

Specification · **Superseded**

Open Model Context Protocol

Version 2025-06-18

Built from [7596f66](#) · 2026-09-23

SUPERSEDED

This version has been superseded by 2026-07-28.

Contents

- 1 Specification
 - Overview
 - Key Details
 - Security and Trust & Safety
 - Learn More
- 2 Key Changes
 - Major changes
 - Other schema changes
 - Full changelog
- 3 Architecture
 - Core Components
 - Design Principles
 - Capability Negotiation
- 4 Base Protocol
 - 4.1 Overview
 - Messages
 - Auth
 - Schema
 - 4.2 Lifecycle
 - Lifecycle Phases
 - Timeouts
 - Error Handling
 - 4.3 Transports
 - stdio
 - Streamable HTTP
 - Custom Transports
 - 4.4 Authorization
 - Introduction
 - Authorization Flow
 - Security Considerations
 - 4.5 Utilities
 - 4.5.1 Cancellation
 - Cancellation Flow
 - Behavior Requirements
 - Timing Considerations
 - Implementation Notes
 - Error Handling
 - 4.5.2 Ping
 - Overview
 - Message Format
 - Behavior Requirements
 - Usage Patterns
 - Implementation Considerations
 - Error Handling

4.5.3 Progress

Progress Flow

Behavior Requirements

Implementation Notes

5 Client Features

5.1 Roots

User Interaction Model

Capabilities

Protocol Messages

Message Flow

Data Types

Error Handling

Security Considerations

Implementation Guidelines

5.2 Sampling

User Interaction Model

Capabilities

Protocol Messages

Message Flow

Data Types

Error Handling

Security Considerations

5.3 Elicitation

User Interaction Model

Capabilities

Protocol Messages

Message Flow

Request Schema

Response Actions

Security Considerations

6 Server Features

6.1 Overview

6.2 Prompts

User Interaction Model

Capabilities

Protocol Messages

Message Flow

Data Types

Error Handling

Implementation Considerations

Security

6.3 Resources

User Interaction Model

Capabilities

Protocol Messages

Message Flow

- Data Types
- Common URI Schemes
- Error Handling
- Security Considerations

6.4 Tools

- User Interaction Model
- Capabilities
- Protocol Messages
- Message Flow
- Data Types
- Error Handling
- Security Considerations

6.5 Utilities

6.5.1 Completion

- User Interaction Model
- Capabilities
- Protocol Messages
- Message Flow
- Data Types
- Error Handling
- Implementation Considerations
- Security

6.5.2 Logging

- User Interaction Model
- Capabilities
- Log Levels
- Protocol Messages
- Message Flow
- Error Handling
- Implementation Considerations
- Security

6.5.3 Pagination

- Pagination Model
- Response Format
- Request Format
- Pagination Flow
- Operations Supporting Pagination
- Implementation Guidelines
- Error Handling

7 Schema Reference

- JSON-RPC
- Common Types
- Content
- completion/complete
- elicitation/create
- initialize

logging/setLevel
notifications/cancelled
notifications/initialized
notifications/message
notifications/progress
notifications/prompts/list_changed
notifications/resources/list_changed
notifications/resources/updated
notifications/roots/list_changed
notifications/tools/list_changed
ping
prompts/get
prompts/list
resources/list
resources/read
resources/subscribe
resources/templates/list
resources/unsubscribe
roots/list
sampling/createMessage
tools/call
tools/list

1 Specification

Model Context Protocol (MCP) is an open protocol that enables seamless integration between LLM applications and external data sources and tools. Whether you're building an AI-powered IDE, enhancing a chat interface, or creating custom AI workflows, MCP provides a standardized way to connect LLMs with the context they need.

This specification defines the authoritative protocol requirements, based on the TypeScript schema in [schema.ts](#).

For implementation guides and examples, visit openmodelcontextprotocol.org.

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.

Overview

MCP provides a standardized way for applications to:

- Share contextual information with language models
- Expose tools and capabilities to AI systems
- Build composable integrations and workflows

The protocol uses [JSON-RPC 2.0](#) messages to establish communication between:

- **Hosts:** LLM applications that initiate connections
- **Clients:** Connectors within the host application
- **Servers:** Services that provide context and capabilities

MCP takes some inspiration from the [Language Server Protocol](#), which standardizes how to add support for programming languages across a whole ecosystem of development tools. In a similar way, MCP standardizes how to integrate additional context and tools into the ecosystem of AI applications.

Key Details

Base Protocol

- [JSON-RPC](#) message format
- Stateful connections
- Server and client capability negotiation

Features

Servers offer any of the following features to clients:

- **Resources:** Context and data, for the user or the AI model to use
- **Prompts:** Templated messages and workflows for users
- **Tools:** Functions for the AI model to execute

Clients may offer the following features to servers:

- **Sampling:** Server-initiated agentic behaviors and recursive LLM interactions
- **Roots:** Server-initiated inquiries into uri or filesystem boundaries to operate in
- **Elicitation:** Server-initiated requests for additional information from users

Additional Utilities

- Configuration
- Progress tracking
- Cancellation
- Error reporting
- Logging

Security and Trust & Safety

The Model Context Protocol enables powerful capabilities through arbitrary data access and code execution paths. With this power comes important security and trust considerations that all implementors must carefully address.

Key Principles

1. User Consent and Control

- Users must explicitly consent to and understand all data access and operations
- Users must retain control over what data is shared and what actions are taken
- Implementors should provide clear UIs for reviewing and authorizing activities

2. Data Privacy

- Hosts must obtain explicit user consent before exposing user data to servers
- Hosts must not transmit resource data elsewhere without user consent
- User data should be protected with appropriate access controls

3. Tool Safety

- Tools represent arbitrary code execution and must be treated with appropriate caution.
 - In particular, descriptions of tool behavior such as annotations should be considered untrusted, unless obtained from a trusted server.
- Hosts must obtain explicit user consent before invoking any tool
- Users should understand what each tool does before authorizing its use

4. LLM Sampling Controls

- Users must explicitly approve any LLM sampling requests
- Users should control:
 - Whether sampling occurs at all
 - The actual prompt that will be sent
 - What results the server can see
- The protocol intentionally limits server visibility into prompts

Implementation Guidelines

While MCP itself cannot enforce these security principles at the protocol level, implementors **SHOULD**:

1. Build robust consent and authorization flows into their applications
2. Provide clear documentation of security implications
3. Implement appropriate access controls and data protections
4. Follow security best practices in their integrations
5. Consider privacy implications in their feature designs

Learn More

Explore the detailed specification for each protocol component:

[Architecture](#)[Base Protocol](#)[Server Features](#)[Client Features](#)[Contributing](#)

2 Key Changes

This document lists changes made to the Model Context Protocol (MCP) specification since the previous revision, 2025-03-26.

Major changes

1. Remove support for JSON-RPC **batching** (PR [#416](#))
2. Add support for structured tool output (PR [#371](#))
3. Classify MCP servers as OAuth Resource Servers, adding protected resource metadata to discover the corresponding Authorization server. (PR [#338](#))
4. Require MCP clients to implement Resource Indicators as described in [RFC 8707](#) to prevent malicious servers from obtaining access tokens. (PR [#734](#))
5. Clarify security considerations and best practices in the authorization spec and in a new security best practices page.
6. Add support for **elicitation**, enabling servers to request additional information from users during interactions. (PR [#382](#))
7. Add support for **resource links** in tool call results. (PR [#603](#))
8. Require negotiated protocol version to be specified via `MCP-Protocol-Version` header in subsequent requests when using HTTP (PR [#548](#)).
9. Change **SHOULD** to **MUST** in Lifecycle Operation

Other schema changes

1. Add `_meta` field to additional interface types (PR [#710](#)), and specify proper usage.
2. Add `context` field to `CompletionRequest`, providing for completion requests to include previously-resolved variables (PR [#598](#)).
3. Add `title` field for human-friendly display names, so that `name` can be used as a programmatic identifier (PR [#663](#))

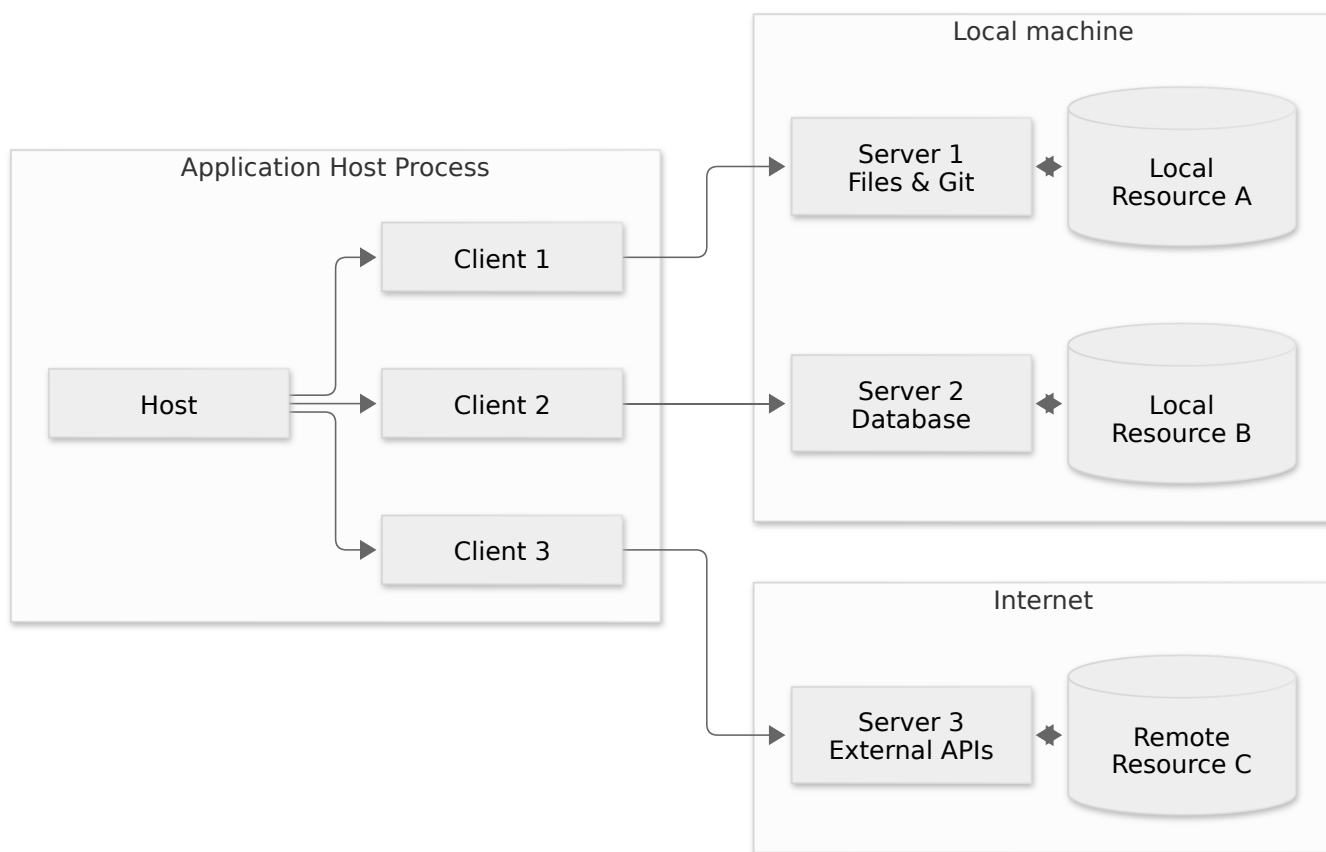
Full changelog

For a complete list of all changes that have been made since the last protocol revision, [see GitHub](#).

3 Architecture

The Model Context Protocol (MCP) follows a client-host-server architecture where each host can run multiple client instances. This architecture enables users to integrate AI capabilities across applications while maintaining clear security boundaries and isolating concerns. Built on JSON-RPC, MCP provides a stateful session protocol focused on context exchange and sampling coordination between clients and servers.

