest update. Bumped on every regeneration.

8.4. Methods Inventory

The `embedded_methods` array carries the twelve protocol-floor methods every AGTP server supports (see [AGTP] §6). The `custom_methods` array carries any server-specific methods declared in `policies.methods` (Section 3.3). The `custom_methods` field *MAY* be omitted when empty.

These arrays are non-normatively populated; agents that want a complete picture of what the server invokes *SHOULD* consult the `endpoints` array, which is the authoritative source for what methods are bound on what paths.

8.5. Agents Disclosure

The three agent-related fields describe what agents the server hosts and how they are exposed:

`agent_disclosure`: One of "public", "authenticated", "private".
"public" means the `hosted_agents` array is populated and visible to all callers. "authenticated" means the array is populated only for callers presenting a verified identity. "private" means the array is empty in this manifest; the server does not disclose what agents it hosts.

`hosted_agents`: Array of agent identity references the server hosts.
Format is operator-defined; typical entries include a canonical Agent-ID and a brief description.

`agent_disclosure_notice`: Optional human-readable note explaining the server's disclosure policy. Useful when `agent_disclosure` is "authenticated" or "private" and the operator wishes to clarify how a caller can request elevated disclosure.

8.6. Protocol and API Surface

`apis`: Optional array of legacy HTTP API descriptors this server fronts (e.g., entries pointing at OpenAPI specifications). Servers that expose a parallel HTTP surface alongside AGTP list those HTTP endpoints here; the presence of entries in this array is the signal that the server operates a parallel HTTP face.

`hosted_protocols`: Optional array of higher-level agent protocols the server hosts on top of AGTP (e.g., MCP, A2A, ACP). Each entry is an object with the protocol name and the AGTP method+path it is reachable via. See the composition section in [AGTP].

8.7. Policies Block

The policies block carries five normative operational toggles:

`wildcards_accepted`: One of true, false. Whether the server accepts ad-hoc method invocations against undeclared endpoints. See Section 5.8.

`anonymous_discovery`: One of true, false. Whether unauthenticated callers may retrieve the manifest and invoke read-only built-in endpoints. Servers serving traffic from open networks typically set this to true; servers in trusted-network deployments **MAY** set this to false to require authenticated identity even on discovery.

`scope_required_for_invocation`: One of true, false. Whether every endpoint invocation requires the agent to carry an Authority-Scope header. true is the default; the implementation **MUST NOT** invoke a handler when scope is required and absent. Servers with no scope-gated endpoints **MAY** set this to false.

`synthesis_enabled`: One of true, false. Whether the server participates in PROPOSE-and-synthesis runtime contract negotiation. When false, PROPOSE requests **MUST** be refused with 463 Proposal Rejected and the reason synthesis-disabled.

`max_synthesis_depth`: A positive integer specifying the maximum number of composed steps a synthesized endpoint **MAY** combine. Plans exceeding this depth are refused; the server falls through to counter-proposal or 463. Default: 10.

8.8. Manifest Signature

The manifest body **SHOULD** carry a `manifest_signature` field holding a JWS envelope over the canonical JSON encoding of the manifest body (with the `manifest_signature` field itself excluded from the signing input). Clients **MAY** verify signatures when present and **SHOULD** warn when absent.

A future revision of this specification will require `manifest_signature` (**MUST**); reference implementations are expected to ship signing support in alignment with that revision. The signing algorithm, canonical encoding rules, and key-publication mechanism are specified separately in [AGTP-CERT].

For v00, the `manifest_signature` field **MAY** be null or absent; a manifest without a valid signature **MUST NOT** be rejected by clients on signature grounds alone.

8.9. Endpoint Projection

The endpoints array in the manifest is a projection of the server's internal endpoint catalog. The projection differs from the full endpoint definition in one important way: the handler's type-specific reference field (function, recipe, or url) and any other binding-specific implementation details are dropped, leaving only the binding kind under handler.type.

Internal endpoint definition (on disk, in TOML or equivalent):

```
{
  "method": "BOOK",
  "path": "/room",
  "description": "...",
  "semantic": { ... },
  "input_schema": { ... },
  "output_schema": { ... },
  "errors": [ ... ],
  "required_scopes": [ ... ],
  "handler": {
    "type": "registered_function",
    "function": "rooms.handlers.book_room"
  }
}
```

Manifest projection (over the wire):

```
{
  "method": "BOOK",
  "path": "/room",
  "description": "...",
  "semantic": { ... },
  "input_schema": { ... },
  "output_schema": { ... },
  "errors": [ ... ],
  "required_scopes": [ ... ],
  "handler": { "type": "registered_function" }
}
```

The type-specific reference field (function, recipe, or url) and any other binding-specific implementation fields are server-internal and ***MUST NOT*** appear in the manifest projection. Agents need to know the binding kind to reason about expected latency, retry behavior, and attribution; agents do not need to know which specific function, recipe, or upstream realizes the binding.

Servers ***MUST*** apply this projection on every manifest response.

8.10. Manifest Retrieval

Agents retrieve the server manifest using the AGTP DISCOVER method without an Agent-ID header. The absence of an Agent-ID header distinguishes a server-level discovery request (returns the manifest) from an agent-level discovery request (returns endpoint candidates matching specified criteria).

The response *MUST* carry Content-Type application/vnd.agtp.manifest+json. The manifest *SHOULD* be signed; see the Manifest Signature subsection above.

