Model Context Protocol
Internet-Draft
Intended status: Experimental
Expires: 3 September 2026

C. Jennings
Cisco Systems
I. Swett
Google
J. Rosenberg
Five9
S. Nandakumar
Cisco Systems
2 March 2026

Model Context Protocol and Agent Skills over Media over QUIC Transport
draft-jennings-ai-mcp-over-moq-00

Abstract

This document defines how to use Media over QUIC Transport (MOQT) as the underlying transport protocol for the Model Context Protocol (MCP). MCP is a protocol that enables seamless integration between language model applications and external data sources and tools. MOQT provides efficient, low-latency, publish-subscribe media delivery over QUIC and WebTransport.

This specification describes the mapping of MCP messages onto MOQT objects and defines the procedures for establishing and maintaining MCP sessions over MOQT. It covers transport of MCP's core primitives including resources, tools, prompts, and notifications through dedicated MOQT tracks with appropriate priority management and delivery guarantees.

A key focus of this document is the delivery and execution of Agent Skills - composed instructions that extend AI capabilities beyond atomic tool operations. Skills use progressive loading (metadata, instructions, resources) that aligns naturally with MOQT's object-based delivery, enabling efficient bandwidth utilization and aggressive caching strategies.

The specification also describes relay support for scalable MCP deployments, including subscription aggregation, content caching, and multi-hop architectures that enable global distribution of MCP services with optimized performance.

About This Document

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

The latest revision of this draft can be found at https://example.org/mcp-moqt/. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-mcp-over-moqt/.

Discussion of this document takes place on the Model Context Protocol Working Group mailing list (mailto:mcp@example.org), which is archived at https://example.org/mcp/.

Source for this draft and an issue tracker can be found at https://github.com/example/mcp-moqt.

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 3 September 2026.

Copyright Notice

Table of Contents

1. Introduction

The Model Context Protocol (MCP) [MCP] enables integration between LLM applications and external data sources using JSON-RPC 2.0. Media over QUIC Transport (MOQT) [MOQT] provides efficient publish-subscribe delivery over QUIC [QUIC] with CDN support.

This document specifies MCP transport over MOQT, addressing limitations of existing transports:

* *WebSocket*: No prioritization, head-of-line blocking, no caching/relay

* *HTTP*: Request-response model conflicts with event-driven architecture, SSE has limited bidirectional support

MOQT provides:

* *Native publish-subscribe*: MCP resources, tools, and capabilities map to tracks with independent lifecycle and versioning

* *Priority-aware delivery*: Critical operations receive appropriate bandwidth

* *Multiplexing*: QUIC prevents head-of-line blocking between operations

* *Relay infrastructure*: Caching and subscription aggregation at edge

* *Network resilience*: QUIC connection migration for mobile applications

1.1. Terminology

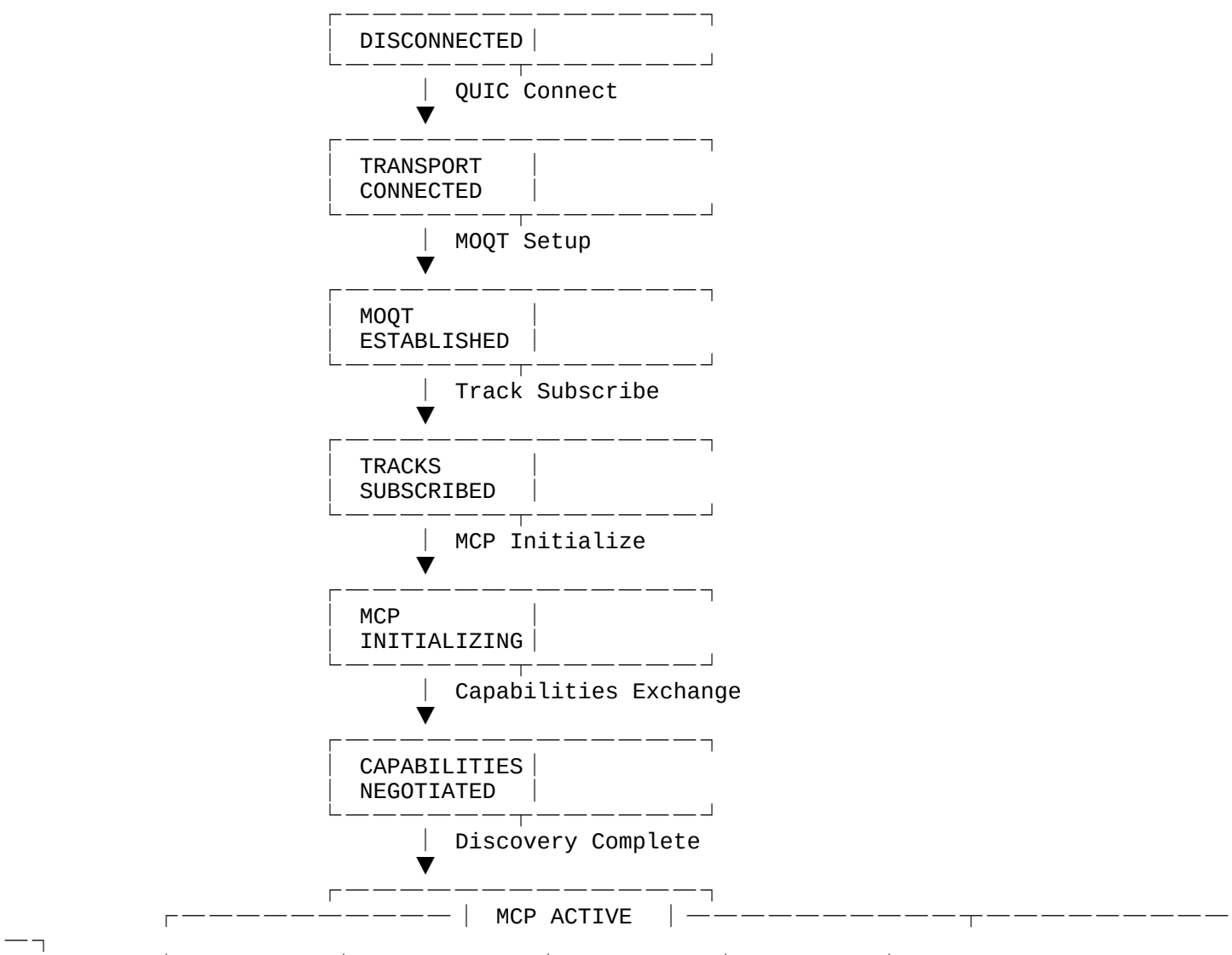
The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