Core Components



Host

The host process acts as the container and coordinator:

- Creates and manages multiple client instances
- Controls client connection permissions and lifecycle
- Enforces security policies and consent requirements
- Handles user authorization decisions

- Coordinates AI/LLM integration and sampling
- Manages context aggregation across clients

Clients

Each client is created by the host and maintains an isolated server connection:

- Establishes one stateful session per server
- Handles protocol negotiation and capability exchange
- Routes protocol messages bidirectionally
- Manages subscriptions and notifications
- Maintains security boundaries between servers

A host application creates and manages multiple clients, with each client having a 1:1 relationship with a particular server.

Servers

Servers provide specialized context and capabilities:

- Expose resources, tools and prompts via MCP primitives
- Operate independently with focused responsibilities
- Request sampling through client interfaces
- Must respect security constraints
- Can be local processes or remote services

Design Principles

MCP is built on several key design principles that inform its architecture and implementation:

1. Servers should be extremely easy to build

- Host applications handle complex orchestration responsibilities
- Servers focus on specific, well-defined capabilities
- Simple interfaces minimize implementation overhead
- Clear separation enables maintainable code

2. Servers should be highly composable

- Each server provides focused functionality in isolation
- Multiple servers can be combined seamlessly
- Shared protocol enables interoperability
- Modular design supports extensibility

3. Servers should not be able to read the whole conversation, nor "see into" other servers

- Servers receive only necessary contextual information
- Full conversation history stays with the host
- Each server connection maintains isolation

- Cross-server interactions are controlled by the host
- Host process enforces security boundaries

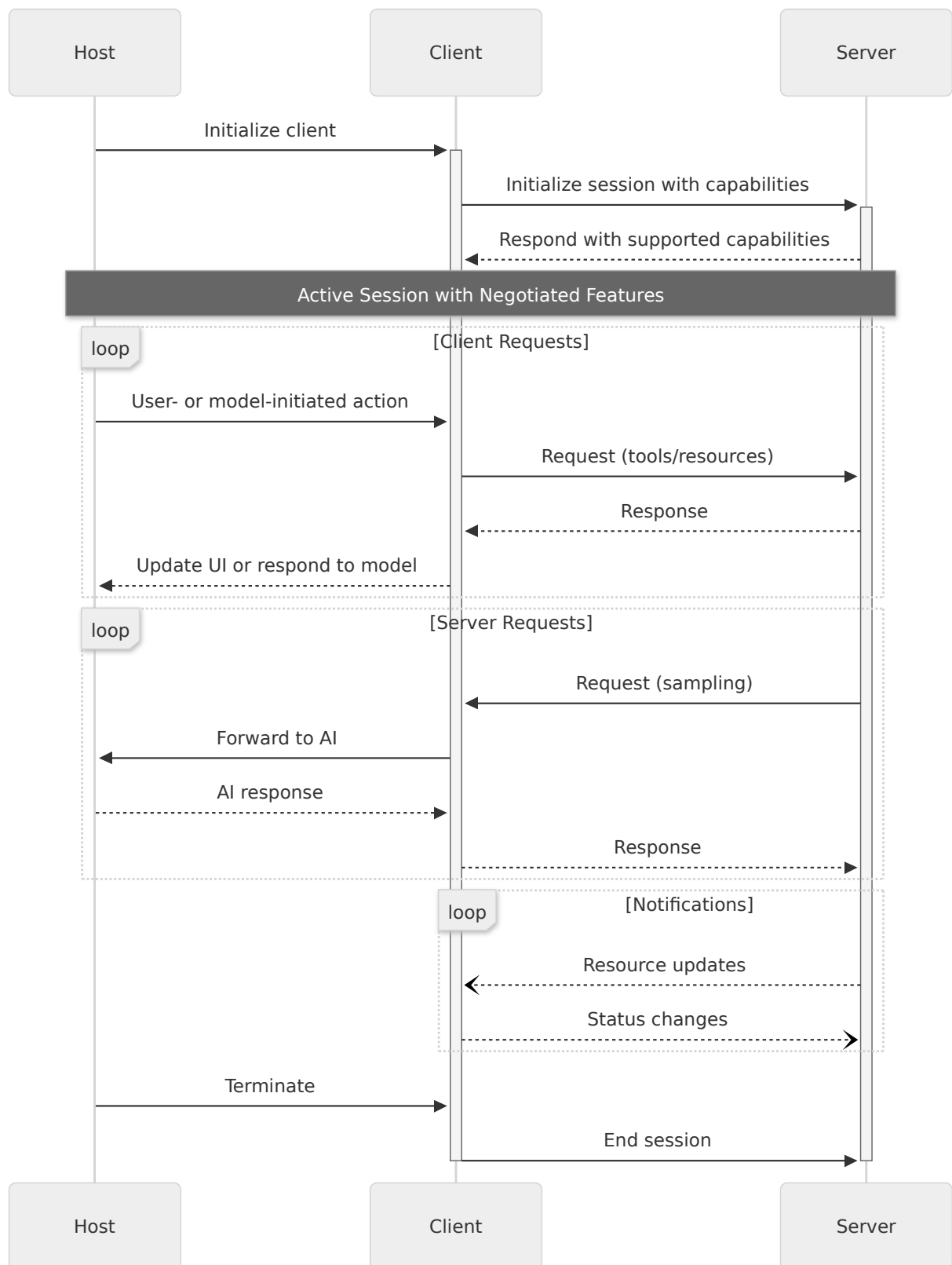
4. Features can be added to servers and clients progressively

- Core protocol provides minimal required functionality
- Additional capabilities can be negotiated as needed
- Servers and clients evolve independently
- Protocol designed for future extensibility
- Backwards compatibility is maintained

Capability Negotiation

The Model Context Protocol uses a capability-based negotiation system where clients and servers explicitly declare their supported features during initialization. Capabilities determine which protocol features and primitives are available during a session.

- Servers declare capabilities like resource subscriptions, tool support, and prompt templates
- Clients declare capabilities like sampling support and notification handling
- Both parties must respect declared capabilities throughout the session
- Additional capabilities can be negotiated through extensions to the protocol



Each capability unlocks specific protocol features for use during the session. For example:

- Implemented server features must be advertised in the server's capabilities
- Emitting resource subscription notifications requires the server to declare subscription support
- Tool invocation requires the server to declare tool capabilities
- Sampling requires the client to declare support in its capabilities

This capability negotiation ensures clients and servers have a clear understanding of supported functionality while maintaining protocol extensibility.

4 Base Protocol

4.1 Overview

The Model Context Protocol consists of several key components that work together:

- **Base Protocol:** Core JSON-RPC message types
- **Lifecycle Management:** Connection initialization, capability negotiation, and session control
- **Authorization:** Authentication and authorization framework for HTTP-based transports
- **Server Features:** Resources, prompts, and tools exposed by servers
- **Client Features:** Sampling and root directory lists provided by clients
- **Utilities:** Cross-cutting concerns like logging and argument completion

All implementations **MUST** support the base protocol and lifecycle management components. Other components **MAY** be implemented based on the specific needs of the application.

These protocol layers establish clear separation of concerns while enabling rich interactions between clients and servers. The modular design allows implementations to support exactly the features they need.

Messages

All messages between MCP clients and servers **MUST** follow the JSON-RPC 2.0 specification. The protocol defines these types of messages:

Requests

Requests are sent from the client to the server or vice versa, to initiate an operation.

```
{
  jsonrpc: "2.0";
  id: string | number;
  method: string;
  params?: {
    [key: string]: unknown;
  };
}
```

- Requests **MUST** include a string or integer ID.
- Unlike base JSON-RPC, the ID **MUST NOT** be `null`.
- The request ID **MUST NOT** have been previously used by the requestor within the same session.

Responses

Responses are sent in reply to requests, containing the result or error of the operation.

```
{
  jsonrpc: "2.0";
  id: string | number;
  result?: {
    [key: string]: unknown;
  }
  error?: {
    code: number;
    message: string;
    data?: unknown;
  }
}
```

- Responses **MUST** include the same ID as the request they correspond to.
- **Responses** are further sub-categorized as either **successful results** or **errors**. Either a `result` or an `error` **MUST** be set. A response **MUST NOT** set both.
- Results **MAY** follow any JSON object structure, while errors **MUST** include an error code and message at minimum.
- Error codes **MUST** be integers.

Notifications

Notifications are sent from the client to the server or vice versa, as a one-way message. The receiver **MUST NOT** send a response.

```
{
  jsonrpc: "2.0";
  method: string;
  params?: {
    [key: string]: unknown;
  };
}
```

- Notifications **MUST NOT** include an ID.

Auth

MCP provides an Authorization framework for use with HTTP. Implementations using an HTTP-based transport **SHOULD** conform to this specification, whereas implementations using STDIO transport **SHOULD NOT** follow this specification, and instead retrieve credentials from the environment.

Additionally, clients and servers **MAY** negotiate their own custom authentication and authorization strategies.

For further discussions and contributions to the evolution of MCP's auth mechanisms, join us in [GitHub Discussions](#) to help shape the future of the protocol!

Schema

The full specification of the protocol is defined as a [TypeScript schema](#). This is the source of truth for all protocol messages and structures.

There is also a [JSON Schema](#), which is automatically generated from the TypeScript source of truth, for use with various automated tooling.

General fields

`_meta`

The `_meta` property/parameter is reserved by MCP to allow clients and servers to attach additional metadata to their interactions.

Certain key names are reserved by MCP for protocol-level metadata, as specified below; implementations **MUST NOT** make assumptions about values at these keys.

Additionally, definitions in the [schema](#) may reserve particular names for purpose-specific metadata, as declared in those definitions.

Key name format: valid `_meta` key names have two segments: an optional **prefix**, and a **name**.

Prefix:

- If specified, **MUST** be a series of labels separated by dots (`.`), followed by a slash (`/`).
 - Labels **MUST** start with a letter and end with a letter or digit; interior characters can be letters, digits, or hyphens (`-`).
- Any prefix beginning with zero or more valid labels, followed by `modelcontextprotocol` or `mcp`, followed by any valid label, is **reserved** for MCP use.
 - For example: `modelcontextprotocol.io/`, `mcp.dev/`, `api.modelcontextprotocol.org/`, and `tools.mcp.com/` are all reserved.

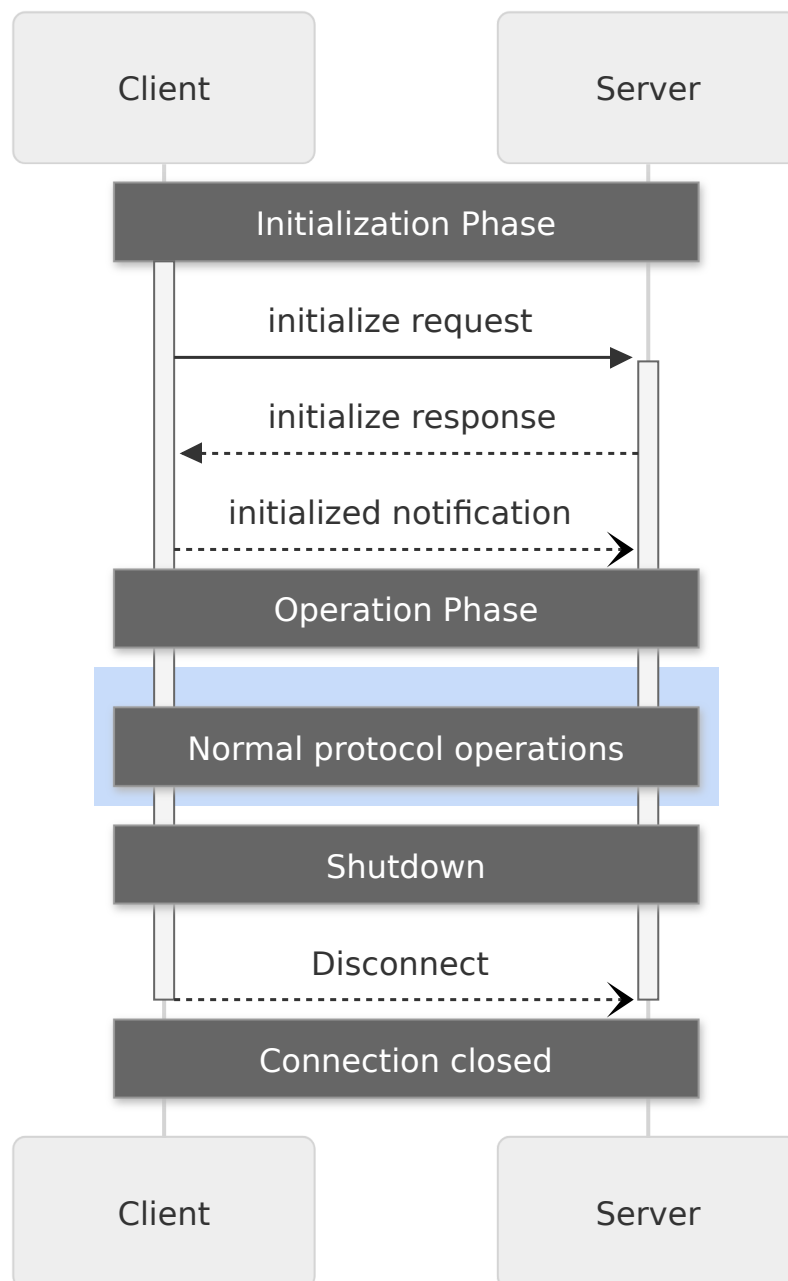
Name:

- Unless empty, **MUST** begin and end with an alphanumeric character (`[a-z0-9A-Z]`).
- **MAY** contain hyphens (`-`), underscores (`_`), dots (`.`), and alphanumerics in between.