8.11. Manifest Caching

Servers *SHOULD* include Cache-Control and ETag headers on manifest responses. Agents *MAY* cache manifests according to those directives. The server.updated timestamp and the document_version field permit agents to detect cache staleness without revalidation: a changed document_version means the manifest content has changed in a way the operator considers significant.

9. Method Policy

9.1. Purpose

The method policy declares per-server constraints on which methods are accepted, which legacy verbs are opted in, and which methods on which paths are redirected to their canonical form. The policy is the small, declarative complement to the endpoint catalog: the catalog says what an endpoint does; the policy says what the server accepts in the first place.

In v00, the method policy is a sub-block of the manifest's policies (Section 8) carried at policies.methods. There is no separate policy file format; the policy is part of the server's configuration and is exposed on the manifest like any other operational policy.

9.2. Policy Shape

The policies.methods block is a JSON object with four optional fields:

allow: Either the string "*" (the server admits every method in the catalog plus any method it has registered an endpoint for), or an array of method names (the server admits only the listed methods plus the embedded twelve floor methods). Default: "*".

disallow: Array of method names the server explicitly rejects.

Disallowed methods are rejected with 405 Method Not Allowed even when they would otherwise be permitted by allow. Useful for narrowing the surface without enumerating an exhaustive allow list. Default: empty array.

legacy: Either an array of legacy HTTP verb names the server opts into (e.g., ["GET", "POST"]), the string "*" (all five legacy verbs accepted), or the string "NONE" (no legacy support). Granular per-verb opt-in supersedes the binary enabled/disabled toggle. Default: "NONE".

redirects: Array of method-and-path redirect entries. Each entry has the shape:

```
{
  "from_method": "BOOK",
  "from_path": "/room",
  "to_method": "RESERVE",
  "to_path": "/room"
}
```

Either from_path or to_path *MAY* be omitted, signaling "redirect on method-only without path matching" or "redirect to the same path as the request." When a request matches a redirect entry, the server processes it as if the to_method/to_path had been requested instead.

9.3. Example

A representative TOML deployment configuration:

```
[policies.methods]
allow = "*"
disallow = ["PATCH", "TRANSFER"]
legacy = ["GET"]

[[policies.methods.redirects]]
from_method = "BOOK"
from_path = "/room"
to_method = "RESERVE"
to_path = "/room"
```

The equivalent manifest projection:

```
"policies": {  
  "methods": {  
    "allow": "*",  
    "disallow": ["PATCH", "TRANSFER"],  
    "legacy": ["GET"],  
    "redirects": [  
      {  
        "from_method": "BOOK",  
        "from_path": "/room",  
        "to_method": "RESERVE",  
        "to_path": "/room"  
      }  
    ]  
  }  
}
```

9.4. Catalog-Graceful Skip Semantics

When the server loads `policies.methods` and an entry under `allow`, `disallow`, or `redirects` references a method removed from the catalog version the server has loaded, the entry *MUST* be skipped at load time with an operator-visible warning. The server continues to operate; the policy entry has no effect. This permits operators to upgrade the catalog without simultaneously rewriting the policy block to remove all references to retired methods.

The `legacy` field continues to enforce the strict set of five HTTP verbs (GET, POST, PUT, DELETE, PATCH); typos or non-HTTP verbs in `legacy` *MUST* be rejected as a hard configuration error.

9.5. Discovery

Agents discover the server's method policy by retrieving the server manifest via target-less DISCOVER (Section 8). The `policies.methods` block is part of the manifest body and is covered by the manifest signature (Section 8) when present.

Agents that need only an endpoint inventory (which methods on which paths are exposed, with one-line descriptions) *MAY* invoke DISCOVER /methods (Section 5.8) for a lightweight response that bypasses the full manifest weight.

10. PROPOSE and Synthesis

10.1. Purpose

PROPOSE is the AGTP method that triggers runtime contract negotiation. An agent that needs an endpoint the server does not advertise submits a PROPOSE request carrying an AGTP-API-conformant endpoint specification; the server evaluates the proposal against the contract layer and either instantiates the endpoint as a session-scoped artifact or refuses.

PROPOSE is one of the twelve floor methods defined in [AGTP]. Its parameter table is specified there. This document specifies the synthesis semantics that govern server-side evaluation.

10.2. Negotiation Flow

- Step 1: Agent retrieves the server manifest (which carries the method policy as 'policies.methods')
- Step 2: Agent determines the needed endpoint is not declared
- Step 3: Agent constructs an AGTP-API-conformant endpoint definition
- Step 4: Agent sends PROPOSE with the endpoint definition in body
- Step 5a: Server accepts and synthesizes
 - Server evaluates contract conformance and capability envelope
 - Server composes the endpoint from existing endpoints
 - Server returns 263 Proposal Approved with the
 - AGTP-API endpoint definition for the synthesized endpoint
 - Synthesis-ID matches original proposal
 - Synthesized endpoint is session-scoped by default
 - Agent MAY invoke the endpoint immediately
- Step 5b: Server rejects (463)
 - Server returns 463 Proposal Rejected with structured reason
 - Response SHOULD reference manifest entries that satisfy a similar capability if available
 - Agent MAY modify proposal and retry (maximum 3 turns)
 - After 3 rejections agent MUST ESCALATE

10.3. Server-Side Synthesis Evaluation

On receipt of a PROPOSE request, the server **MUST** evaluate:

1. **Method conformance.** Is the proposed method in the AGTP-API approved method catalog, or is it a custom method declared in the server's policies.methods (Section 3.3)? If not, return 459 Method Violation.
2. **Path conformance.** Does the proposed path satisfy Section 5? If not, return 460 Endpoint Violation.