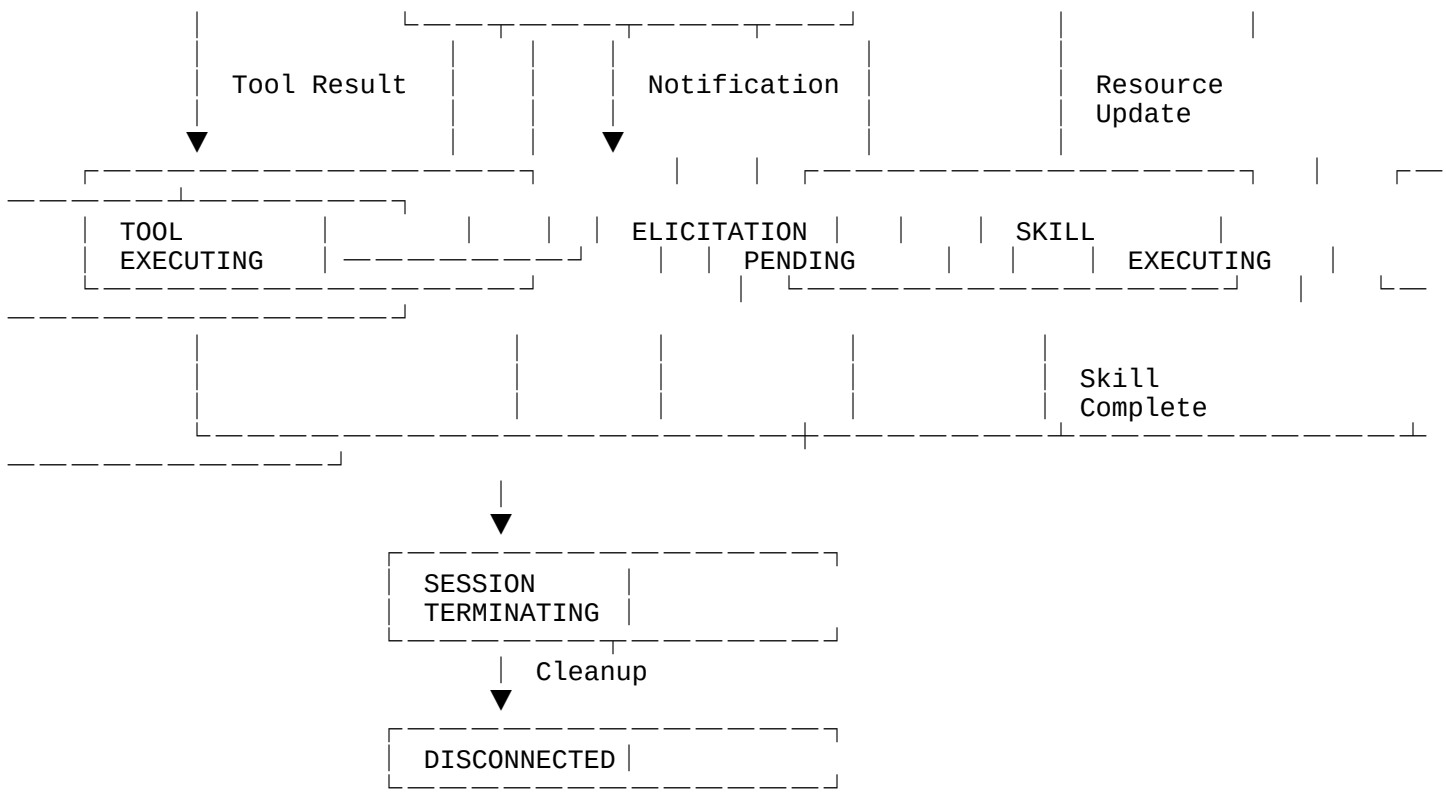
This document uses terminology from [MCP] and [MOQT].

1.2. Protocol Overview

MCP messages (JSON-RPC 2.0) are embedded in MOQT payloads. Each MCP primitive maps to dedicated tracks: resources, tools, prompts, notifications, and skills. Sessions begin with capability negotiation, then clients subscribe to relevant tracks for ongoing communication.

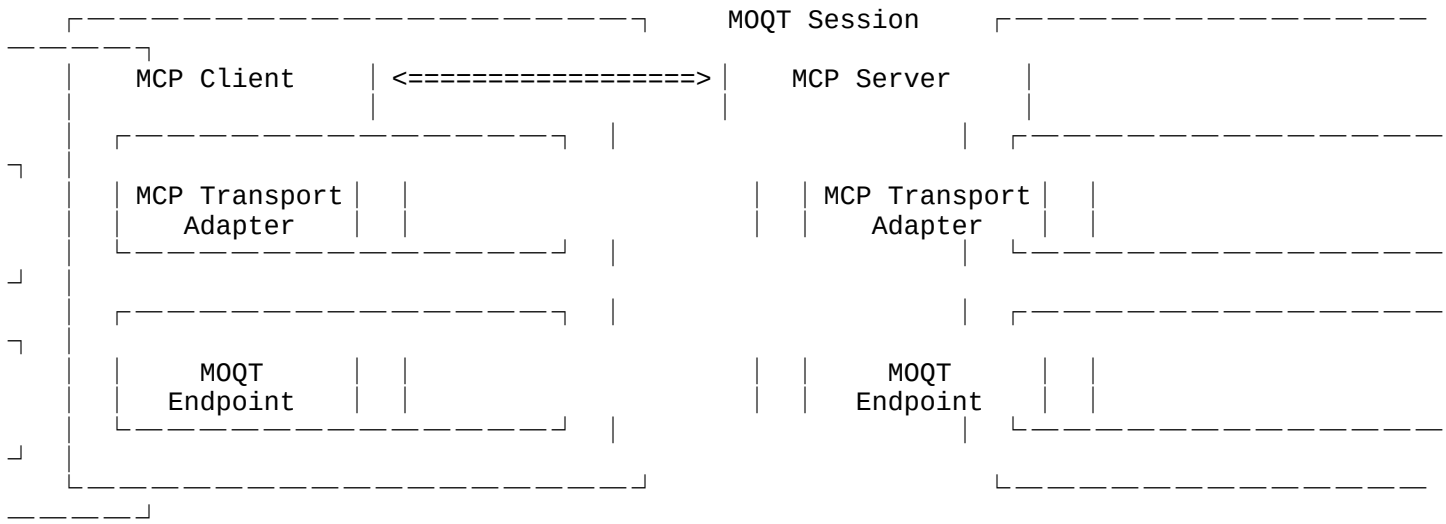
1.2.1. MCP Lifecycle State Chart





1.2.2. System Components

Below figure shows system components proposed in this specification.



The MCP client-host-server architecture maps to MOQT as follows:

* *MCP Host*: Acts as MOQT client, managing multiple server connections

* *MCP Client*: Implemented as MCP transport adapter within the host

* *MCP Server*: Acts as MOQT publisher, exposing resources, tools, and prompts

2. MOQT Tracks for MCP

2.1. Track Types

This section defines the track naming conventions and mappings for MCP operations.

2.1.1. Control Tracks

Control tracks handle MCP session management using two unidirectional tracks for bidirectional communication. The client-to-server track carries initialize requests, ping messages, capability negotiations, session teardown, and tool cancellations. The server-to-client track carries responses, pong messages, status updates, and teardown confirmations.

Track namespace:

(mcp, <session-id>, control)

Track name (Client-to-Server):

(client-to-server)

Track name (Server-to-Client):

(server-to-client)

Groups and objects are mapped as follows:

Each MCP control message is encoded as a single MOQT object within a group. Group IDs are assigned sequentially starting from 0, incrementing for each new control message. Object IDs within each group are always 0 since each control message maps to exactly one object. Objects contain JSON-encoded MCP control messages as defined in the MCP specification.

2.1.2. Resource Tracks

Resource tracks map MCP resources, delivering server-published content and metadata that serve as the main data sources exposed by MCP servers to clients.

Track namespace:

(mcp, <session-id>, resources)

Track name:

(<resource-uri>)

Groups and objects are mapped as follows:

Each resource version is assigned a unique group ID, starting from 0 and incrementing with each resource update. Within each group, objects represent chunks or segments of the resource content. Object IDs start at 0 and increment sequentially for each chunk. Objects contain the actual resource data encoded according to the resource's declared content type (binary data, JSON, text, etc.).

2.1.3. Tool Tracks

Tool tracks map MCP tool execution, facilitating tool invocation requests, responses, and progress updates using MOQT's FETCH operations.

Track namespace:

(mcp, <session-id>, tools)

Track name:

(<tool-name>)

Groups and objects are mapped as follows:

Each tool invocation creates a new group with a unique group ID assigned by the requesting client. Within each group, object ID 0 contains the tool request (JSON-encoded MCP tool call), subsequent object IDs contain tool responses and progress updates. Objects are JSON-encoded according to the MCP tool execution protocol.

2.1.4. Prompt Tracks

Prompt tracks map MCP prompts, distributing pre-defined templates and instructions that standardize common operations across the MCP ecosystem.

Track namespace:

(mcp, <session-id>, prompts)

Track name:

(<prompt-name>)

Groups and objects are mapped as follows:

Each prompt version is assigned a unique group ID starting from 0, incrementing with each prompt update. Within each group, object ID 0 contains the prompt template and metadata. Objects contain JSON-encoded prompt definitions including template text, parameter schemas, and versioning information.

2.1.5. Notification Tracks

Notification tracks map MCP notifications, providing asynchronous event delivery for server-sent notifications, progress updates, and system events.