4.2 Lifecycle

The Model Context Protocol (MCP) defines a rigorous lifecycle for client-server connections that ensures proper capability negotiation and state management.

1. **Initialization:** Capability negotiation and protocol version agreement
2. **Operation:** Normal protocol communication
3. **Shutdown:** Graceful termination of the connection



Lifecycle Phases

Initialization

The initialization phase **MUST** be the first interaction between client and server. During this phase, the client and server:

- Establish protocol version compatibility
- Exchange and negotiate capabilities
- Share implementation details

The client **MUST** initiate this phase by sending an `initialize` request containing:

- Protocol version supported
- Client capabilities
- Client implementation information

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "initialize",
  "params": {
    "protocolVersion": "2025-06-18",
    "capabilities": {
      "roots": {
        "listChanged": true
      },
      "sampling": {},
      "elicitation": {}
    },
    "clientInfo": {
      "name": "ExampleClient",
      "title": "Example Client Display Name",
      "version": "1.0.0"
    }
  }
}
```

The server **MUST** respond with its own capabilities and information:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "protocolVersion": "2025-06-18",
    "capabilities": {
      "logging": {},
      "prompts": {
        "listChanged": true
      },
      "resources": {
        "subscribe": true,
        "listChanged": true
      },
      "tools": {
        "listChanged": true
      }
    },
    "serverInfo": {
      "name": "ExampleServer",
      "title": "Example Server Display Name",
      "version": "1.0.0"
    },
    "instructions": "Optional instructions for the client"
  }
}
```

After successful initialization, the client **MUST** send an `initialized` notification to indicate it is ready to begin normal operations:

```
{
  "jsonrpc": "2.0",
  "method": "notifications/initialized"
}
```

- The client **SHOULD NOT** send requests other than pings before the server has responded to the `initialize` request.
- The server **SHOULD NOT** send requests other than pings and logging before receiving the `initialized` notification.

Version Negotiation

In the `initialize` request, the client **MUST** send a protocol version it supports. This **SHOULD** be the *latest* version supported by the client.

If the server supports the requested protocol version, it **MUST** respond with the same version. Otherwise, the server **MUST** respond with another protocol version it supports. This **SHOULD** be the *latest* version supported by the server.

If the client does not support the version in the server's response, it **SHOULD** disconnect.

NOTE

If using HTTP, the client **MUST** include the `MCP-Protocol-Version: <protocol-version>` HTTP header on all subsequent requests to the MCP server. For details, see the Protocol Version Header section in Transports.

Capability Negotiation

Client and server capabilities establish which optional protocol features will be available during the session.

Key capabilities include:

Category	Capability	Description
Client	roots	Ability to provide filesystem roots
Client	sampling	Support for LLM sampling requests
Client	elicitation	Support for server elicitation requests
Client	experimental	Describes support for non-standard experimental features
Server	prompts	Offers prompt templates
Server	resources	Provides readable resources
Server	tools	Exposes callable tools
Server	logging	Emits structured log messages
Server	completions	Supports argument autocompletion
Server	experimental	Describes support for non-standard experimental features

Capability objects can describe sub-capabilities like:

- `listChanged` : Support for list change notifications (for prompts, resources, and tools)
- `subscribe` : Support for subscribing to individual items' changes (resources only)

Operation

During the operation phase, the client and server exchange messages according to the negotiated capabilities.

Both parties **MUST**:

- Respect the negotiated protocol version
- Only use capabilities that were successfully negotiated

Shutdown

During the shutdown phase, one side (usually the client) cleanly terminates the protocol connection. No specific shutdown messages are defined—instead, the underlying transport mechanism should be used to signal connection termination:

stdio

For the stdio transport, the client **SHOULD** initiate shutdown by:

1. First, closing the input stream to the child process (the server)
2. Waiting for the server to exit, or sending `SIGTERM` if the server does not exit within a reasonable time
3. Sending `SIGKILL` if the server does not exit within a reasonable time after `SIGTERM`

The server **MAY** initiate shutdown by closing its output stream to the client and exiting.

HTTP

For HTTP transports, shutdown is indicated by closing the associated HTTP connection(s).

Timeouts

Implementations **SHOULD** establish timeouts for all sent requests, to prevent hung connections and resource exhaustion. When the request has not received a success or error response within the timeout period, the sender **SHOULD** issue a cancellation notification for that request and stop waiting for a response.

SDKs and other middleware **SHOULD** allow these timeouts to be configured on a per-request basis.

Implementations **MAY** choose to reset the timeout clock when receiving a progress notification corresponding to the request, as this implies that work is actually happening. However, implementations **SHOULD** always enforce a maximum timeout, regardless of progress notifications, to limit the impact of a misbehaving client or server.

Error Handling

Implementations **SHOULD** be prepared to handle these error cases:

- Protocol version mismatch

- Failure to negotiate required capabilities
- Request timeouts

Example initialization error:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "error": {
    "code": -32602,
    "message": "Unsupported protocol version",
    "data": {
      "supported": ["2024-11-05"],
      "requested": "1.0.0"
    }
  }
}
```

4.3 Transports

MCP uses JSON-RPC to encode messages. JSON-RPC messages **MUST** be UTF-8 encoded.

The protocol currently defines two standard transport mechanisms for client-server communication:

1. stdio, communication over standard in and standard out
2. Streamable HTTP

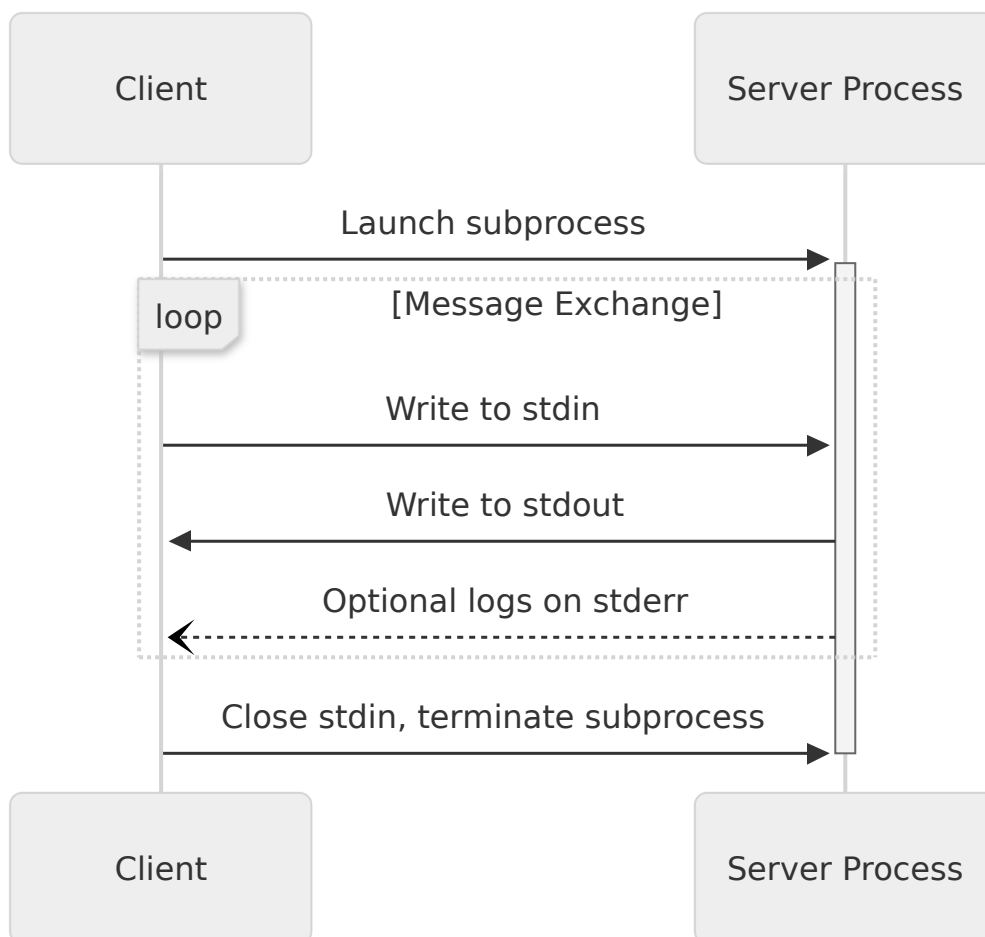
Clients **SHOULD** support stdio whenever possible.

It is also possible for clients and servers to implement custom transports in a pluggable fashion.

stdio

In the **stdio** transport:

- The client launches the MCP server as a subprocess.
- The server reads JSON-RPC messages from its standard input (`stdin`) and sends messages to its standard output (`stdout`).
- Messages are individual JSON-RPC requests, notifications, or responses.
- Messages are delimited by newlines, and **MUST NOT** contain embedded newlines.
- The server **MAY** write UTF-8 strings to its standard error (`stderr`) for logging purposes. Clients **MAY** capture, forward, or ignore this logging.
- The server **MUST NOT** write anything to its `stdout` that is not a valid MCP message.
- The client **MUST NOT** write anything to the server's `stdin` that is not a valid MCP message.



Streamable HTTP

INFO

This replaces the HTTP+SSE transport from protocol version 2024-11-05. See the backwards compatibility guide below.

In the **Streamable HTTP** transport, the server operates as an independent process that can handle multiple client connections. This transport uses HTTP POST and GET requests. Server can optionally make use of Server-Sent Events (SSE) to stream multiple server messages. This permits basic MCP servers, as well as more feature-rich servers supporting streaming and server-to-client notifications and requests.

The server **MUST** provide a single HTTP endpoint path (hereafter referred to as the **MCP endpoint**) that supports both POST and GET methods. For example, this could be a URL like `https://example.com/mcp`.

Security Warning

When implementing Streamable HTTP transport:

1. Servers **MUST** validate the `Origin` header on all incoming connections to prevent DNS rebinding attacks
2. When running locally, servers **SHOULD** bind only to localhost (127.0.0.1) rather than all network interfaces (0.0.0.0)
3. Servers **SHOULD** implement proper authentication for all connections

Without these protections, attackers could use DNS rebinding to interact with local MCP servers from remote websites.

Sending Messages to the Server

Every JSON-RPC message sent from the client **MUST** be a new HTTP POST request to the MCP endpoint.

1. The client **MUST** use HTTP POST to send JSON-RPC messages to the MCP endpoint.
2. The client **MUST** include an `Accept` header, listing both `application/json` and `text/event-stream` as supported content types.
3. The body of the POST request **MUST** be a single JSON-RPC *request*, *notification*, or *response*.
4. If the input is a JSON-RPC *response* or *notification*:
 - If the server accepts the input, the server **MUST** return HTTP status code 202 Accepted with no body.
 - If the server cannot accept the input, it **MUST** return an HTTP error status code (e.g., 400 Bad Request). The HTTP response body **MAY** comprise a JSON-RPC *error response* that has no `id`.
5. If the input is a JSON-RPC *request*, the server **MUST** either return `Content-Type: text/event-stream`, to initiate an SSE stream, or `Content-Type: application/json`, to return one JSON object. The client **MUST** support both these cases.
6. If the server initiates an SSE stream:
 - The SSE stream **SHOULD** eventually include JSON-RPC *response* for the JSON-RPC *request* sent in the POST body.
 - The server **MAY** send JSON-RPC *requests* and *notifications* before sending the JSON-RPC *response*. These messages **SHOULD** relate to the originating client *request*.
 - The server **SHOULD NOT** close the SSE stream before sending the JSON-RPC *response* for the received JSON-RPC *request*, unless the session expires.
 - After the JSON-RPC *response* has been sent, the server **SHOULD** close the SSE stream.
 - Disconnection **MAY** occur at any time (e.g., due to network conditions). Therefore:
 - Disconnection **SHOULD NOT** be interpreted as the client cancelling its request.
 - To cancel, the client **SHOULD** explicitly send an MCP `CancelledNotification`.
 - To avoid message loss due to disconnection, the server **MAY** make the stream resumable.

Listening for Messages from the Server

1. The client **MAY** issue an HTTP GET to the MCP endpoint. This can be used to open an SSE stream, allowing the server to communicate to the client, without the client first sending data via HTTP POST.
2. The client **MUST** include an `Accept` header, listing `text/event-stream` as a supported content type.

3. The server **MUST** either return `Content-Type: text/event-stream` in response to this HTTP GET, or else return HTTP 405 Method Not Allowed, indicating that the server does not offer an SSE stream at this endpoint.
4. If the server initiates an SSE stream:
 - The server **MAY** send JSON-RPC *requests* and *notifications* on the stream.
 - These messages **SHOULD** be unrelated to any concurrently-running JSON-RPC *request* from the client.
 - The server **MUST NOT** send a JSON-RPC *response* on the stream **unless** resuming a stream associated with a previous client request.
 - The server **MAY** close the SSE stream at any time.
 - The client **MAY** close the SSE stream at any time.

Multiple Connections

1. The client **MAY** remain connected to multiple SSE streams simultaneously.
2. The server **MUST** send each of its JSON-RPC messages on only one of the connected streams; that is, it **MUST NOT** broadcast the same message across multiple streams.
 - The risk of message loss **MAY** be mitigated by making the stream resumable.

Resumability and Redelivery

To support resuming broken connections, and redelivering messages that might otherwise be lost:

1. Servers **MAY** attach an `id` field to their SSE events, as described in the [SSE standard](#).
 - If present, the ID **MUST** be globally unique across all streams within that session—or all streams with that specific client, if session management is not in use.
2. If the client wishes to resume after a broken connection, it **SHOULD** issue an HTTP GET to the MCP endpoint, and include the `Last-Event-ID` header to indicate the last event ID it received.
 - The server **MAY** use this header to replay messages that would have been sent after the last event ID, *on the stream that was disconnected*, and to resume the stream from that point.
 - The server **MUST NOT** replay messages that would have been delivered on a different stream.

In other words, these event IDs should be assigned by servers on a *per-stream* basis, to act as a cursor within that particular stream.

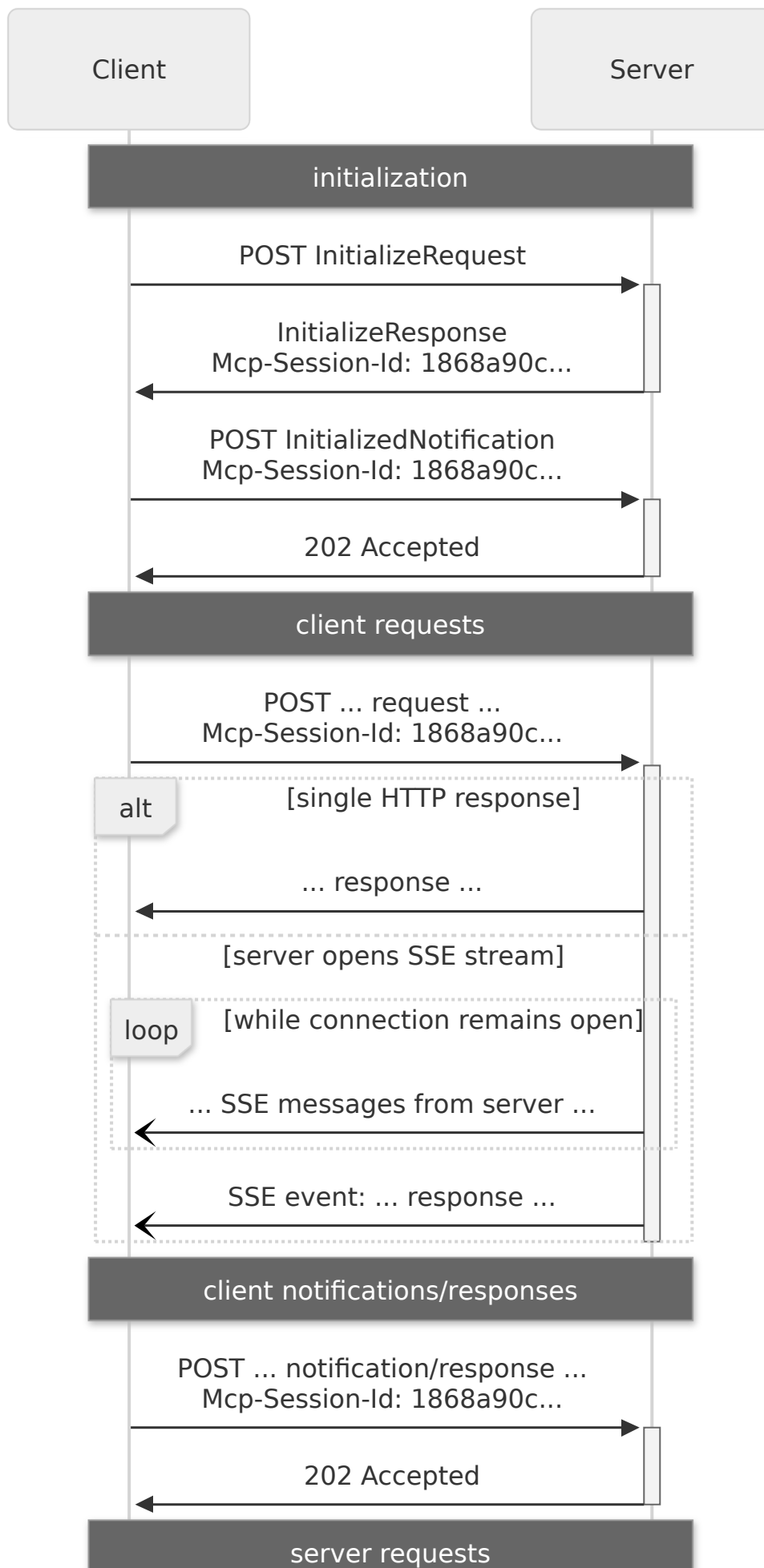
Session Management

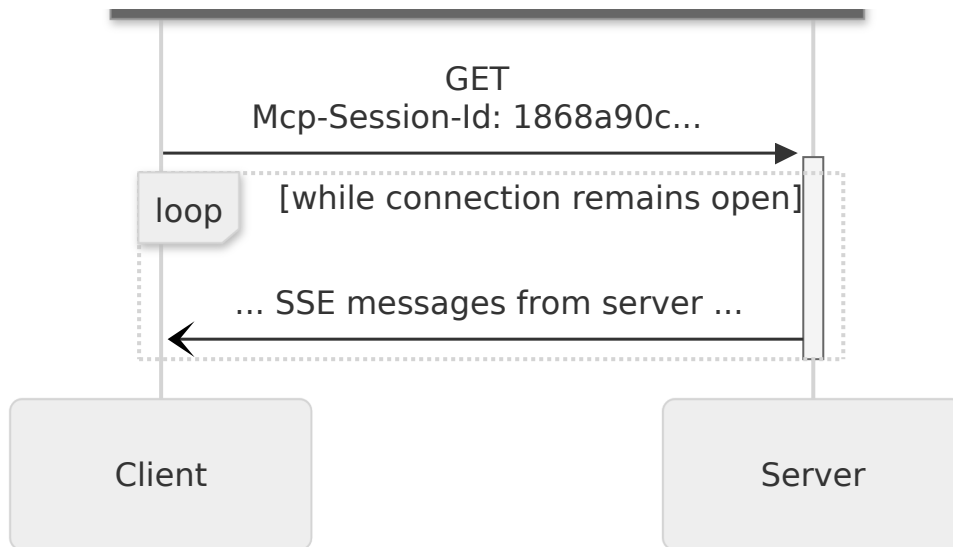
An MCP "session" consists of logically related interactions between a client and a server, beginning with the initialization phase. To support servers which want to establish stateful sessions:

1. A server using the Streamable HTTP transport **MAY** assign a session ID at initialization time, by including it in an `Mcp-Session-Id` header on the HTTP response containing the `InitializeResult`.
 - The session ID **SHOULD** be globally unique and cryptographically secure (e.g., a securely generated UUID, a JWT, or a cryptographic hash).
 - The session ID **MUST** only contain visible ASCII characters (ranging from 0x21 to 0x7E).

2. If an `Mcp-Session-Id` is returned by the server during initialization, clients using the Streamable HTTP transport **MUST** include it in the `Mcp-Session-Id` header on all of their subsequent HTTP requests.
 - Servers that require a session ID **SHOULD** respond to requests without an `Mcp-Session-Id` header (other than initialization) with HTTP 400 Bad Request.
3. The server **MAY** terminate the session at any time, after which it **MUST** respond to requests containing that session ID with HTTP 404 Not Found.
4. When a client receives HTTP 404 in response to a request containing an `Mcp-Session-Id`, it **MUST** start a new session by sending a new `InitializeRequest` without a session ID attached.
5. Clients that no longer need a particular session (e.g., because the user is leaving the client application) **SHOULD** send an HTTP DELETE to the MCP endpoint with the `Mcp-Session-Id` header, to explicitly terminate the session.
 - The server **MAY** respond to this request with HTTP 405 Method Not Allowed, indicating that the server does not allow clients to terminate sessions.

Sequence Diagram





Protocol Version Header

If using HTTP, the client **MUST** include the `MCP-Protocol-Version: <protocol-version>` HTTP header on all subsequent requests to the MCP server, allowing the MCP server to respond based on the MCP protocol version.

For example: `MCP-Protocol-Version: 2025-06-18`

The protocol version sent by the client **SHOULD** be the one negotiated during initialization.

For backwards compatibility, if the server does *not* receive an `MCP-Protocol-Version` header, and has no other way to identify the version - for example, by relying on the protocol version negotiated during initialization - the server **SHOULD** assume protocol version `2025-03-26`.