3. **Semantic block conformance.** Does the proposed semantic block carry all required fields with valid values? If not, return 400 Bad Request.
4. **Schema conformance.** Are the input and output schemas valid JSON Schema documents? If not, return 400 Bad Request.
5. **Authority sufficiency.** Does the proposing agent's declared Authority-Scope cover the proposed authority requirements? If not, return 262 Authorization Required with body type scope-required.
6. **Capability envelope.** Can the server actually synthesize the proposed endpoint from its existing capabilities? If not, return 463 Proposal Rejected with reason: composition-impossible.
7. **Policy acceptance.** Does the server's policy permit the proposed method-and-path combination? If not, return 463 with reason: policy-refused.

If all evaluations pass, the server synthesizes the endpoint and returns 263 Proposal Approved.

10.4. Synthesis-Composed Endpoints

When a server synthesizes an endpoint, the server is composing the proposed behavior from existing endpoints internally. The synthesized endpoint is exposed to the agent as a single endpoint, but the server's handler implementation invokes one or more underlying endpoints to fulfill the request.

Authority **MUST** be preserved through the composition. The agent's Authority-Scope **MUST** be sufficient for every underlying endpoint the synthesis invokes. If a composed step would require authority the agent does not hold, the synthesis **MUST** fail at evaluation time (step 5) rather than at execution time.

10.5. Session-Scoped vs Persistent Synthesis

By default, a synthesized endpoint is session-scoped: it exists for the duration of the AGTP session that proposed it and is removed when the session ends. The agent **MAY** request persistent synthesis by setting `persistent: true` in the PROPOSE body; the server **MAY** honor or refuse the persistence request independently of the synthesis decision. Persistent synthesis subjects the proposed endpoint to the server's regular endpoint catalog and is reflected in subsequent manifest retrievals.

11. Status Codes

AGTP-API contributes the following structural rejection codes to the AGTP status code space. The codes are registered with IANA per [AGTP] Section 9.3.

Code	Name	Returned When
261	Negotiation In Progress	Server has accepted PROPOSE for evaluation; agent polls via QUERY /proposals/{proposal_id} for terminal status
262	Authorization Required	Request requires credential establishment, additional Authority-Scope, wildcards consent, or authenticated identity that is not yet present
263	Proposal Approved	Server has synthesized and instantiated the proposed endpoint
404	Not Found	The path does not exist on this server
405	Method Not Allowed	Method and path are valid but server policy does not expose this combination
459	Method Violation	The method is not in the AGTP-API approved catalog
460	Endpoint Violation	The path violates AGTP-API path grammar (method-name leakage)
463	Proposal Rejected	Server cannot or will not synthesize the proposed endpoint

Table 1: AGTP-API Status Codes

The four structural rejection codes 404, 405, 459, 460 form the contract-level failure surface. Each code is independently actionable and addresses a distinct failure mode:

- * ***404*** -- the path does not exist. The agent has the wrong resource locator.
- * ***405*** -- the method and path are individually valid but the server does not expose this combination. The agent ***MAY*** try a different method on the same path, or ***MAY*** PROPOSE the combination if it is not a policy refusal.
- * ***459*** -- the method itself is the problem. The agent ***MUST*** pick a different method from the AGTP-API catalog. PROPOSE will not succeed because the method name is not legal.
- * ***460*** -- the path violates path grammar. The agent ***MUST*** restructure the path so that no path segment matches an approved method name.

These codes are not interchangeable. A 405 is a policy decision; a 459 is a vocabulary violation; a 460 is a structural violation. Servers ***MUST*** return the most specific code that applies.

The 405 response body ***MUST*** carry the following fields:

allowed_methods_for_path: The list of methods the server's policy or registry exposes on the path. Permits the agent to retry with a method the server actually accepts.

redirects_for_path: A map of method-to-method redirects from policies.methods (Section 9) that apply to this path. ***MAY*** be empty if the server has no redirects configured. Permits the agent to learn that, for example, BOOK is redirected to RESERVE on this server, and to retry under the canonical method directly.

These fields together let the agent recover from a 405 without an out-of-band catalog lookup or a second round-trip.

12. Handler Binding

The handler binding declares how an endpoint's behavior is realized at runtime. AGTP-API defines three binding types.

12.1. Composition

A composition handler exposes a derived endpoint whose behavior is realized by orchestrating one or more other endpoints declared on the same server. No application code is invoked; the server's synthesis runtime executes the composition steps in order, threading outputs into subsequent inputs while preserving authority through the composition.

Composition handlers reference a named recipe:

```
"handler": {  
  "type": "composition",  
  "recipe": "audit-via-query-and-summarize"  
}
```

The recipe value is the name of a recipe defined in the server's recipe registry (typically in an `agtp-recipes.toml` file loaded at server startup). Recipes are reusable units of composition logic; one recipe *MAY* back multiple endpoints. Inline composition steps are not supported in v00; every composition endpoint *MUST* reference a registered recipe by name.

A composition handler whose recipe value does not resolve to a recipe registered at the server *MUST* cause registration to fail.

Composition endpoints *MUST* include `composition_failed` in their errors list. This is the named error returned when a composition step fails (any underlying step returns a non-success status). The error response carries details with the recipe identifier, the failed step number, the underlying step method, the underlying status, and any captured outputs from preceding steps.

Authority *MUST* be preserved through the composition. The agent's Authority-Scope *MUST* be sufficient for every underlying endpoint the composition invokes. Authority sufficiency is verified at registration time when possible (when step paths are statically known) and at invocation time for parameterized step paths.