Track namespace:

(mcp, <session-id>, notifications)

Track name:

(<category>)

The category parameter classifies notification types to enable efficient subscription management and routing. Common category values include:

* progress - Tool execution progress updates and status reports
* resources - Resource change notifications (listChanged, updated)
* prompts - Prompt template updates and availability changes
* system - Server status updates, connection events, error conditions
* elicitation - User input request notifications and responses
* tools - Tool availability changes and capability updates

Custom categories may be defined for application-specific notification types following the pattern <domain>/<type> (e.g., ai/model_updated, workspace/file_changed).

Groups and objects are mapped as follows:

Each notification event creates a new group with group IDs assigned sequentially starting from 0. Within each group, object ID 0 contains the notification payload. Objects contain JSON-encoded notification messages as defined in the MCP specification, including event type, timestamp, and payload data.

2.1.6. Elicitation Tracks

Elicitation tracks map MCP user input collection, handling interactive flows for gathering user consent and data through pairs of unidirectional tracks. The elicitation flow uses a unidirectional track pair to collect user input efficiently while preserving privacy. Each request creates isolated track pairs, allowing concurrent operations. Servers request user consent and input, while clients retain control to reject or modify requests.

Track namespace:

(mcp, <session-id>, elicitation, <request-id>)

Track name (Server-to-Client):

(<server-to-client>)

Track name (Client-to-Server):

(<client-to-server>)

The Server-to-Client elicitation track is published by the MCP Server and subscribed to by the MCP Client, carrying:

* Elicitation request initiation
* Input schema definitions
* Validation error messages
* Request timeout notifications
* Request cancellation notices

The Client-to-Server elicitation track is published by the MCP Client and subscribed to by the MCP Server, carrying:

* User consent responses
* Input data submissions

* Request rejection notifications

* Request cancellation confirmations

Groups and objects are mapped as follows:

Each elicitation exchange creates new groups with group IDs starting from 0 and incrementing for each message in the exchange. Within each group, object ID 0 contains the elicitation message (request, response, or status update). Objects contain JSON-encoded elicitation messages including input schemas, validation requirements, and user responses.

2.1.7. Log Tracks

Log tracks map MCP logging information, providing debugging and monitoring capabilities by carrying diagnostic information for system troubleshooting and performance optimization.

Track namespace:

(mcp, <session-id>, logs)

Track name:

(<category>)

The category parameter organizes log entries by severity level and source component to enable selective monitoring and efficient log processing. Standard category values include:

* error - Error conditions, exceptions, and critical failures

* warn - Warning messages and non-critical issues requiring attention

* info - General informational messages about system operations

* debug - Detailed debugging information for troubleshooting

* trace - Fine-grained execution tracing for performance analysis

* audit - Security and compliance audit trails

* metrics - Performance metrics and system statistics

Component-specific categories may be defined using the pattern <level>/<component> (e.g., error/transport, debug/tools, info/resources) to provide granular filtering capabilities.

Groups and objects are mapped as follows:

Each log entry creates a new group with group IDs assigned sequentially starting from 0, ordered by log timestamp. Within each group, object ID 0 contains the log entry. Objects contain JSON-encoded log messages including severity level, timestamp, source component, and message content.

3. Protocol Operation

3.1. MOQT Session Establishment

The session establishment process begins with establishing the underlying MOQT session, which can be either a direct QUIC connection or a WebTransport [WebTransport] session depending on the deployment environment. QUIC connections provide optimal performance for server-to- server communication, while WebTransport enables browser-based clients to participate in MCP sessions.

The MOQT handshake phase involves the exchange of CLIENT_SETUP (0x20) and SERVER_SETUP (0x21) messages that negotiate protocol versions, supported features, and operational parameters.

3.2. Proposed Session ID Discovery

MCP sessions require unique identifiers to organize track namespaces and enable proper message routing. This section defines a well-known track approach for session discovery that allows clients to dynamically discover available MCP services and obtain session identifiers through standardized FETCH operations.

3.2.1. Well-Known Discovery Track

The discovery mechanism uses a well-known track namespace and name that all MCP servers must support. This track serves as the entry point for clients to discover available MCP sessions and obtain the necessary session identifiers for subsequent operations.

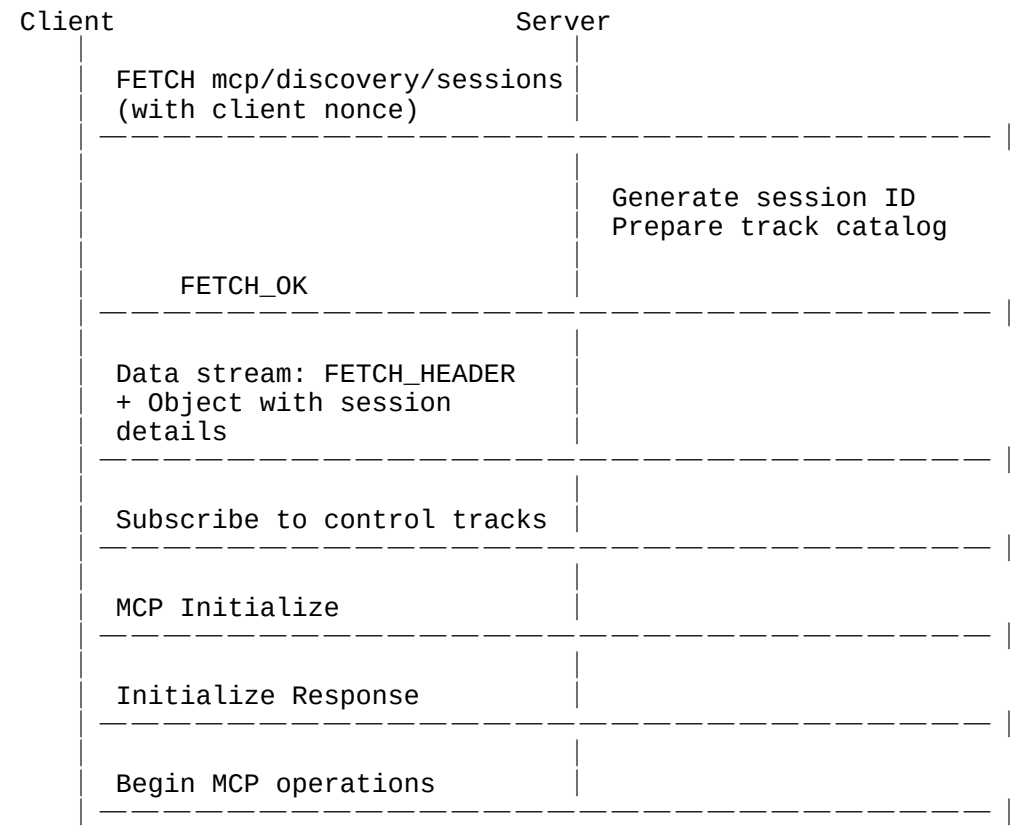
Track Namespace: mcp/discovery
Track Name: sessions

Clients initiate discovery by sending a FETCH request to this well-known track, providing a client-side nonce or identifier as a parameter. The server responds with a JSON-encoded MOQT object containing server details, a newly minted session identifier, and information about available tracks on that server.