If the server receives a request with an invalid or unsupported `MCP-Protocol-Version`, it **MUST** respond with `400 Bad Request`.

Backwards Compatibility

Clients and servers can maintain backwards compatibility with the deprecated HTTP+SSE transport (from protocol version 2024-11-05) as follows:

Servers wanting to support older clients should:

- Continue to host both the SSE and POST endpoints of the old transport, alongside the new "MCP endpoint" defined for the Streamable HTTP transport.
 - It is also possible to combine the old POST endpoint and the new MCP endpoint, but this may introduce unneeded complexity.

Clients wanting to support older servers should:

1. Accept an MCP server URL from the user, which may point to either a server using the old transport or the new transport.
2. Attempt to POST an `InitializeRequest` to the server URL, with an `Accept` header as defined above:

- If it succeeds, the client can assume this is a server supporting the new Streamable HTTP transport.
- If it fails with an HTTP 4xx status code (e.g., 405 Method Not Allowed or 404 Not Found):
 - Issue a GET request to the server URL, expecting that this will open an SSE stream and return an `endpoint` event as the first event.
 - When the `endpoint` event arrives, the client can assume this is a server running the old HTTP+SSE transport, and should use that transport for all subsequent communication.

Custom Transports

Clients and servers **MAY** implement additional custom transport mechanisms to suit their specific needs. The protocol is transport-agnostic and can be implemented over any communication channel that supports bidirectional message exchange.

Implementers who choose to support custom transports **MUST** ensure they preserve the JSON-RPC message format and lifecycle requirements defined by MCP. Custom transports **SHOULD** document their specific connection establishment and message exchange patterns to aid interoperability.

4.4 Authorization

Introduction

Purpose and Scope

The Model Context Protocol provides authorization capabilities at the transport level, enabling MCP clients to make requests to restricted MCP servers on behalf of resource owners. This specification defines the authorization flow for HTTP-based transports.

Protocol Requirements

Authorization is **OPTIONAL** for MCP implementations. When supported:

- Implementations using an HTTP-based transport **SHOULD** conform to this specification.
- Implementations using an STDIO transport **SHOULD NOT** follow this specification, and instead retrieve credentials from the environment.
- Implementations using alternative transports **MUST** follow established security best practices for their protocol.

Standards Compliance

This authorization mechanism is based on established specifications listed below, but implements a selected subset of their features to ensure security and interoperability while maintaining simplicity:

- OAuth 2.1 IETF DRAFT ([draft-ietf-oauth-v2-1-13](#))
- OAuth 2.0 Authorization Server Metadata ([RFC8414](#))
- OAuth 2.0 Dynamic Client Registration Protocol ([RFC7591](#))
- OAuth 2.0 Protected Resource Metadata ([RFC9728](#))

Authorization Flow

Roles

A protected *MCP server* acts as an [OAuth 2.1 resource server](#), capable of accepting and responding to protected resource requests using access tokens.

An *MCP client* acts as an [OAuth 2.1 client](#), making protected resource requests on behalf of a resource owner.

The *authorization server* is responsible for interacting with the user (if necessary) and issuing access tokens for use at the MCP server. The implementation details of the authorization server are

beyond the scope of this specification. It may be hosted with the resource server or a separate entity. The Authorization Server Discovery section specifies how an MCP server indicates the location of its corresponding authorization server to a client.

Overview

1. Authorization servers **MUST** implement OAuth 2.1 with appropriate security measures for both confidential and public clients.
2. Authorization servers and MCP clients **SHOULD** support the OAuth 2.0 Dynamic Client Registration Protocol ([RFC7591](#)).
3. MCP servers **MUST** implement OAuth 2.0 Protected Resource Metadata ([RFC9728](#)). MCP clients **MUST** use OAuth 2.0 Protected Resource Metadata for authorization server discovery.
4. Authorization servers **MUST** provide OAuth 2.0 Authorization Server Metadata ([RFC8414](#)). MCP clients **MUST** use the OAuth 2.0 Authorization Server Metadata.

Authorization Server Discovery

This section describes the mechanisms by which MCP servers advertise their associated authorization servers to MCP clients, as well as the discovery process through which MCP clients can determine authorization server endpoints and supported capabilities.

Authorization Server Location

MCP servers **MUST** implement the OAuth 2.0 Protected Resource Metadata ([RFC9728](#)) specification to indicate the locations of authorization servers. The Protected Resource Metadata document returned by the MCP server **MUST** include the `authorization_servers` field containing at least one authorization server.

The specific use of `authorization_servers` is beyond the scope of this specification; implementers should consult OAuth 2.0 Protected Resource Metadata ([RFC9728](#)) for guidance on implementation details.

Implementors should note that Protected Resource Metadata documents can define multiple authorization servers. The responsibility for selecting which authorization server to use lies with the MCP client, following the guidelines specified in [RFC9728 Section 7.6 "Authorization Servers"](#).

MCP servers **MUST** use the HTTP header `WWW-Authenticate` when returning a *401 Unauthorized* to indicate the location of the resource server metadata URL as described in [RFC9728 Section 5.1 "WWW-Authenticate Response"](#).

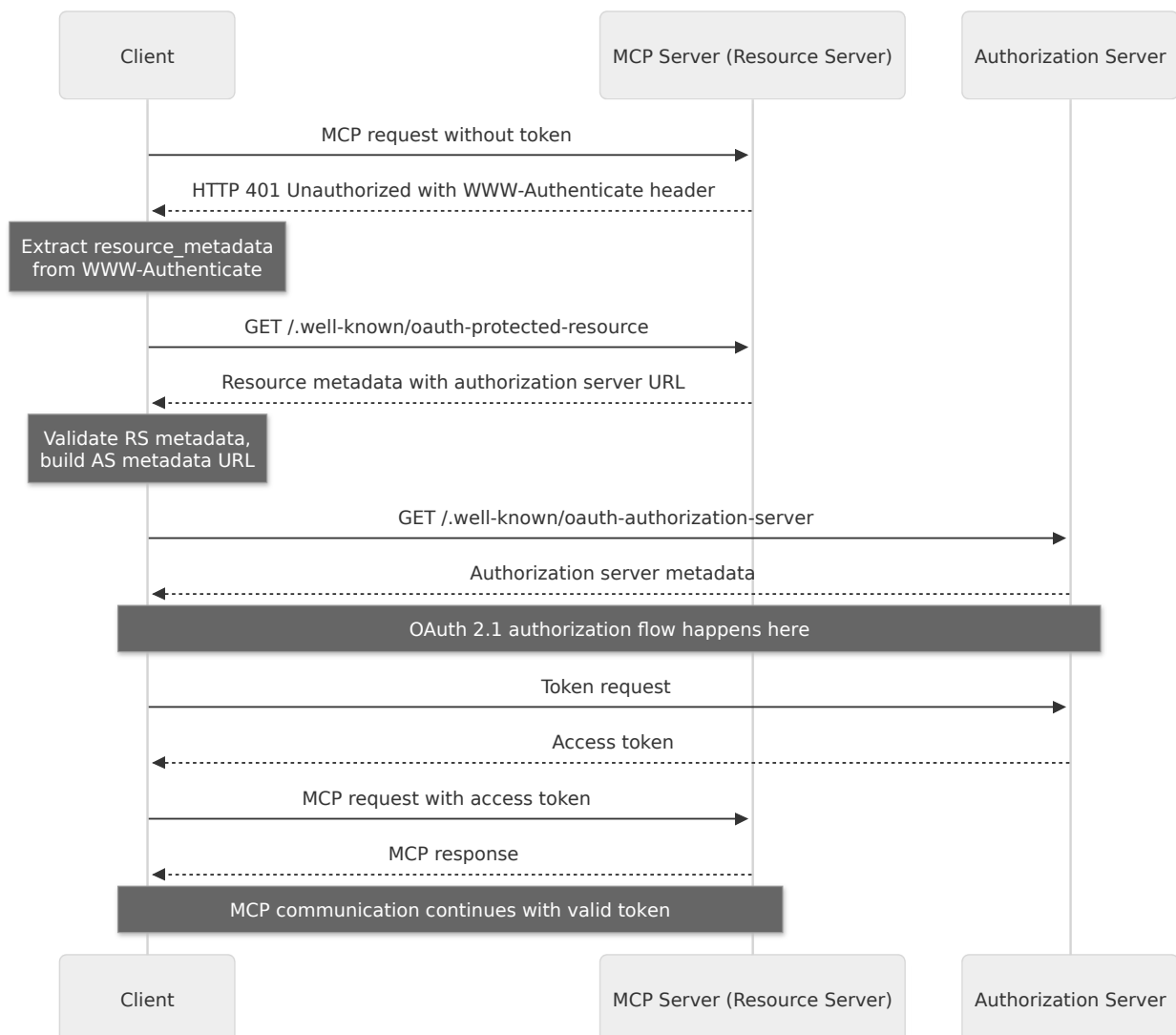
MCP clients **MUST** be able to parse `WWW-Authenticate` headers and respond appropriately to `HTTP 401 Unauthorized` responses from the MCP server.

Server Metadata Discovery

MCP clients **MUST** follow the OAuth 2.0 Authorization Server Metadata [RFC8414](#) specification to obtain the information required to interact with the authorization server.

Sequence Diagram

The following diagram outlines an example flow:



Dynamic Client Registration

MCP clients and authorization servers **SHOULD** support the OAuth 2.0 Dynamic Client Registration Protocol [RFC7591](#) to allow MCP clients to obtain OAuth client IDs without user interaction. This provides a standardized way for clients to automatically register with new authorization servers, which is crucial for MCP because:

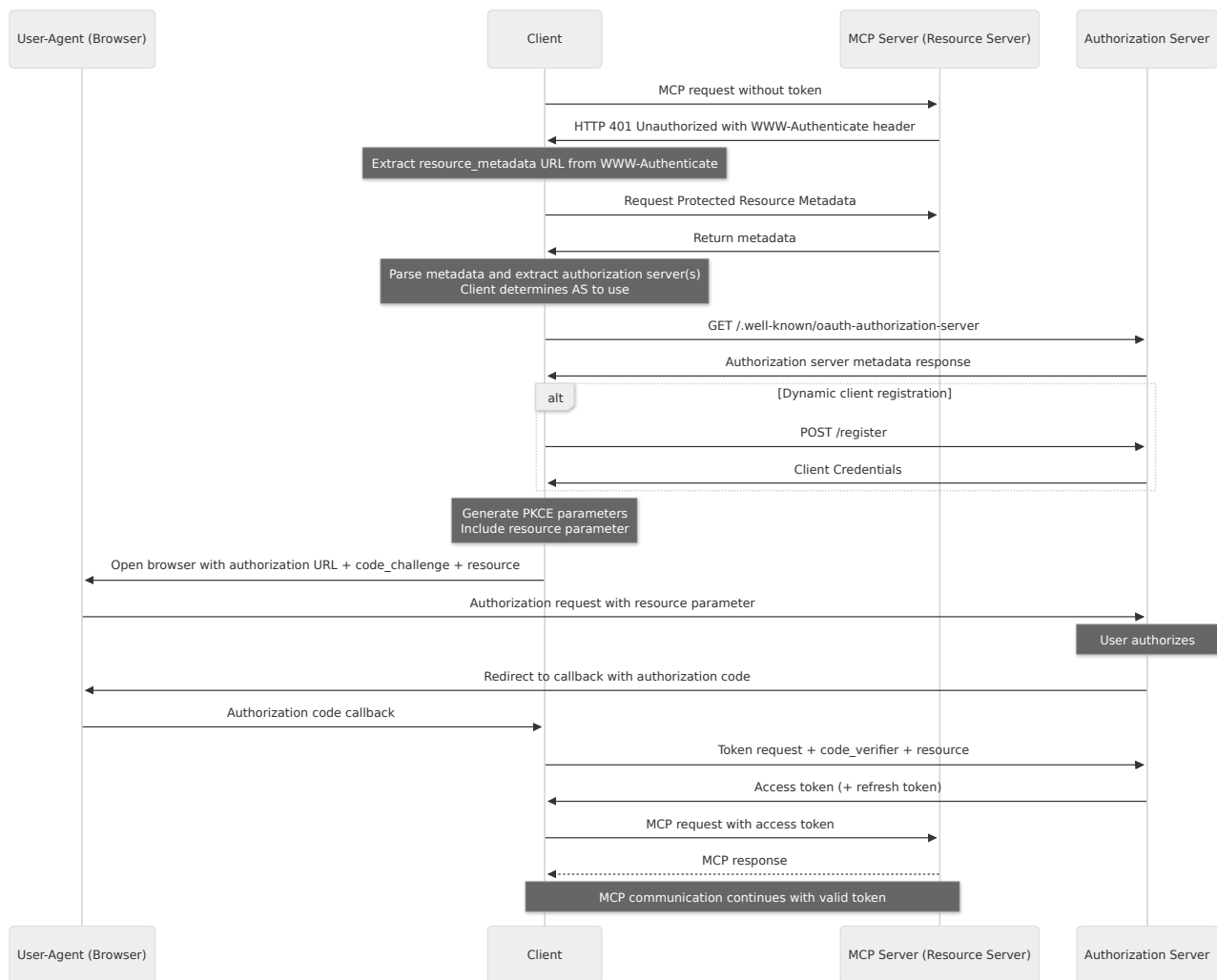
- Clients may not know all possible MCP servers and their authorization servers in advance.
- Manual registration would create friction for users.
- It enables seamless connection to new MCP servers and their authorization servers.
- Authorization servers can implement their own registration policies.

Any authorization servers that *do not* support Dynamic Client Registration need to provide alternative ways to obtain a client ID (and, if applicable, client credentials). For one of these authorization servers, MCP clients will have to either:

1. Hardcode a client ID (and, if applicable, client credentials) specifically for the MCP client to use when interacting with that authorization server, or
2. Present a UI to users that allows them to enter these details, after registering an OAuth client themselves (e.g., through a configuration interface hosted by the server).

Authorization Flow Steps

The complete Authorization flow proceeds as follows:



Resource Parameter Implementation

MCP clients **MUST** implement Resource Indicators for OAuth 2.0 as defined in [RFC 8707](#) to explicitly specify the target resource for which the token is being requested. The `resource` parameter:

1. **MUST** be included in both authorization requests and token requests.
2. **MUST** identify the MCP server that the client intends to use the token with.
3. **MUST** use the canonical URI of the MCP server as defined in [RFC 8707 Section 2](#).

Canonical Server URI

For the purposes of this specification, the canonical URI of an MCP server is defined as the resource identifier as specified in [RFC 8707 Section 2](#) and aligns with the `resource` parameter in [RFC 9728](#).

MCP clients **SHOULD** provide the most specific URI that they can for the MCP server they intend to access, following the guidance in [RFC 8707](#). While the canonical form uses lowercase scheme and host components, implementations **SHOULD** accept uppercase scheme and host components for robustness and interoperability.

Examples of valid canonical URIs:

- `https://mcp.example.com/mcp`
- `https://mcp.example.com`
- `https://mcp.example.com:8443`
- `https://mcp.example.com/server/mcp` (when path component is necessary to identify individual MCP server)

Examples of invalid canonical URIs:

- `mcp.example.com` (missing scheme)
- `https://mcp.example.com#fragment` (contains fragment)

Note: While both `https://mcp.example.com/` (with trailing slash) and `https://mcp.example.com` (without trailing slash) are technically valid absolute URIs according to [RFC 3986](#), implementations **SHOULD** consistently use the form without the trailing slash for better interoperability unless the trailing slash is semantically significant for the specific resource.

For example, if accessing an MCP server at `https://mcp.example.com`, the authorization request would include:

```
&resource=https%3A%2F%2Fmcp.example.com
```

MCP clients **MUST** send this parameter regardless of whether authorization servers support it.

Access Token Usage

Token Requirements

Access token handling when making requests to MCP servers **MUST** conform to the requirements defined in [OAuth 2.1 Section 5 "Resource Requests"](#). Specifically:

1. MCP client **MUST** use the Authorization request header field defined in [OAuth 2.1 Section 5.1.1](#):

```
Authorization: Bearer <access-token>
```

Note that authorization **MUST** be included in every HTTP request from client to server, even if they are part of the same logical session.

- 2. Access tokens **MUST NOT** be included in the URI query string

Example request:

```
GET /mcp HTTP/1.1
Host: mcp.example.com
Authorization: Bearer eyJhbGciOiJIUzI1NiIs...
```

Token Handling

MCP servers, acting in their role as an OAuth 2.1 resource server, **MUST** validate access tokens as described in [OAuth 2.1 Section 5.2](#). MCP servers **MUST** validate that access tokens were issued specifically for them as the intended audience, according to [RFC 8707 Section 2](#). If validation fails, servers **MUST** respond according to [OAuth 2.1 Section 5.3](#) error handling requirements. Invalid or expired tokens **MUST** receive a HTTP 401 response.

MCP clients **MUST NOT** send tokens to the MCP server other than ones issued by the MCP server's authorization server.

Authorization servers **MUST** only accept tokens that are valid for use with their own resources.

MCP servers **MUST NOT** accept or transit any other tokens.

Error Handling

Servers **MUST** return appropriate HTTP status codes for authorization errors:

Status Code	Description	Usage
401	Unauthorized	Authorization required or token invalid
403	Forbidden	Invalid scopes or insufficient permissions
400	Bad Request	Malformed authorization request

Security Considerations

Implementations **MUST** follow OAuth 2.1 security best practices as laid out in [OAuth 2.1 Section 7](#). "[Security Considerations](#)".

Token Audience Binding and Validation

[RFC 8707](#) Resource Indicators provide critical security benefits by binding tokens to their intended audiences **when the Authorization Server supports the capability**. To enable current and

future adoption:

- MCP clients **MUST** include the `resource` parameter in authorization and token requests as specified in the Resource Parameter Implementation section
- MCP servers **MUST** validate that tokens presented to them were specifically issued for their use

The Security Best Practices document outlines why token audience validation is crucial and why token passthrough is explicitly forbidden.

Token Theft

Attackers who obtain tokens stored by the client, or tokens cached or logged on the server can access protected resources with requests that appear legitimate to resource servers.

Clients and servers **MUST** implement secure token storage and follow OAuth best practices, as outlined in [OAuth 2.1, Section 7.1](#).

Authorization servers **SHOULD** issue short-lived access tokens to reduce the impact of leaked tokens. For public clients, authorization servers **MUST** rotate refresh tokens as described in [OAuth 2.1 Section 4.3.1 "Token Endpoint Extension"](#).

Communication Security

Implementations **MUST** follow [OAuth 2.1 Section 1.5 "Communication Security"](#).

Specifically:

1. All authorization server endpoints **MUST** be served over HTTPS.
2. All redirect URIs **MUST** be either `localhost` or use HTTPS.

Authorization Code Protection

An attacker who has gained access to an authorization code contained in an authorization response can try to redeem the authorization code for an access token or otherwise make use of the authorization code. (Further described in [OAuth 2.1 Section 7.5](#))

To mitigate this, MCP clients **MUST** implement PKCE according to [OAuth 2.1 Section 7.5.2](#). PKCE helps prevent authorization code interception and injection attacks by requiring clients to create a secret verifier-challenge pair, ensuring that only the original requestor can exchange an authorization code for tokens.

Open Redirection

An attacker may craft malicious redirect URIs to direct users to phishing sites.

MCP clients **MUST** have redirect URIs registered with the authorization server.

Authorization servers **MUST** validate exact redirect URIs against pre-registered values to prevent redirection attacks.

MCP clients **SHOULD** use and verify state parameters in the authorization code flow and discard any results that do not include or have a mismatch with the original state.

Authorization servers **MUST** take precautions to prevent redirecting user agents to untrusted URI's, following suggestions laid out in [OAuth 2.1 Section 7.12.2](#)

Authorization servers **SHOULD** only automatically redirect the user agent if it trusts the redirection URI. If the URI is not trusted, the authorization server MAY inform the user and rely on the user to make the correct decision.

Confused Deputy Problem

Attackers can exploit MCP servers acting as intermediaries to third-party APIs, leading to confused deputy vulnerabilities. By using stolen authorization codes, they can obtain access tokens without user consent.

MCP proxy servers using static client IDs **MUST** obtain user consent for each dynamically registered client before forwarding to third-party authorization servers (which may require additional consent).

Access Token Privilege Restriction

An attacker can gain unauthorized access or otherwise compromise an MCP server if the server accepts tokens issued for other resources.

This vulnerability has two critical dimensions:

1. **Audience validation failures.** When an MCP server doesn't verify that tokens were specifically intended for it (for example, via the audience claim, as mentioned in [RFC9068](#)), it may accept tokens originally issued for other services. This breaks a fundamental OAuth security boundary, allowing attackers to reuse legitimate tokens across different services than intended.
2. **Token passthrough.** If the MCP server not only accepts tokens with incorrect audiences but also forwards these unmodified tokens to downstream services, it can potentially cause the "confused deputy" problem, where the downstream API may incorrectly trust the token as if it came from the MCP server or assume the token was validated by the upstream API. See the Token Passthrough section of the Security Best Practices guide for additional details.

MCP servers **MUST** validate access tokens before processing the request, ensuring the access token is issued specifically for the MCP server, and take all necessary steps to ensure no data is returned to unauthorized parties.

A MCP server **MUST** follow the guidelines in [OAuth 2.1 - Section 5.2](#) to validate inbound tokens.

MCP servers **MUST** only accept tokens specifically intended for themselves and **MUST** reject tokens that do not include them in the audience claim or otherwise verify that they are the intended recipient of the token. See the Security Best Practices Token Passthrough section for details.

If the MCP server makes requests to upstream APIs, it may act as an OAuth client to them. The access token used at the upstream API is a separate token, issued by the upstream authorization server. The MCP server **MUST NOT** pass through the token it received from the MCP client.

MCP clients **MUST** implement and use the `resource` parameter as defined in [RFC 8707 - Resource Indicators for OAuth 2.0](#) to explicitly specify the target resource for which the token is being requested. This requirement aligns with the recommendation in [RFC 9728 Section 7.4](#). This ensures that access tokens are bound to their intended resources and cannot be misused across different services.

4.5 Utilities

4.5.1 Cancellation

The Model Context Protocol (MCP) supports optional cancellation of in-progress requests through notification messages. Either side can send a cancellation notification to indicate that a previously-issued request should be terminated.

Cancellation Flow

When a party wants to cancel an in-progress request, it sends a `notifications/cancelled` notification containing:

- The ID of the request to cancel
- An optional reason string that can be logged or displayed

```
{
  "jsonrpc": "2.0",
  "method": "notifications/cancelled",
  "params": {
    "requestId": "123",
    "reason": "User requested cancellation"
  }
}
```

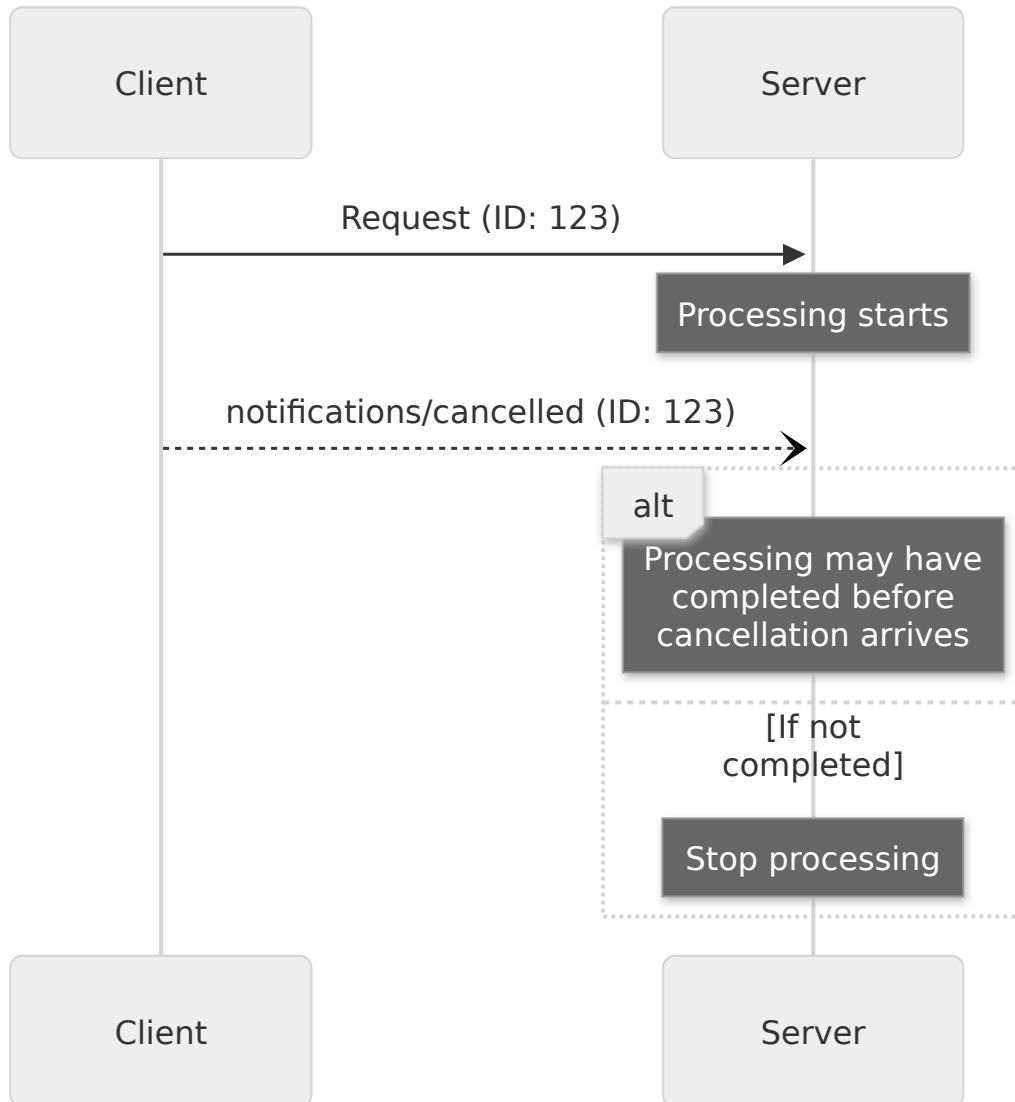
Behavior Requirements

1. Cancellation notifications **MUST** only reference requests that:
 - Were previously issued in the same direction
 - Are believed to still be in-progress
2. The `initialize` request **MUST NOT** be cancelled by clients
3. Receivers of cancellation notifications **SHOULD**:
 - Stop processing the cancelled request
 - Free associated resources
 - Not send a response for the cancelled request
4. Receivers **MAY** ignore cancellation notifications if:
 - The referenced request is unknown
 - Processing has already completed
 - The request cannot be cancelled
5. The sender of the cancellation notification **SHOULD** ignore any response to the request that arrives afterward

Timing Considerations

Due to network latency, cancellation notifications may arrive after request processing has completed, and potentially after a response has already been sent.

Both parties **MUST** handle these race conditions gracefully:



Implementation Notes

- Both parties **SHOULD** log cancellation reasons for debugging
- Application UIs **SHOULD** indicate when cancellation is requested

Error Handling

Invalid cancellation notifications **SHOULD** be ignored:

- Unknown request IDs
- Already completed requests

- Malformed notifications

This maintains the "fire and forget" nature of notifications while allowing for race conditions in asynchronous communication.

4.5.2 Ping

The Model Context Protocol includes an optional ping mechanism that allows either party to verify that their counterpart is still responsive and the connection is alive.

Overview

The ping functionality is implemented through a simple request/response pattern. Either the client or server can initiate a ping by sending a `ping` request.

Message Format

A ping request is a standard JSON-RPC request with no parameters:

```
{
  "jsonrpc": "2.0",
  "id": "123",
  "method": "ping"
}
```

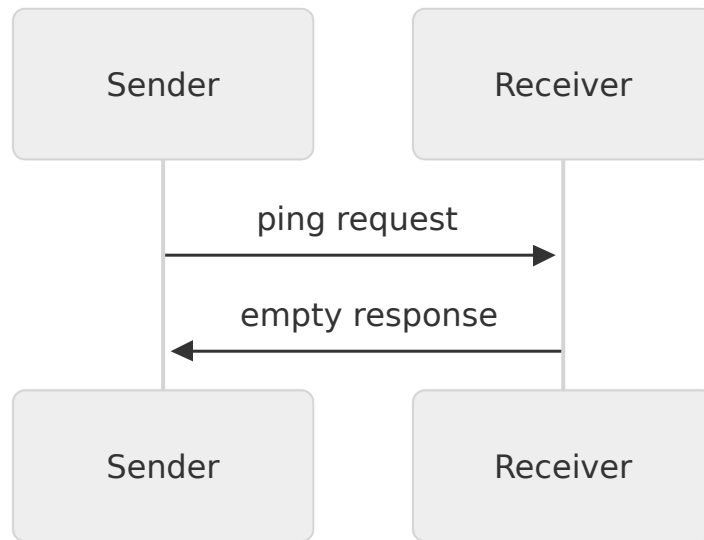
Behavior Requirements

1. The receiver **MUST** respond promptly with an empty response:

```
{
  "jsonrpc": "2.0",
  "id": "123",
  "result": {}
}
```

2. If no response is received within a reasonable timeout period, the sender **MAY**:
 - Consider the connection stale
 - Terminate the connection
 - Attempt reconnection procedures

Usage Patterns



Implementation Considerations

- Implementations **SHOULD** periodically issue pings to detect connection health
- The frequency of pings **SHOULD** be configurable
- Timeouts **SHOULD** be appropriate for the network environment
- Excessive pinging **SHOULD** be avoided to reduce network overhead

Error Handling

- Timeouts **SHOULD** be treated as connection failures
- Multiple failed pings **MAY** trigger connection reset
- Implementations **SHOULD** log ping failures for diagnostics

4.5.3 Progress

The Model Context Protocol (MCP) supports optional progress tracking for long-running operations through notification messages. Either side can send progress notifications to provide updates about operation status.

Progress Flow

When a party wants to *receive* progress updates for a request, it includes a `progressToken` in the request metadata.

- Progress tokens **MUST** be a string or integer value
- Progress tokens can be chosen by the sender using any means, but **MUST** be unique across all active requests.

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "some_method",
  "params": {
    "_meta": {
      "progressToken": "abc123"
    }
  }
}
```

The receiver **MAY** then send progress notifications containing:

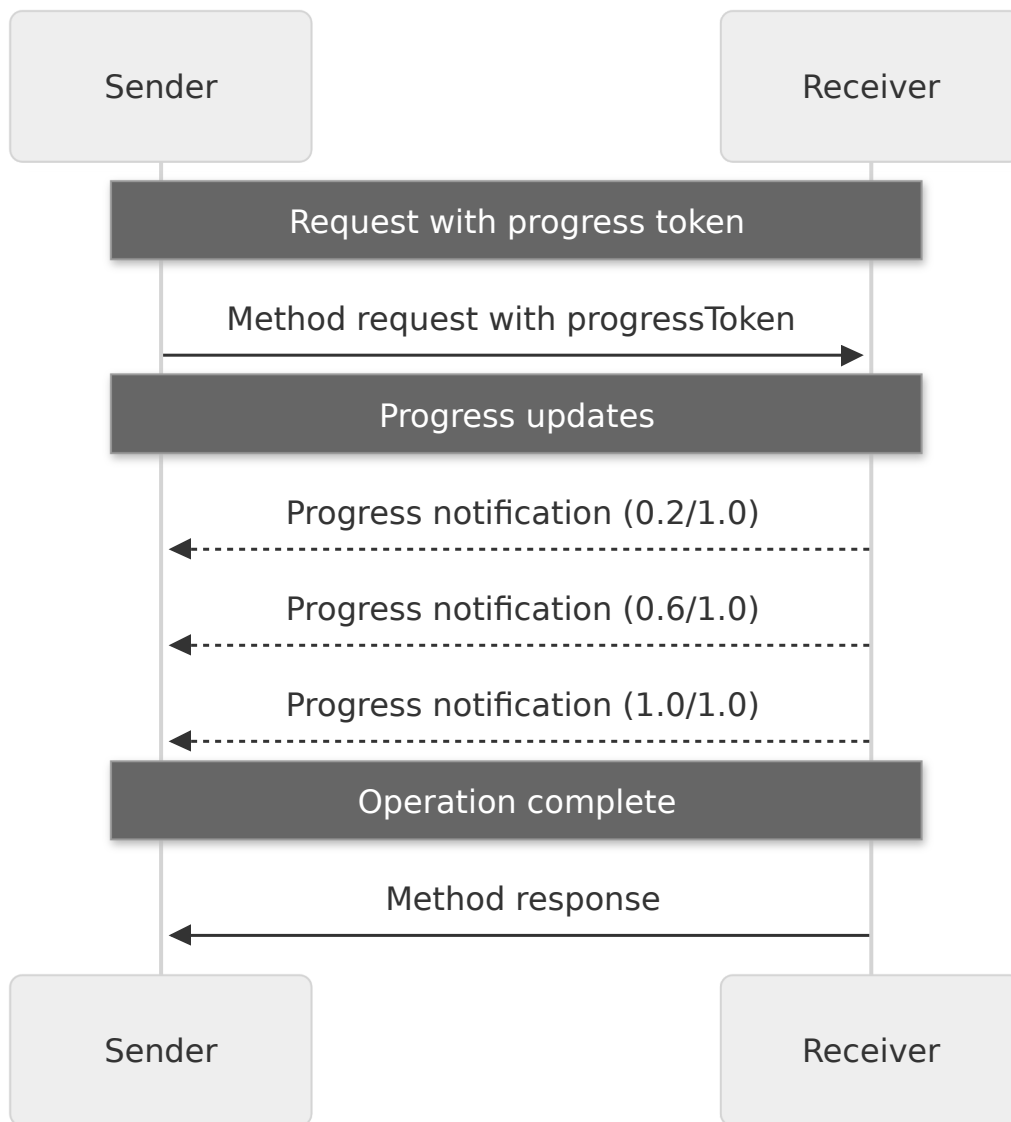
- The original progress token
- The current progress value so far
- An optional "total" value
- An optional "message" value

```
{
  "jsonrpc": "2.0",
  "method": "notifications/progress",
  "params": {
    "progressToken": "abc123",
    "progress": 50,
    "total": 100,
    "message": "Reticulating splines..."
  }
}
```

- The `progress` value **MUST** increase with each notification, even if the total is unknown.
- The `progress` and the `total` values **MAY** be floating point.
- The `message` field **SHOULD** provide relevant human readable progress information.

Behavior Requirements

1. Progress notifications **MUST** only reference tokens that:
 - Were provided in an active request
 - Are associated with an in-progress operation
2. Receivers of progress requests **MAY**:
 - Choose not to send any progress notifications
 - Send notifications at whatever frequency they deem appropriate
 - Omit the total value if unknown



Implementation Notes

- Senders and receivers **SHOULD** track active progress tokens
- Both parties **SHOULD** implement rate limiting to prevent flooding
- Progress notifications **MUST** stop after completion

5 Client Features

5.1 Roots

The Model Context Protocol (MCP) provides a standardized way for clients to expose filesystem "roots" to servers. Roots define the boundaries of where servers can operate within the filesystem, allowing them to understand which directories and files they have access to. Servers can request the list of roots from supporting clients and receive notifications when that list changes.

User Interaction Model

Roots in MCP are typically exposed through workspace or project configuration interfaces.

For example, implementations could offer a workspace/project picker that allows users to select directories and files the server should have access to. This can be combined with automatic workspace detection from version control systems or project files.

However, implementations are free to expose roots through any interface pattern that suits their needs—the protocol itself does not mandate any specific user interaction model.

Capabilities

Clients that support roots **MUST** declare the `roots` capability during initialization:

```
{
  "capabilities": {
    "roots": {
      "listChanged": true
    }
  }
}
```

`listChanged` indicates whether the client will emit notifications when the list of roots changes.