This is the binding type most commonly produced by PROPOSE + synthesis: a synthesized endpoint is a composition of existing endpoints, and *MAY* be promoted to a permanent declared composition by adding its recipe to the server's recipe registry and writing an endpoint TOML that references it by name.

12.2. Registered Function

```
"handler": {  
  "type": "registered_function",  
  "function": "rooms.handlers.book_room"  
}
```

The endpoint binds to a Python function loaded by the server. The function value is a dotted path identifying the module and the callable within it; the server imports the module at registration time and resolves the function attribute. Single dotted-path form is normative; separate module and function fields are not supported in v00.

The function receives the validated input and returns the response payload. The handler API contract is specified separately (see the reference implementation's handler documentation). This is the binding type used for endpoints whose logic exceeds what composition can express.

12.3. External Service

```
"handler": {
  "type": "external_service",
  "url": "https://internal.acme.tld/booking",
  "method": "POST",
  "headers": {
    "Authorization": "Bearer ${BOOKING_API_TOKEN}",
    "Content-Type": "application/json"
  },
  "input_transform": {
    "guest_id": "guestId",
    "room_type": "roomType"
  },
  "output_transform": {
    "confirmation_code": "confirmationNumber"
  },
  "error_map": {
    "404": "room_not_found",
    "409": "room_unavailable"
  },
  "timeout_seconds": 30
}
```

The endpoint binds to an HTTPS-reachable external service. The server proxies the request to the external endpoint and returns the response. Useful for AGTP frontends to existing HTTP-based services. The external service is not part of the AGTP contract; the AGTP server presents a contract-compliant facade.

External service fields:

url: The fully-qualified URL of the upstream HTTP endpoint. The URL **MUST** use the https:// scheme. Plain HTTP **MUST NOT** be accepted at registration time.

method: The HTTP method to invoke on the upstream. One of GET, POST, PUT, PATCH, DELETE, HEAD, OPTIONS. **REQUIRED**.

headers: Optional map of HTTP headers to send to the upstream on every request. Values **MAY** include `${VAR}` placeholders that are resolved against process environment variables at server startup. Unresolved placeholders **MUST** cause registration to fail.

input_transform: Optional map renaming AGTP request fields to the upstream's expected field names. Maps AGTP-side name to upstream-side name. Fields not in the map pass through unchanged. v00 supports key-rename only; value transformation is anticipated in a future revision.

output_transform: Optional map renaming upstream response fields to AGTP-side names. The map is inverted at response time: the upstream-side name is the source, the AGTP-side name is the target. Fields not in the map pass through unchanged.

error_map: Map of upstream HTTP status code (as a string) to AGTP error code declared in the endpoint's errors list. Servers **SHOULD** configure `error_map` for every `external_service` binding; without it, upstream HTTP errors surface to agents as generic `upstream_error`. Every value in `error_map` **MUST** appear in the endpoint's errors; registration **MUST** fail otherwise.

timeout_seconds: Positive number specifying the upstream request timeout. Servers **SHOULD** set this explicitly. Default if absent: 30 seconds. Production deployments **SHOULD NOT** rely on the default; explicit timeouts make upstream-dependency posture inspectable.

External service endpoints **MUST** declare the following five upstream-failure error codes in their errors list, in addition to any business-level errors:

Error Code	Returned When
upstream_timeout	Upstream did not respond within timeout_seconds
upstream_connection_error	DNS, connection, or TLS failure reaching the upstream
upstream_malformed_response	Upstream returned a 2xx response whose body is not valid JSON
upstream_authentication_failed	Upstream returned 401 or 403 not covered by error_map
upstream_error	Upstream returned 4xx or 5xx not covered by error_map

Table 2: Mandatory External Service Error Codes

Without these declarations, transport-level upstream failures would surface to the calling agent as opaque server errors. Mandating them makes the failure surface inspectable.

Agent identity is **NOT** propagated to the upstream by default. This is a security default: an upstream HTTP service has no mechanism to validate AGTP agent credentials, and forwarding them would expose principal identity to systems outside the AGTP trust boundary. Servers wishing to propagate identity to a specific upstream **MAY** do so via explicit header configuration in the headers map (e.g., a server-issued bearer token), but **MUST NOT** forward AGTP-Agent-ID or AGTP-Principal-ID by default.

13. Conformance

13.1. Single Tier

A conforming AGTP server implements:

1. The full endpoint primitive (Section 6).
2. Method validation against the AGTP-API approved catalog at the server's declared catalog_version.
3. Path validation against the AGTP-API path grammar.

4. Method policy enforcement at dispatch time per the `policies.methods` sub-block of the manifest (Section 9).
5. A server manifest retrievable via DISCOVER without an Agent-ID header, carrying `agtp_version`, `agtp_api_version`, `document_version`, `catalog_version`, and `catalog_versions_supported` per Section 8.
6. The endpoint projection of Section 8.9: manifest responses expose `handler.type` only, not the full handler binding.
7. PROPOSE handling with the synthesis evaluation flow specified in this document, gated by `policies.synthesis_enabled`.
8. The structural rejection status codes 404, 405, 459, 460, plus the contract-negotiation codes 261, 262, 263, 463.
9. The semantic block on every published endpoint.
10. Schema validation for input and output, with the strictness asymmetry of Section 13.3 (inputs strict, outputs permissive by default).
11. The AGTP-Catalog-Warning response header on every response whose method is deprecated in the catalog version in use.
12. The AGTP-Endpoint-Warning response header on every response from a deprecated endpoint (Section 6.4).
13. The DISCOVER `/methods` built-in endpoint defined in Section 5.8, returning a compact inventory of the server's registered endpoints.