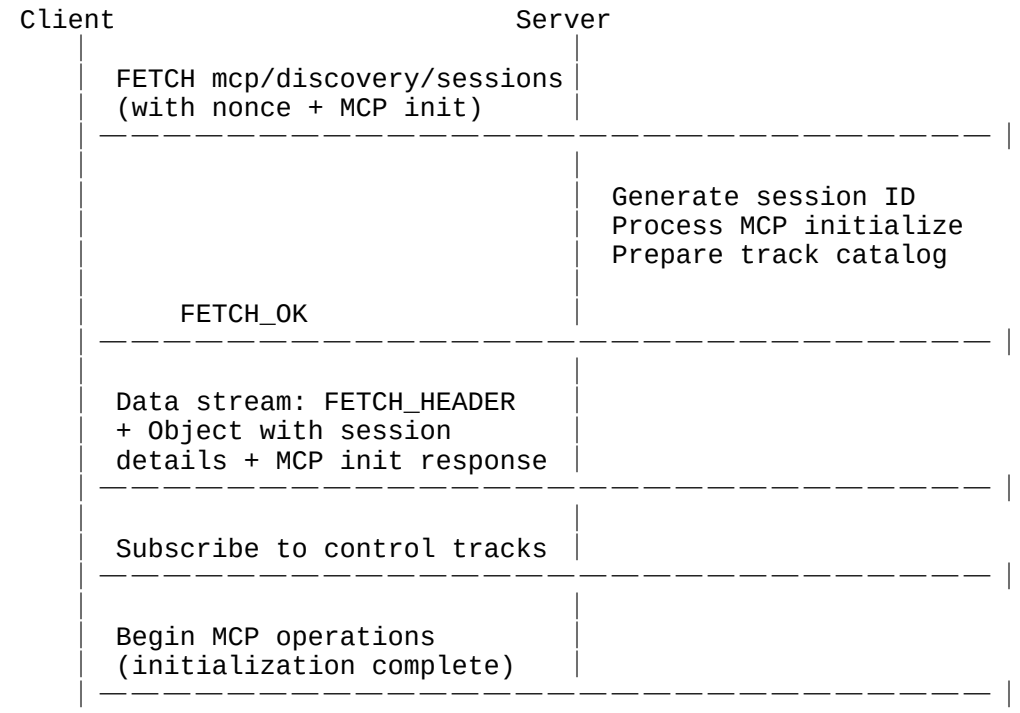
3.2.2. Discovery Flow

The session discovery process follows this sequence:

3.2.2.1. Standard Discovery Flow



3.2.2.2. Optimized Discovery Flow (RTT Reduction)



The optimized flow reduces session establishment from 4 round-trips to 2 by combining discovery and MCP initialization into a single exchange.

3.2.3. Discovery Request Format

Clients send FETCH requests to the well-known discovery track with a client-generated nonce or identifier in the payload:

```
FETCH {
  Type (i) = 0x16,
  Length (16),
  Request ID (i) = 1,
  Subscriber Priority (8) = 30,
  Group Order (8) = 0,
  Fetch Type (i) = 0x1,  // Standalone
  Track Namespace (tuple) = (... "mcp", "discovery"),
  Track Name Length (i) = 8,
  Track Name = "sessions",
  Start Location = {Group: 0, Object: 0},
  End Location = {Group: 0, Object: 1},
  Number of Parameters (i) = 1,
  Parameters = {
    "mcp_payload": {
      "jsonrpc": "2.0",
      "id": 1,
      "method": "discovery/request_session",
      "params": {
        "client_nonce": "client-nonce-abc123",
        "client_info": {
          "name": "ExampleClient",
          "version": "1.0.0"
        },
        "requested_capabilities": ["resources", "tools", "prompts", "skills"]
      }
    }
  }
}
```

3.2.3.1. RTT Optimization with Combined Initialize

To reduce round-trip time, clients MAY include their MCP initialize request within the discovery FETCH payload. This allows the server to perform both session discovery and MCP initialization in a single exchange:

```
FETCH {
  Type (i) = 0x16,
  Length (16),
  Request ID (i) = 1,
  Subscriber Priority (8) = 30,
  Group Order (8) = 0,
  Fetch Type (i) = 0x1,  // Standalone
  Track Namespace (tuple) = ("mcp", "discovery"),
  Track Name Length (i) = 8,
  Track Name = "sessions",
  Start Location = {Group: 0, Object: 0},
  End Location = {Group: 0, Object: 1},
  Number of Parameters (i) = 1,
  Parameters = {
    "mcp_payload": {
      "jsonrpc": "2.0",
      "id": 1,
      "method": "discovery/request_session_with_init",
      "params": {
        "client_nonce": "client-nonce-abc123",
        "client_info": {
          "name": "ExampleClient",
          "version": "1.0.0"
        },
        "requested_capabilities": ["resources", "tools", "prompts", "skills"],
        "mcp_initialize": {
          "protocolVersion": "2025-06-18",
          "capabilities": {
            "resources": { "subscribe": true },
            "tools": { "progress": true },
            "prompts": {},
            "elicitation": {},
            "skills": { "progressive_loading": true, "streaming": true }
          },
          "clientInfo": {
            "name": "ExampleClient",
            "version": "1.0.0"
          }
        }
      }
    }
  }
}
```

3.2.4. Discovery Response Format

The server responds with a JSON-encoded MOQT object containing the newly minted session ID and control track information:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "session_id": "session-uuid-456",
    "server_info": {
      "name": "ExampleServer",
      "version": "2.0.0",
      "protocol_version": "2025-06-18"
    },
    "control_tracks": {
      "client_to_server":
         "mcp/session-uuid-456/control/client-to-server",
      "server_to_client":
         "mcp/session-uuid-456/control/server-to-client"
    },
    "session_namespace": "mcp/session-uuid-456",
    "session_expires": "2025-06-18T12:00:00Z"
  }
}
```

3.2.4.1. Combined Discovery and Initialize Response

When the client uses the RTT optimization by including MCP initialize in the discovery request, the server responds with both session discovery and MCP initialization results:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "session_id": "session-uuid-456",
    "server_info": {
      "name": "ExampleServer",
      "version": "2.0.0",
      "protocol_version": "2025-06-18"
    },
    "control_tracks": {
      "client_to_server":
         "mcp/session-uuid-456/control/client-to-server",
      "server_to_client":
         "mcp/session-uuid-456/control/server-to-client"
    },
    "session_namespace": "mcp/session-uuid-456",
    "session_expires": "2025-06-18T12:00:00Z",
    "mcp_initialize_response": {
      "protocolVersion": "2025-06-18",
      "capabilities": {
        "resources": { "subscribe": true, "listChanged": true },
        "tools": { "progress": true },
        "prompts": { "listChanged": true },
        "elicitation": {},
        "skills": { "progressive_loading": true, "streaming": true }
      },
      "serverInfo": {
        "name": "ExampleServer",
        "version": "2.0.0"
      }
    }
  }
}
```

After establishing a session, clients discover available capabilities (resources, tools, prompts, and skills) using the mechanisms described in Section 4.2.

After receiving the discovery response, clients subscribe to control tracks and proceed with capability discovery (see Section 4.2).

3.3. Priority Management

MOQT’s object priorities optimize MCP message delivery by considering the importance of different track types specified.

The following priority assignments are RECOMMENDED, but applications MAY adjust them as needed:

Priority	Range	Track Type	Example
CRITICAL	1-5	Session Control	initialize, terminate
HIGHEST	6-15	User Elicitation	permission prompts
HIGH	16-30	Skills/Tool Execution	/commit, git operations
MEDIUM-HI	31-45	Notifications	resource changes
MEDIUM	46-60	Skill Instructions	prompt loading
MEDIUM-LO	61-75	Resources	documentation
LOW	76-90	Tool Schemas	capability metadata
LOWEST	91-127	Logs	debug, trace

Table 1

Priorities can also be dynamically adjusted based on:

* User interaction urgency (elicitation requests get higher priority)

* Skill/tool execution criticality (error handling gets priority boost)

* Resource size and frequency (large resources get lower priority)

* Server load conditions (adaptive priority scaling)

* Active user engagement (boost skill loading when user is waiting)

3.4. Error Handling

Error handling in MCP over MOQT leverages both MOQT's native error mechanisms and MCP's JSON-RPC error responses.

3.4.1. Transport-Level Errors

MOQT provides several error handling mechanisms that apply to MCP operations:

* *SUBSCRIBE_ERROR*: Returned when a client cannot subscribe to an MCP track. Common causes include invalid session IDs, unauthorized access, or non-existent resources.

* *FETCH_ERROR*: Returned when a FETCH operation fails. For tool execution, this indicates the tool request could not be processed at the transport level.

* *GOAWAY*: Indicates the server is terminating the session. Clients SHOULD attempt to reconnect and re-establish the MCP session.