There is no halfway tier. Servers that implement only method dispatch without endpoints are not AGTP-conformant; they are something else, perhaps a method-based RPC layer that happens to use AGTP framing.

13.2. Validation Timing

Conforming servers perform two distinct categories of validation, which **SHOULD** occur at different times.

Manifest validation at startup. Before accepting traffic, a conforming server **SHOULD** validate every endpoint declared in its manifest:

- * The method appears in the AGTP-API approved catalog (or is a custom method declared in the server's policies.methods).
- * The path conforms to the path grammar of Section 5.
- * The semantic block carries every required field.
- * The input and output schemas are syntactically valid JSON Schema documents.
- * The handler binding (Section 12) is resolvable: a composition's referenced steps exist; a registered_function's module and function are importable; an external_service's URL is well-formed and reachable (or marked deferred).

A server that detects manifest validation failures at startup **SHOULD** reject the offending endpoints with clear logs and **SHOULD NOT** silently ignore them. Servers **MAY** choose to fail startup entirely on manifest validation failure or to start serving with the valid subset of endpoints; the choice is a deployment policy decision.

Request validation at runtime. Once serving, the server validates each inbound request against the endpoint's input schema before invoking the handler:

- * Input body validates against the endpoint's input_schema. On failure, the server returns 422 Unprocessable Entity with a schema validation error report.
- * The agent's Authority-Scope covers the endpoint's required_scopes declarations. On failure, the server returns 455 Scope Violation per [AGTP].
- * The verb-and-path combination is exposed by the server's current policy. On failure, the server returns the appropriate structural rejection code (404, 405, 459, or 460).

Validation **MUST** complete before handler invocation. Servers **MUST NOT** invoke the handler with input that has not satisfied schema validation; this guarantee is what allows handler authors to assume their context structure is well-formed.

The startup-vs-runtime split is a recommendation, not a strict requirement. Servers **MAY** defer startup validation if their endpoint catalog is dynamic (e.g., a PROPOSE-driven server that only synthesizes endpoints on demand). Such servers **MUST** still validate at the moment an endpoint is registered or synthesized, just not at process start.

13.3. Schema Strictness

Schema validation is asymmetric between inputs and outputs.

Input schemas **MUST** be validated with `additionalProperties: false`. Fields not declared in the schema **MUST** cause validation to fail with 422. This catches typos and version mismatches early and prevents agents from sending unrecognized fields under the mistaken assumption that the server will silently ignore them.

Output schemas **SHOULD** be validated with `additionalProperties: true`. A handler that returns more fields than the schema declared is forward-compatible: agents can ignore extra fields. A handler that returns missing required fields fails validation either way. Servers **MAY** opt into strict output validation as a deployment policy.

This asymmetry is normative because it has interoperability consequences. An agent submitting a request to two different servers can rely on both rejecting unrecognized input fields. An agent consuming a response from two different servers can rely on forward-compatible field handling without the version of one server breaking integration with the other.

13.4. Conformance Test Suite

The AGTP-API conformance test suite is maintained at:

<https://agtp.io/api/conformance>

The test suite exercises every conformance requirement in this section. Servers claiming AGTP-API conformance **MUST** pass the test suite at the version they declare in their manifest's `agtp_api_version` field.

14. Relationship to OpenAPI

OpenAPI and AGTP-API perform similar work for different consumers. OpenAPI describes HTTP APIs for human developers and code generators. AGTP-API describes agent-grade endpoints for autonomous agents. The two formats overlap substantially: both have method, path, request schema, response schema, errors, and descriptions. AGTP-API does not compete with OpenAPI; the two formats are complementary, and AGTP-API deployments commonly interoperate with OpenAPI-described HTTP services in three distinct ways.

14.1. OpenAPI as an Authoring Source

A developer with an existing OpenAPI specification can convert it to AGTP-API endpoint declarations. The HTTP method maps to an AGTP verb (with translation: GET maps to QUERY or DISCOVER depending on intent; POST maps to EXECUTE, BOOK, PURCHASE, or another action verb depending on the operation). The path remains. The request schema becomes input_schema. The response schema becomes output_schema. Errors become the errors list. The OpenAPI description becomes the foundation of the semantic block's intent.

Conversion tooling **MAY** automate this translation with reasonable defaults, flagging cases where verb mapping requires human judgment. The AGTP-API approved method catalog (Section 3) is the canonical source for HTTP-verb-to-AGTP-verb conversions: tooling **MUST** select target verbs from the list rather than inventing ad-hoc names, which is what makes converted endpoints interoperable with native AGTP-API endpoints.

Servers **SHOULD NOT** treat machine-converted endpoints as authoritative until a human reviewer has confirmed the semantic block fields, particularly intent, outcome, and impact. Conversion tooling has no reliable way to infer impact from HTTP semantics; that is judgment work.

14.2. OpenAPI as an Export Target

AGTP servers **MAY** render their endpoint catalog as an OpenAPI document for HTTP consumers that do not yet speak AGTP. The reverse translation: AGTP verb to HTTP method (mappings are imperfect; some AGTP verbs have no clean HTTP analog and are best exposed via the legacy verb compatibility mode of Section 3.6); path remains; schemas pass through unchanged.

The exported OpenAPI document is a compatibility surface, not the authoritative description of the service. Servers exporting OpenAPI **SHOULD** mark the export as derived from the AGTP-API manifest and **SHOULD** publish a link to the manifest in the OpenAPI info section.