3.4.2. MCP-Level Errors

MCP errors are returned as JSON-RPC error responses within MOQT objects:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "error": {
    "code": -32600,
    "message": "Invalid Request",
    "data": {
      "details": "Tool 'unknown_tool' not found"
    }
  }
}
```

Standard MCP error codes apply: - -32700: Parse error - -32600: Invalid request - -32601: Method not found - -32602: Invalid params - -32603: Internal error

3.4.3. Skills-Specific Errors

Skills execution may encounter additional error conditions:

Error Code	Description	Recovery Action
-33001	Skill not found	Check registry, refresh cache
-33002	Required tool unavailable	Verify tool server connectivity
-33003	Dependency resolution failed	Check skill dependencies
-33004	Instruction load timeout	Retry with higher priority
-33005	Context sharing conflict	Resolve context state

Table 2

3.4.4. Error Recovery

Clients SHOULD implement the following recovery strategies:

1. *Transient errors*: Retry with exponential backoff
2. *Session errors*: Re-establish MCP session via discovery
3. *Resource errors*: Invalidate cache and re-fetch
4. *Skill errors*: Fall back to individual tool operations if possible

4. Relay Support

MOQT relays enable scalable MCP deployments by providing intelligent aggregation, caching, and distribution capabilities that optimize performance for AI workflows and reduce server load across multiple clients.

4.1. Subscription Aggregation and Caching

MOQT relays aggregate multiple client subscriptions to the same content into single upstream requests, reducing server load and bandwidth. When clients subscribe to identical resources, tool metadata, or notification streams, the relay maintains one upstream subscription and fans out content to all clients.

Relays also cache content aggressively:

* *Static resources* (documentation, API references): Cached with long TTLs since they change infrequently but are accessed frequently

* *Tool schemas*: Cached until explicitly invalidated, as they define stable interface contracts

* *Capability metadata*: Cached as one-time objects after session initialization

* *Skills registry and instructions*: Cached by version with push-based invalidation

4.2. Capability Discovery

After establishing a session (Section 3.2), clients discover available capabilities (resources, tools, prompts, and skills) using two complementary mechanisms:

1. *Capability Catalog*: Application-level catalog with rich metadata

2. *Namespace-Based Discovery*: Protocol-level discovery using native MOQT messages

Implementations MAY support either or both mechanisms depending on their needs.

4.2.1. Capability Catalog

Servers implement capability catalog tracks that enumerate available MCP capabilities and their associated track patterns within a specific session:

Track namespace: mcp/<session-id>/discovery Track name: catalog

```
{
  "server_catalog": {
    "mcp/{session-id}/resources": {
      "available_tracks": [
        "documentation", "api_schemas", "examples", "templates"
      ],
      "content_types": ["text/markdown",
             "application/json", "text/plain"],
      "update_frequency": "daily",
      "cache_recommended": true
    },
    "mcp/{session-id}/tools/": {
      "available_tracks": [
        "file_operations", "database_query", "code_analysis"
      ],
      "execution_types": ["synchronous", "asynchronous"],
      "progress_tracking": true,
      "cache_recommended": false
    },
    "mcp/{session-id}/prompts/": {
      "available_tracks": [
        "code_review", "data_analysis", "content_generation"
      ],
      "template_versions": ["1.0", "1.1", "2.0"],
      "cache_recommended": true
    }
  },
  "discovery_timestamp": "2025-06-18T10:30:00Z"
}
```

Clients discover available capabilities by issuing FETCH requests to the catalog track for initial discovery, or SUBSCRIBE for real-time updates when capabilities change.

4.2.2. Namespace-Based Discovery

MOQT draft-16 introduces SUBSCRIBE_NAMESPACE and PUBLISH_NAMESPACE messages that enable protocol-level discovery of available tracks under specific namespace prefixes.

4.2.2.1. SUBSCRIBE_NAMESPACE for Track Discovery

Clients use SUBSCRIBE_NAMESPACE to discover all available tracks under an MCP namespace prefix without prior knowledge of specific track names:

```
   SUBSCRIBE_NAMESPACE {
     Request ID (i) = 1,
     Track Namespace = ("mcp", "<session-id>", "resources")
   }
```

The server responds with NAMESPACE messages for each available track:

```
NAMESPACE {
  Request ID (i) = 1,
  Track Namespace = ("mcp", "<session-id>", "resources", "documentation")
}

NAMESPACE {
  Request ID (i) = 1,
  Track Namespace = ("mcp", "<session-id>", "resources", "api-schemas")
}

NAMESPACE_DONE {
  Request ID (i) = 1,
  Status Code (i) = 0,
  Reason Phrase = ""
}
```

4.2.2.2. PUBLISH_NAMESPACE for Server Announcements

MCP servers use PUBLISH_NAMESPACE to proactively announce track namespace availability to connected clients and relays:

```
   PUBLISH_NAMESPACE {
     Request ID (i) = 1,
     Track Namespace = ("mcp", "<session-id>", "tools")
   }
```

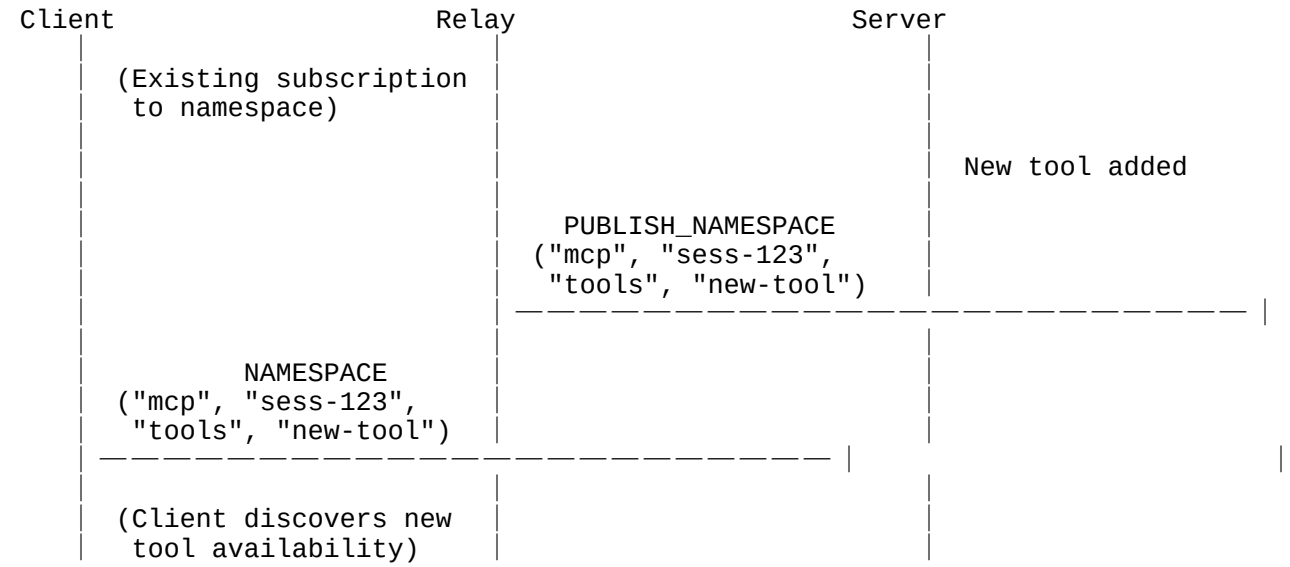
Relays receiving PUBLISH_NAMESPACE forward these announcements to interested subscribers, enabling efficient fan-out of capability updates.

4.2.3. Dynamic Capability Updates

Both discovery mechanisms support real-time updates when capabilities change during a session.

Capability Catalog Updates: Clients subscribed to the capability catalog track receive new catalog objects when capabilities are added, removed, or modified.

Namespace Updates: Servers publish PUBLISH_NAMESPACE announcements that relays forward to clients with active namespace subscriptions:



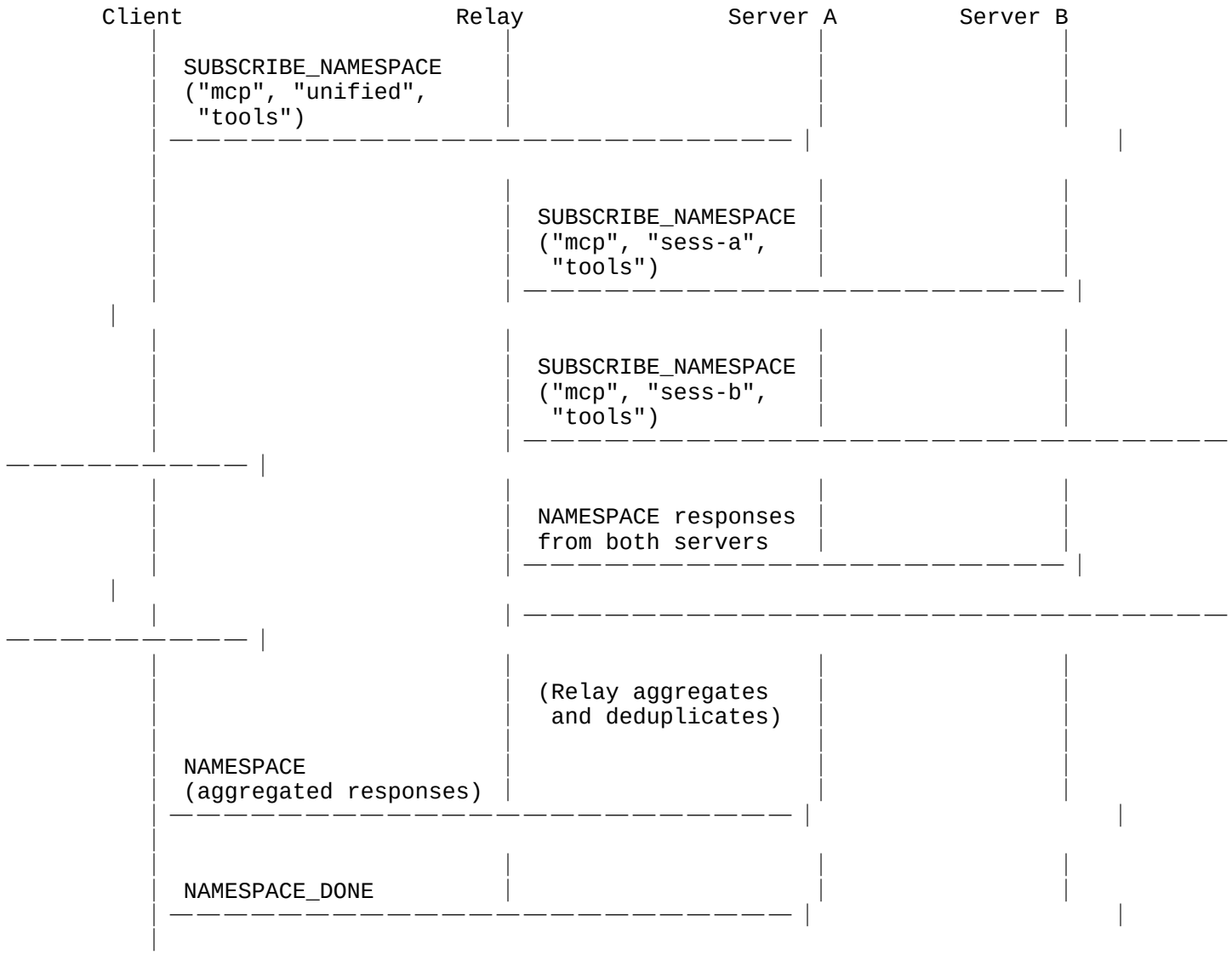
4.2.4. Choosing a Discovery Mechanism

Use Case	Recommended Approach
Rich metadata needed (content types, caching hints)	Capability Catalog
Simple track enumeration	Namespace-Based
Relay aggregation scenarios	Namespace-Based
Legacy MOQT implementations (pre-draft-16)	Capability Catalog
Minimal protocol overhead	Namespace-Based

Table 3

4.2.5. Relay Namespace Aggregation

Relays can aggregate namespace information from multiple upstream servers and present a unified namespace view to downstream clients:



5. Agent Skills and MCP Integration

Agent Skills represent a paradigm shift in how AI capabilities are extended and distributed. Unlike atomic tools that perform single operations, Skills provide composed instructions for complex tasks that may orchestrate multiple tools. This section describes the comprehensive architecture for Skills delivery and execution over MOQT.

5.1. Skills Overview

Skills and Tools serve complementary roles in MCP. Tools provide atomic capabilities invoked via direct RPC calls, while Skills provide composed instructions for complex tasks through prompt expansion and orchestration. Tools are stateless with schemas always available; Skills are context-aware with progressive loading (metadata, then instructions, then resources).

Support for Skills is OPTIONAL. Servers advertising the skills capability MUST implement the skills registry track and support FETCH operations for skill metadata and instructions. Clients SHOULD cache skill content by version. When Skills are not supported, clients fall back to direct tool operations.

Skills use three progressive loading levels optimized for MOQT delivery:

* *Metadata* (~100 tokens): Skill identification, triggers, dependencies

* *Instructions* (<5k tokens): Full prompt text and execution logic

* *Resources*: Templates, schemas, and examples loaded during execution

5.2. Skills Track Structure and Data Formats

Skills use a hierarchical track namespace under mcp/<session-id>/skills/:

Track	Namespace	Name	Purpose
Registry	(mcp, <session-id>, skills)	registry	Complete skill catalog for trigger matching
Metadata	(mcp, <session-id>, skills, <skill-id>)	metadata	Skill identification, triggers, dependencies
Instructions	(mcp, <session-id>, skills, <skill-id>)	instructions	Full prompt text and execution logic
Execution	(mcp, <session-id>, skills, <skill-id>, execution)	<invocation-id>	Bidirectional streaming for active invocations

Table 4

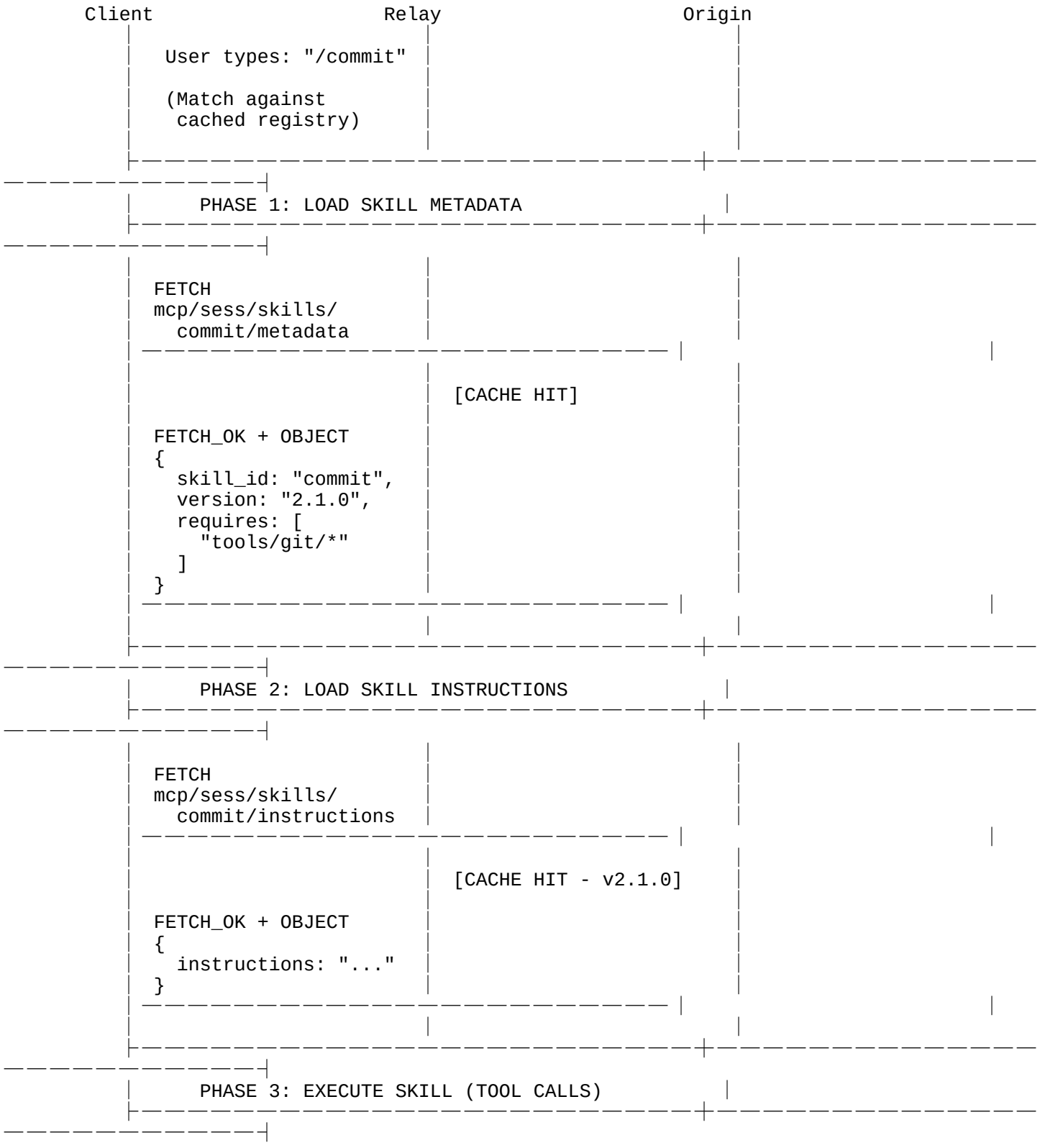
The registry track contains an array of skill entries, each specifying: id, name, version, description, triggers (commands and patterns), loading hints (token sizes, preload preferences), required/optional tools and permissions, and composition relationships (which skills can invoke each other).

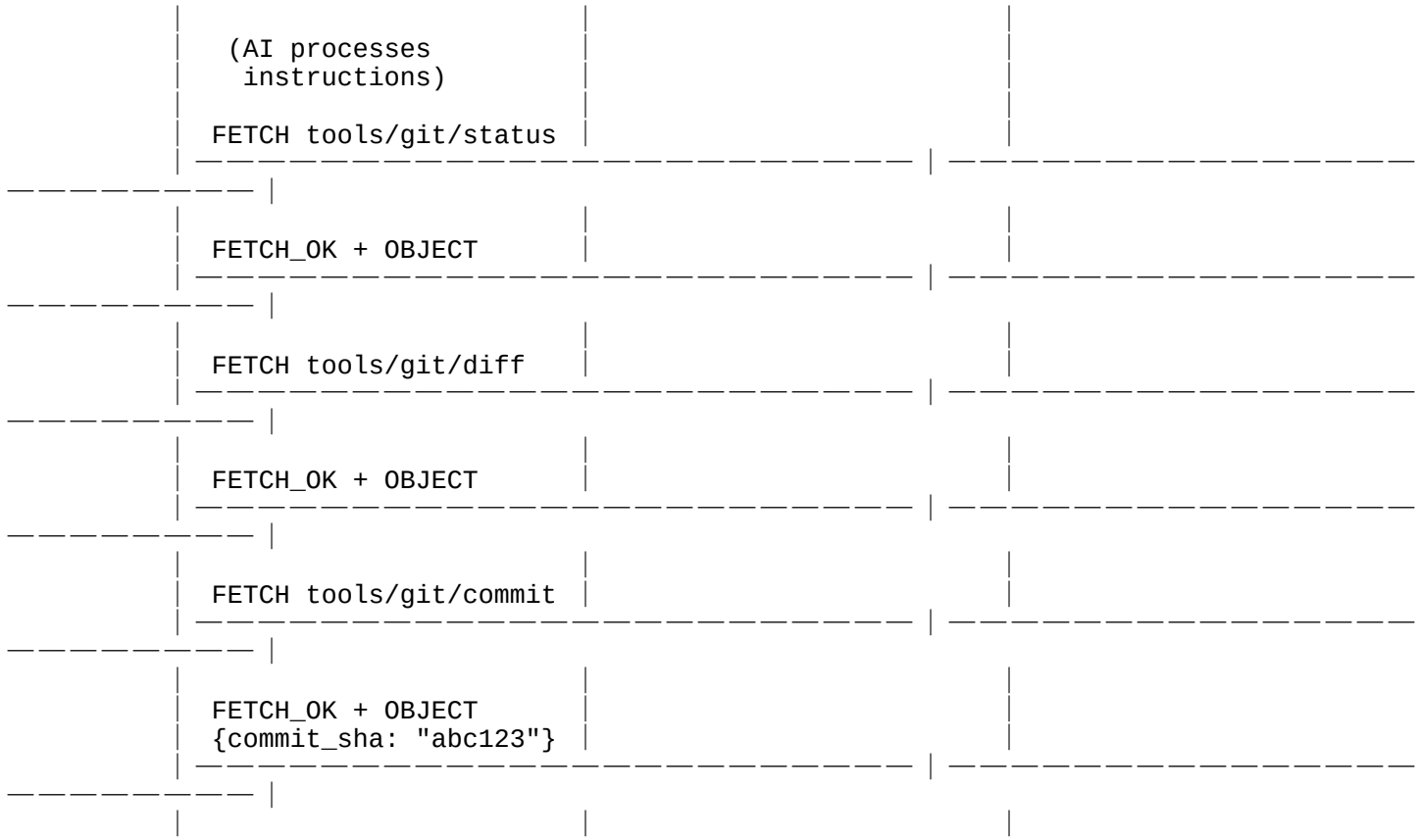
Individual metadata objects provide detailed capability information including content hashes for integrity verification, activation context requirements, tool and resource dependencies, permission requirements, and execution characteristics (streaming support, cancellation, timeouts).

Instructions objects contain the complete prompt text with step-by-step execution logic, parameter schemas with types and defaults, and output format specifications for success and error cases.

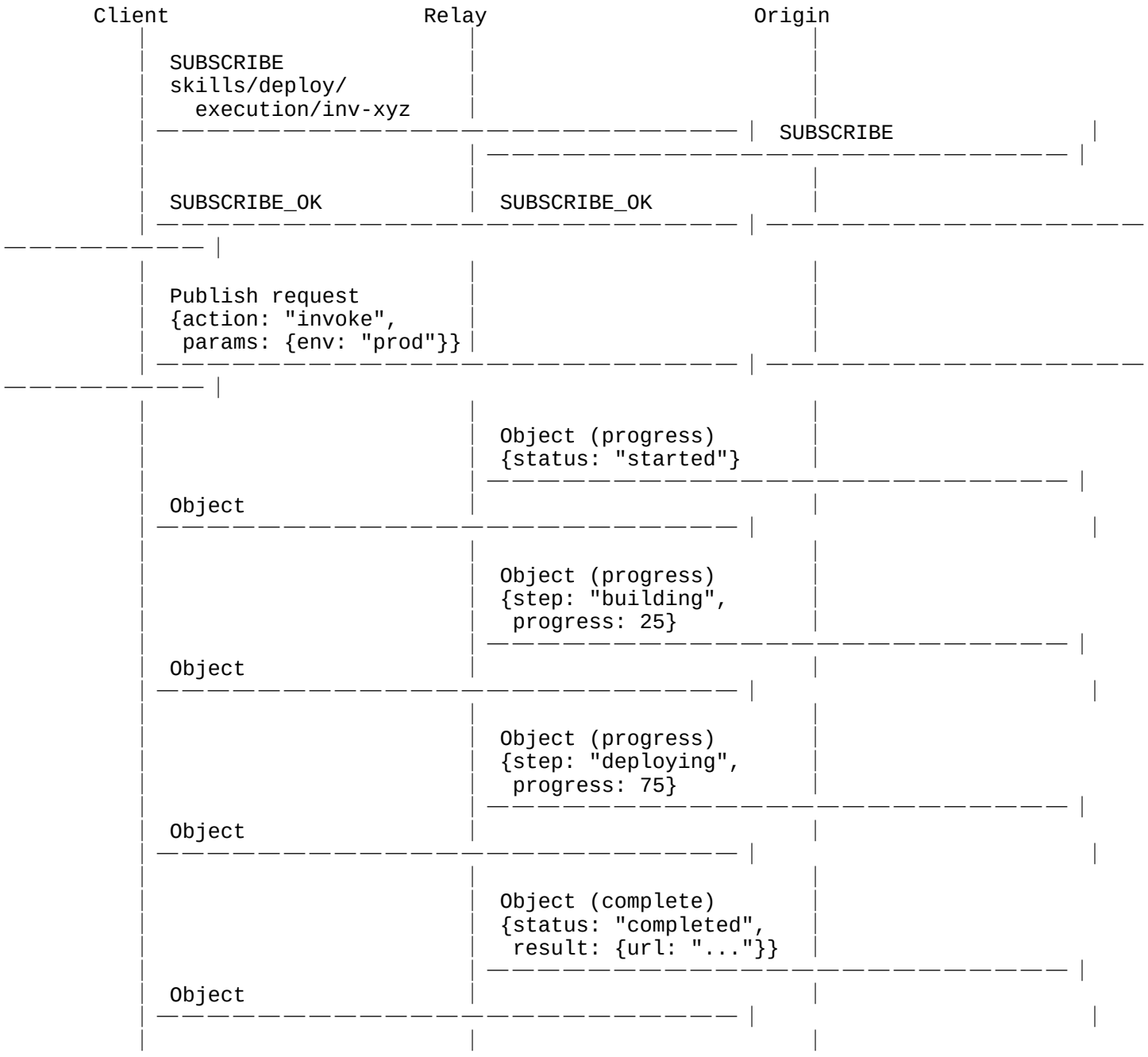
5.3. Skills Invocation Protocol

5.3.1. Invocation Flow


Client
Relay
Origin
User types: "/commit"
(Match against cached registry)
PHASE 1: LOAD SKILL METADATA
FETCH mcp/sess/skills/ commit/metadata
[CACHE HIT]
FETCH_OK + OBJECT
{
skill_id: "commit",
version: "2.1.0",
requires: [
"tools/git/*"
]
}
PHASE 2: LOAD SKILL INSTRUCTIONS
FETCH mcp/sess/skills/ commit/instructions
[CACHE HIT - v2.1.0]
FETCH_OK + OBJECT
{
instructions: "..."
}
PHASE 3: EXECUTE SKILL (TOOL CALLS)



5.3.2. Streaming Skill Execution with Progress



5.3.3. Skills Execution Request Format