14.3. OpenAPI as a Coexistence Layer

A server **MAY** run AGTP and HTTP side by side. The AGTP endpoint's handler is the same code that the HTTP endpoint invokes. OpenAPI documents the HTTP face; the AGTP-API manifest documents the agent face. The handler does not know which protocol invoked it. This deployment posture is signaled in the manifest by populating the `apis` array (Section 8) with descriptors pointing at the server's OpenAPI documents.

Coexistence is the path most large organizations will take during transition: AGTP is added as a parallel face without requiring modification to existing HTTP infrastructure. Implementation guidance for coexistence deployments is provided in Appendix A.

15. What AGTP-API Does Not Specify

AGTP-API explicitly excludes the following operational concerns, which remain middleware territory:

- * Rate limiting strategies and policies.
- * Observability mechanisms (metrics, logs, traces).
- * Caching strategies beyond manifest caching.
- * Custom authentication beyond AGTP agent identity.
- * Request and response transformation.
- * Server deployment topology and scaling.
- * HTTP middleware integration patterns.

These are addressed by reference middleware repositories that ship separately and non-normatively. The contract-plumbing distinction is intentional: the protocol governs contracts; middleware governs operations.

16. Security Considerations

16.1. Verb-Path Tampering

A malicious server could declare endpoints whose semantic blocks misrepresent the actual behavior. Agents that rely on semantic blocks for authorization decisions **MUST** also enforce Authority-Scope checks against actual outcomes, not declared outcomes.

16.2. Synthesis Authority Preservation

The synthesis evaluation step (step 5 in Section 10.3) **MUST** verify that authority is preserved through composition. A synthesis that elevates authority by composing endpoints with sufficient individual scopes into a behavior the agent should not be permitted is a known threat. Servers **MUST** reject syntheses where the composed behavior exceeds what the agent's Authority-Scope permits, regardless of whether the individual steps are within scope.

16.3. Method Policy Tampering

The method policy is carried as the policies.methods sub-block of the server manifest and is covered by the manifest signature (Section 8) when present. Agents that retrieve the manifest over AGTP transport (which mandates TLS 1.3 or higher per [AGTP]) and verify the manifest signature when present are protected against policy substitution. Agents that accept manifest content without signature verification are vulnerable to substitution of the method policy along with the rest of the manifest.

16.4. Wildcard Abuse

The wildcards: true consent on the agent identity document and wildcards_accepted: true on the server policy are mutual consents that enable ad-hoc method invocation. Agents **SHOULD NOT** declare wildcards: true by default; servers **SHOULD NOT** declare wildcards_accepted: true for any operation with non-trivial impact tier. The 262 Authorization Required response with body type wildcards-required is the safe default.

16.5. Verb List Trust

Agents trust the AGTP-API approved method catalog as published at <https://agtp.io/api/methods.json>. The catalog **MUST** be served over HTTPS with content integrity verification. Agents **MAY** pin the catalog version they have validated and refuse upgrades without explicit operator approval.

17. IANA Considerations

This document defines the following AGTP-API-specific allocations. The corresponding AGTP IANA registry entries are recorded in [AGTP] Section 9.

17.1. AGTP-API Status Code Assignments

The following codes are registered in the AGTP Status Code Registry with AGTP-API as the authoritative reference:

- * 261 Negotiation In Progress
- * 262 Authorization Required
- * 263 Proposal Approved
- * 405 Method Not Allowed (AGTP-API semantics)
- * 459 Method Violation
- * 460 Endpoint Violation
- * 463 Proposal Rejected

17.2. Media Type Registrations

This document defines two media types and requests their registration in the IANA Media Types registry. Full registration templates are provided in the appendices. The master registry of all AGTP-family media types is maintained in [AGTP].

Media Type	Use	IANA Status
application/ vnd.agtp.manifest+json	AGTP server manifest format	Planned (this document)
application/ vnd.agtp.endpoint+json	AGTP-API endpoint definition format	Planned (this document)

Table 3: Media Types Defined by AGTP-API

application/vnd.agtp.manifest+json The AGTP server manifest format. Carried on responses to a server-level DISCOVER request (DISCOVER without an Agent-ID header). The registration application will be filed concurrent with publication of this document.

application/vnd.agtp.endpoint+json The AGTP-API endpoint definition format. Used in PROPOSE request bodies and in 263 Proposal Approved response bodies. The registration application will be filed concurrent with publication of this document.

Until the IANA registrations complete, the reference implementation **MAY** continue to emit application/json for manifest and endpoint responses. Implementations are encouraged to emit the AGTP-specific types as soon as they are accepted by IANA so that content negotiation and MIME-based routing can distinguish AGTP traffic from generic JSON.

17.3. AGTP-API Method Catalog Reference

This document does **NOT** request establishment of an IANA AGTP-API Verb List Registry. As specified in Section 3.1, the verb list is intentionally maintained as an open living artifact at <https://agtp.io/api/methods.json> rather than as a closed IANA registry, so that the catalog can evolve at the cadence of agent deployment rather than the cadence of IETF process. The structural status codes, the media types, and the response headers below are appropriate for IANA registration; the method catalog is not.

17.4. AGTP-API Response Headers

This document requests registration of the following response headers in the IANA HTTP Field Name Registry (or equivalent AGTP-specific registry where applicable):

AGTP-Catalog-Warning: Advisory header surfaced on responses where the request method is deprecated in the catalog version the server is using. Format and semantics specified in Section 4.

AGTP-API-Version: Header surfacing the AGTP-API specification version a manifest or endpoint definition conforms to.