```
{
  "jsonrpc": "2.0",
  "id": "exec-001",
  "method": "skills/invoke",
  "params": {
    "skill_id": "commit",
    "invocation_id": "inv-xyz789",
    "parameters": {
      "push": false
    },
    "context": {
      "working_directory": "/home/user/project",
      "git_state": {
        "branch": "feature/new-feature",
        "has_staged": true
      }
    },
    "options": {
      "stream_progress": true,
      "timeout_ms": 60000
    }
  }
}
```

5.3.4. Skills Execution Response Stream

```
// Object 0: Acknowledgment
{
  "jsonrpc": "2.0",
  "id": "exec-001",
  "result": {
    "status": "accepted",
    "invocation_id": "inv-xyz789"
  }
}

// Object 1: Progress
{
  "jsonrpc": "2.0",
  "method": "skills/progress",
  "params": {
    "invocation_id": "inv-xyz789",
    "step": 1,
    "total_steps": 4,
    "message": "Checking repository status...",
    "tool_call": {"tool": "git/status", "status": "in_progress"}
  }
}
```

```
// Object 2: Progress
{
  "jsonrpc": "2.0",
  "method": "skills/progress",
  "params": {
    "invocation_id": "inv-xyz789",
    "step": 2,
    "message": "Analyzing changes...",
    "tool_call": {"tool": "git/diff", "status": "completed"}
  }
}

// Object 3: Completion
{
  "jsonrpc": "2.0",
  "id": "exec-001",
  "result": {
    "status": "completed",
    "output": {
      "commit_sha": "abc123def456",
      "message": "Add user authentication flow",
      "files_committed": ["src/auth.ts", "src/middleware/jwt.ts"]
    }
  }
}
```

6. Multi-Server Context Sharing

When an MCP client connects to multiple servers simultaneously, each connection maintains its own session ID and track namespace for isolation:

```
Server A: mcp/session-a-uuid/resources/documentation
Server B: mcp/session-b-uuid/tools/code_analysis
Server C: mcp/session-c-uuid/prompts/review_templates
```

The client coordinates these sessions while presenting a unified capability view. Key considerations include:

* *Resource synchronization*: Subscribe to related tracks across servers, selecting sources based on performance, recency, or availability

* *Tool execution context*: Track execution state across servers for proper workflow sequencing; pass results between servers as needed

* *Cache coordination*: Reuse cached context across servers where appropriate, tracking versions for consistency

7. Security Considerations

TODO

8. IANA Considerations

TODO

9. Examples

9.1. Basic Session Establishment

This example shows a minimal MCP session establishment over MOQT:

1. Client establishes QUIC connection to server
2. Client sends CLIENT_SETUP, server responds with SERVER_SETUP
3. Client sends FETCH to mcp/discovery/sessions with client info
4. Server responds with session_id and available tracks
5. Client subscribes to control tracks
6. Client sends MCP initialize on client-to-server control track
7. Server responds with capabilities on server-to-client control track
8. Session is now active

9.2. Tool Execution Example

Executing a file read tool:

```
Client → Server:
FETCH mcp/{session-id}/tools/filesystem
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "tools/call",
  "params": {
    "name": "read_file",
    "arguments": {"path": "/project/README.md"}
  }
}

Server → Client:
FETCH_OK + OBJECT
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "content": [
      {"type": "text", "text": "# Project README\n..."}
    ]
  }
}
```

9.3. Skill Invocation Example

Invoking the /commit skill with progressive loading:

1. User types: "/commit"

2. Client checks cached registry, finds "commit" skill

3. Client FETCHes skill metadata (if not cached):
   FETCH mcp/{session-id}/skills/commit/metadata
   → Returns: version, dependencies, required tools

4. Client FETCHes skill instructions:
   FETCH mcp/{session-id}/skills/commit/instructions
   → Returns: full prompt with execution steps

5. AI processes instructions, calls required tools:
   FETCH mcp/{session-id}/tools/git {method: "status"}
   FETCH mcp/{session-id}/tools/git {method: "diff"}
   FETCH mcp/{session-id}/tools/git {method: "commit", args: {...}}

6. Skill completes, returns result to user

9.4. Multi-Server Skill Workflow

A deployment workflow using skills from multiple servers:

User: "/deploy to production"

1. Load /deploy skill from Enterprise Server B
2. /deploy instructions: "First ensure changes committed"
3. Invoke /commit skill from Anthropic Server A
4. /commit executes using GitHub Tools Server
5. Return to /deploy, execute deployment tools
6. Stream progress updates to client
7. Return deployment result with URL

10. References

10.1. Normative References

[MCP] Anthropic, "Model Context Protocol Specification", 18 June 2025, <https://modelcontextprotocol.io/specification/2025-06-18>.

[MOQT] Nandakumar, S., Vasiliev, V., Swett, I., and A. Frindell, "Media over QUIC Transport", 16 December 2025, <https://www.ietf.org/archive/id/draft-ietf-moq-transport-16>.

[QUIC] Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", May 2021, <https://www.rfc-editor.org/rfc/rfc9000.html>.

[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>.

[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>.

10.2. Informative References

[WebTransport] Vasiliev, V., "The WebTransport Protocol Framework", June 2023, <https://www.rfc-editor.org/rfc/rfc9297.html>.

Appendix A. Acknowledgments

TODO

Authors' Addresses

Cullen Jennings
Cisco Systems
Email: fluffy@cisco.com

Ian Swett
Google
Email: ianswett@google.com

Jonathan Rosenberg
Five9
Email: jdrosen@jdrosen.net

Suhas Nandakumar
Cisco Systems
Email: snandaku@cisco.com