18. Open Items

The following items are explicitly out of scope for this revision and are anticipated in future revisions:

- * Mandatory manifest signing. v00 specifies manifest_signature as **SHOULD**; a future revision will make it **MUST**. The signing algorithm, canonical encoding rules for JWS over the manifest body, key-publication mechanism, and rotation semantics will be specified separately in [AGTP-CERT]. Reference implementations are expected to ship signing support in alignment with that revision.

- * Cross-endpoint shared schema definitions. v00 requires schemas to be inline in each endpoint's `input_schema` and `output_schema`. A future revision may add a `schemas` block to the manifest carrying named, reusable JSON Schema fragments referenceable by `$ref` from endpoint schemas. This is a manifest-size and authoring-convenience concern; the wire-level contract is unchanged.
- * Config-driven `document_version`. The reference implementation hardcodes `document_version` to a single value; an operator workflow that bumps `document_version` on every meaningful manifest change is anticipated. The wire contract permits any string; the operator workflow is implementation-specific.
- * The full text of the initial method catalog (approximately 435 methods). v00 references the published JSON at `agtp.io/api/methods.json` rather than embedding the catalog inline. A future revision may include a curated subset as a non-normative appendix; the authoritative catalog remains the published JSON per Section 3.1.
- * Detailed schema for the `synthesis_log` returned with 263 Proposal Approved, recording which underlying endpoints were composed.
- * Federation model for multi-server synthesis (composing endpoints from different servers in a single synthesized endpoint).
- * Multi-version concurrent catalog support. v00 specifies `catalog_versions_supported` as a list, but the runtime semantics for a server processing a request against a non-default catalog version are not specified. Most servers list a single version.
- * Conformance test suite specification (currently described by reference to the `agtp.io` implementation).
- * Authentication passthrough for external_service handlers (forwarding agent identity to upstream services). Out of scope by security default in v00; future revisions may specify opt-in passthrough patterns.
- * Hot reload of recipes, endpoints, and `policies.methods` configuration without server restart.
- * Wildcard opt-in implementation. The spec defines the consent model (`wildcards: true` on the agent identity document and `wildcards_accepted: true` in server policy) and the 262 Authorization Required response with body type `wildcards-required` for unsatisfied consent. The default behavior in v00 is `refuse`: ad-hoc method invocations against undeclared endpoints are

rejected as 459 Method Violation when the method is not in the AGTP-API catalog, or as 405 Method Not Allowed when the method is in the catalog but no endpoint exists for the method-and-path combination. The opt-in path that activates 262 with wildcards-required (when an agent or server has consented to wildcards but the other side has not) is anticipated as an implementation extension in a future revision.

19. Acknowledgments

This document was developed in response to implementation experience with the AGTP v06 and v07 builds. The architectural decision to unify the contract layer under a single companion specification was articulated in the AGTP-API Position Paper (May 2026). The empirical evidence for the action-intent semantic class is documented in [HOOD2026].

20. References

20.1. Normative References

- [AGTP] Hood, C., "Agent Transfer Protocol (AGTP)", Work in Progress, Internet-Draft, draft-hood-independent-agtp-07, 2026, <<https://datatracker.ietf.org/doc/html/draft-hood-independent-agtp-07>>.
- [JSON-SCHEMA] Wright, A., Andrews, H., Hutton, B., and G. Dennis, "JSON Schema: A Media Type for Describing JSON Documents", Work in Progress, Internet-Draft, draft-bhutton-json-schema-01, 2020, <<https://datatracker.ietf.org/doc/html/draft-bhutton-json-schema-01>>.
- [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>>.
- [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>>.

- [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>>.

20.2. Informative References

- [AGTP-CERT] Hood, C., "AGTP Agent Certificate Extension", Work in Progress, Internet-Draft, draft-hood-agtp-agent-cert-00, 2026, <<https://datatracker.ietf.org/doc/html/draft-hood-agtp-agent-cert-00>>.
- [AGTP-MERCHANT] Hood, C., "AGTP Merchant Identity and Agentic Commerce Binding", Work in Progress, Internet-Draft, draft-hood-agtp-merchant-identity-01, 2026, <<https://datatracker.ietf.org/doc/html/draft-hood-agtp-merchant-identity-01>>.
- [AGTP-TRUST] Hood, C., "AGTP Trust and Verification Specification", Work in Progress, Internet-Draft, draft-hood-agtp-trust-00, 2026, <<https://datatracker.ietf.org/doc/html/draft-hood-agtp-trust-00>>.
- [HOOD2026] Hood, C., "Semantic Method Naming and LLM Agent Accuracy: A Controlled Benchmark of REST/CRUD versus Agentic API Interface Design", Working Paper Available by request. March 2026., 2026.
- [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>>.

Appendix A. Implementation Guidance (Informative)

This appendix is non-normative. It describes how a typical AGTP-API server is built and operated, in support of the normative spec. Implementations are not required to follow these patterns; they are documented here to give developers a starting point and to make the intended deployment posture concrete.

A.1. Endpoint Authoring

A common pattern for endpoint authoring uses one declaration file per endpoint, dropped into a directory the server scans at startup:

- * A human-editable declaration format (such as TOML, YAML, or JSON) carries the endpoint's verb, path, semantic block, schemas, errors, and handler reference.
- * A handler function in the server's implementation language carries the business logic, with a uniform signature that takes a context object and returns a structured response or named error.
- * The server scans the endpoints directory at startup, validates every declaration per Section 13.2, imports the referenced handler modules, and registers the valid endpoints in memory.

The reference Python implementation of an AGTP-API server uses TOML for endpoint declarations and a context \rightarrow response | error handler signature. The exact TOML schema and handler patterns are documented at:

<https://agtp.io/api/reference/authoring>

The TOML format is one serialization of the endpoint primitive defined normatively in Section 6. Other implementations are free to use other serializations (YAML, JSON, programmatic registration, database-backed registries) provided the in-memory endpoint structure conforms to this specification.

A.2. Runtime Dispatch Pattern

A typical request lifecycle in a conforming server:

1. The server receives an AGTP request and parses the verb and path.
2. The dispatcher matches verb-and-path against the endpoint registry.
3. On match, the dispatcher validates the request body against the endpoint's input_schema.
4. The dispatcher checks the agent's Authority-Scope against the endpoint's required_scopes declaration.
5. The dispatcher constructs a context object containing validated input and agent identity, and invokes the handler.
6. The handler executes business logic and returns a structured response or named error.
7. The dispatcher translates the result into an AGTP response.

This pattern keeps contract enforcement in the dispatcher and business logic in the handler. Handlers do not perform schema validation, authority checks, or status code translation; the dispatcher does those uniformly across all endpoints.

A.3. Tooling

The reference implementation publishes tooling at agtp.io/api/tools:

- * An OpenAPI → AGTP-API conversion tool that reads an OpenAPI 3.x specification and produces a directory of endpoint declarations with `external_service` handlers pointing at the original HTTP endpoints. Where method mapping requires judgment, the tool flags the endpoint for human review with suggestions from the AGTP-API approved method catalog.
- * An AGTP-API → OpenAPI export tool for servers that need to publish HTTP-side documentation derived from their AGTP manifest.
- * A manifest validator that checks server manifests against this specification.
- * A conformance test runner that exercises the conformance requirements of Section 13.1 against a deployed server.
- * An `agtp-catalog-diff` tool that diffs two catalog versions and optionally scans a deployment directory to identify endpoints, recipes, and policies.methods entries that would break under a candidate catalog upgrade. See Section 4.

These tools are non-normative reference implementations. Other parties *MAY* publish equivalent tools.

A.4. Operator Notes on Catalog Versioning

The catalog version a server declares in its manifest is the version the server validates incoming method names against. Operators upgrading the on-disk verb catalog (typically by regenerating `methods.json` from a new catalog release) *SHOULD* restart the server to pick up the new version. Long-running servers that have not been restarted after a catalog upgrade may continue validating against the previous version they loaded at startup.

The reference `agtp-catalog-diff` tool is the recommended way to preview the impact of a catalog upgrade against a deployment directory before restarting. Run it as a deployment gate to surface endpoint-removal conflicts, recipe-step references to removed methods, and policies.methods entries that would be silently skipped under the new version.

Appendix B. Migration Paths (Informative)

This appendix is non-normative. It describes deployment strategies for organizations adopting AGTP-API on top of existing HTTP services, and for organizations building AGTP-API services from scratch.

B.1. Wrap-and-Expose

The lowest-effort path for an organization with an existing HTTP API. The HTTP API stays where it is. The organization writes AGTP-API endpoint declarations whose handler binding is `external_service` (Section 12.3), pointing at the existing HTTP endpoints. AGTP-API becomes a thin agent-facing facade over the existing HTTP backend. Request translation maps AGTP-API input fields to HTTP request shapes; response translation maps HTTP responses back to AGTP-API output shapes.

The OpenAPI → AGTP-API conversion tool (Appendix A) produces this kind of deployment by default: it reads an OpenAPI specification and generates a directory of endpoint declarations with `external_service` handlers. A 200-endpoint HTTP service can be made AGTP-discoverable in a single working session.

Wrap-and-expose deployments **SHOULD** populate the manifest's `apis` array (Section 8) with the HTTP API descriptor so agents that need the underlying HTTP documentation can locate it.

B.2. Translate-and-Rehost

A medium-effort path. The organization reimplements its HTTP API as native AGTP-API endpoints, moving the handler logic into the AGTP-API server's implementation language directly. The HTTP layer goes away, or is retained only for legacy clients that have not migrated.

The conversion tool generates endpoint declarations and handler stubs from the existing OpenAPI specification, leaving the organization to fill in the business logic. The work is conversion, not greenfield development.

Translate-and-rehost deployments leave the apis array empty once the HTTP layer retires; during transition, the array continues to reference the HTTP face alongside the native AGTP endpoints.

B.3. Coexist-Permanently

No migration. The HTTP API stays as the primary interface; AGTP-API is added as a parallel face for agent traffic. New agent-facing features go in the AGTP-API layer; existing HTTP traffic continues unaffected. Two interfaces, one backend, indefinite coexistence.

This is the path most large enterprises will take when regulatory, contractual, or customer-relationship commitments make a forced migration impractical. AGTP-API is additive in this posture, not replacing.

Coexist-permanently deployments populate apis with HTTP API descriptors that include `http_documentation` URLs pointing at the HTTP-side OpenAPI specifications.

B.4. Path Selection

The right path depends on organizational constraints, not protocol preferences. Wrap-and-expose is the path that scales: it is the one that can move millions of existing APIs to AGTP-API discoverability at low cost. Translate-and-rehost is the right target for organizations committing to AGTP-API as the long-term primary surface. Coexist-permanently is the honest answer for organizations that want agent access without disruption to existing systems.

Author's Address

Chris Hood
Nomotic, Inc.
Email: chris@nomotic.ai
URI: <https://agtp.io>