Protocol Messages

Listing Roots

To retrieve roots, servers send a `roots/list` request:

Request:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "roots/list"
}
```

Response:

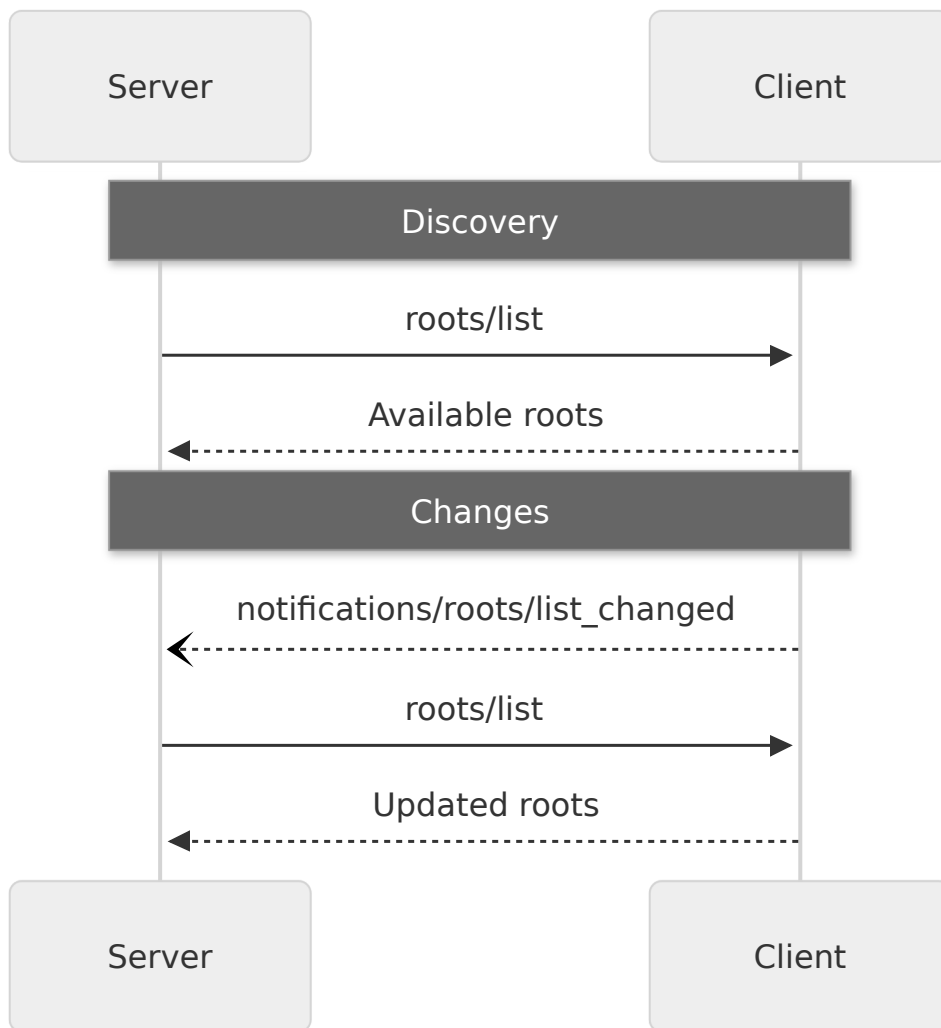
```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "roots": [
      {
        "uri": "file:///home/user/projects/myproject",
        "name": "My Project"
      }
    ]
  }
}
```

Root List Changes

When roots change, clients that support `listChanged` **MUST** send a notification:

```
{
  "jsonrpc": "2.0",
  "method": "notifications/roots/list_changed"
}
```

Message Flow



Data Types

Root

A root definition includes:

- `uri` : Unique identifier for the root. This **MUST** be a `file://` URI in the current specification.
- `name` : Optional human-readable name for display purposes.

Example roots for different use cases:

Project Directory

```
{
  "uri": "file:///home/user/projects/myproject",
  "name": "My Project"
}
```

Multiple Repositories

```
[
  {
    "uri": "file:///home/user/repos/frontend",
    "name": "Frontend Repository"
  },
  {
    "uri": "file:///home/user/repos/backend",
    "name": "Backend Repository"
  }
]
```

Error Handling

Clients **SHOULD** return standard JSON-RPC errors for common failure cases:

- Client does not support roots: `-32601` (Method not found)
- Internal errors: `-32603`

Example error:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "error": {
    "code": -32601,
    "message": "Roots not supported",
    "data": {
      "reason": "Client does not have roots capability"
    }
  }
}
```

Security Considerations

1. Clients **MUST**:

- Only expose roots with appropriate permissions
- Validate all root URIs to prevent path traversal
- Implement proper access controls
- Monitor root accessibility

2. Servers **SHOULD**:

- Handle cases where roots become unavailable
- Respect root boundaries during operations
- Validate all paths against provided roots

Implementation Guidelines

1. Clients **SHOULD**:

- Prompt users for consent before exposing roots to servers
- Provide clear user interfaces for root management
- Validate root accessibility before exposing
- Monitor for root changes

2. Servers **SHOULD**:

- Check for roots capability before usage
- Handle root list changes gracefully
- Respect root boundaries in operations
- Cache root information appropriately

5.2 Sampling

The Model Context Protocol (MCP) provides a standardized way for servers to request LLM sampling ("completions" or "generations") from language models via clients. This flow allows clients to maintain control over model access, selection, and permissions while enabling servers to leverage AI capabilities—with no server API keys necessary. Servers can request text, audio, or image-based interactions and optionally include context from MCP servers in their prompts.

User Interaction Model

Sampling in MCP allows servers to implement agentic behaviors, by enabling LLM calls to occur *nested* inside other MCP server features.

Implementations are free to expose sampling through any interface pattern that suits their needs—the protocol itself does not mandate any specific user interaction model.

WARNING

For trust & safety and security, there **SHOULD** always be a human in the loop with the ability to deny sampling requests.

Applications **SHOULD**:

- Provide UI that makes it easy and intuitive to review sampling requests
- Allow users to view and edit prompts before sending
- Present generated responses for review before delivery

Capabilities

Clients that support sampling **MUST** declare the `sampling` capability during initialization:

```
{
  "capabilities": {
    "sampling": {}
  }
}
```

Protocol Messages

Creating Messages

To request a language model generation, servers send a `sampling/createMessage` request:

Request:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "sampling/createMessage",
  "params": {
    "messages": [
      {
        "role": "user",
        "content": {
          "type": "text",
          "text": "What is the capital of France?"
        }
      }
    ],
    "modelPreferences": {
      "hints": [
        {
          "name": "claude-3-sonnet"
        }
      ],
      "intelligencePriority": 0.8,
      "speedPriority": 0.5
    },
    "systemPrompt": "You are a helpful assistant.",
    "maxTokens": 100
  }
}
```

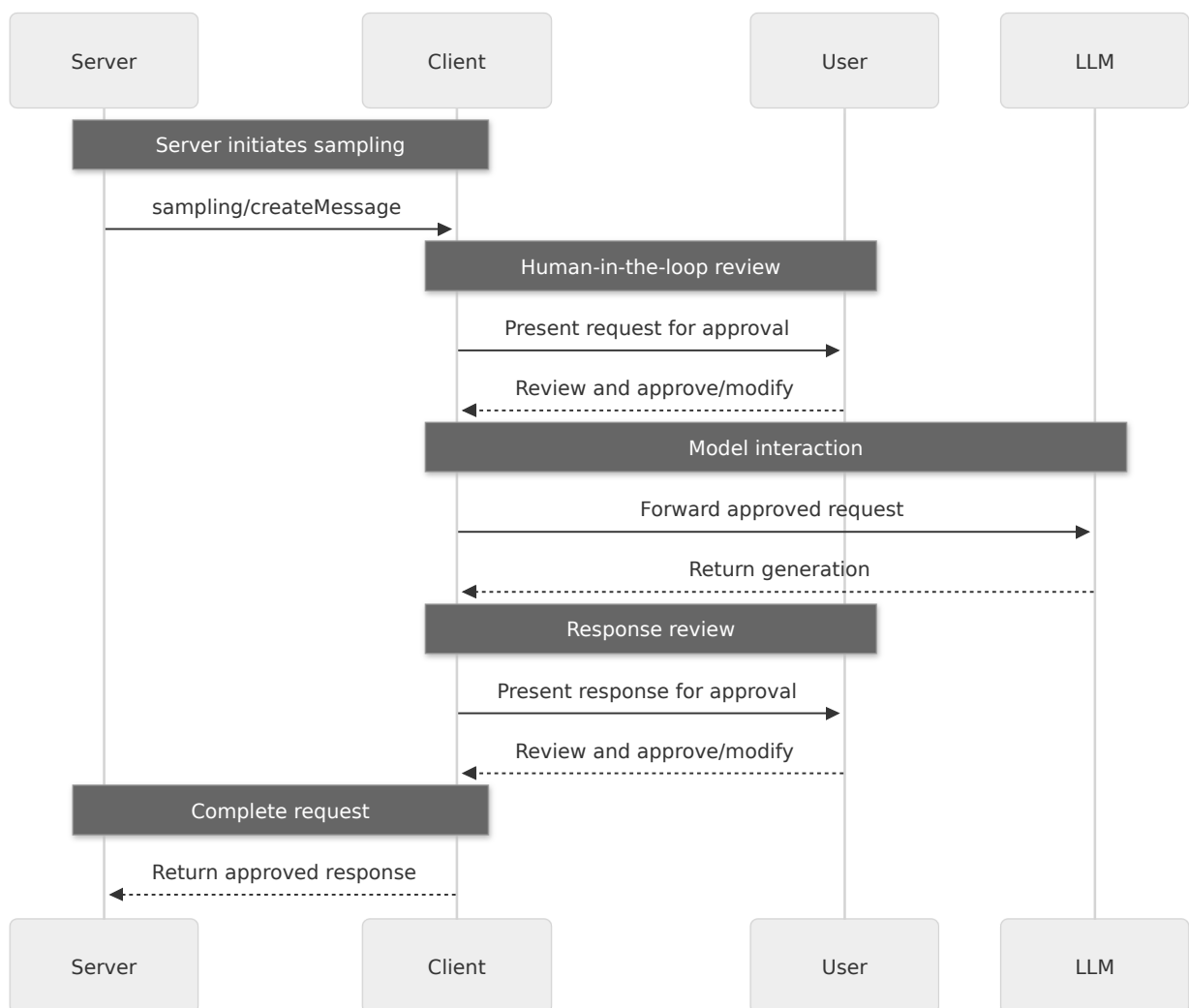
Response:

```

{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "role": "assistant",
    "content": {
      "type": "text",
      "text": "The capital of France is Paris."
    },
  },
  "model": "claude-3-sonnet-20240307",
  "stopReason": "endTurn"
}

```

Message Flow



Data Types

Messages

Sampling messages can contain:

Text Content

```
{
  "type": "text",
  "text": "The message content"
}
```

Image Content

```
{
  "type": "image",
  "data": "base64-encoded-image-data",
  "mimeType": "image/jpeg"
}
```

Audio Content

```
{
  "type": "audio",
  "data": "base64-encoded-audio-data",
  "mimeType": "audio/wav"
}
```

Model Preferences

Model selection in MCP requires careful abstraction since servers and clients may use different AI providers with distinct model offerings. A server cannot simply request a specific model by name since the client may not have access to that exact model or may prefer to use a different provider's equivalent model.

To solve this, MCP implements a preference system that combines abstract capability priorities with optional model hints:

Capability Priorities

Servers express their needs through three normalized priority values (0-1):

- `costPriority` : How important is minimizing costs? Higher values prefer cheaper models.
- `speedPriority` : How important is low latency? Higher values prefer faster models.
- `intelligencePriority` : How important are advanced capabilities? Higher values prefer more capable models.

Model Hints

While priorities help select models based on characteristics, `hints` allow servers to suggest specific models or model families:

- Hints are treated as substrings that can match model names flexibly
- Multiple hints are evaluated in order of preference
- Clients **MAY** map hints to equivalent models from different providers
- Hints are advisory—clients make final model selection

For example:

```
{
  "hints": [
    { "name": "claude-3-sonnet" }, // Prefer Sonnet-class models
    { "name": "claude" } // Fall back to any Claude model
  ],
  "costPriority": 0.3, // Cost is less important
  "speedPriority": 0.8, // Speed is very important
  "intelligencePriority": 0.5 // Moderate capability needs
}
```

The client processes these preferences to select an appropriate model from its available options. For instance, if the client doesn't have access to Claude models but has Gemini, it might map the sonnet hint to `gemini-1.5-pro` based on similar capabilities.

Error Handling

Clients **SHOULD** return errors for common failure cases:

Example error:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "error": {
    "code": -1,
    "message": "User rejected sampling request"
  }
}
```

Security Considerations

1. Clients **SHOULD** implement user approval controls
2. Both parties **SHOULD** validate message content
3. Clients **SHOULD** respect model preference hints
4. Clients **SHOULD** implement rate limiting
5. Both parties **MUST** handle sensitive data appropriately

5.3 Elicitation

NOTE

Elicitation is newly introduced in this version of the MCP specification and its design may evolve in future protocol versions.

The Model Context Protocol (MCP) provides a standardized way for servers to request additional information from users through the client during interactions. This flow allows clients to maintain control over user interactions and data sharing while enabling servers to gather necessary information dynamically. Servers request structured data from users with JSON schemas to validate responses.

User Interaction Model

Elicitation in MCP allows servers to implement interactive workflows by enabling user input requests to occur *nested* inside other MCP server features.

Implementations are free to expose elicitation through any interface pattern that suits their needs—the protocol itself does not mandate any specific user interaction model.

WARNING

For trust & safety and security:

- Servers **MUST NOT** use elicitation to request sensitive information.

Applications **SHOULD**:

- Provide UI that makes it clear which server is requesting information
- Allow users to review and modify their responses before sending
- Respect user privacy and provide clear decline and cancel options

Capabilities

Clients that support elicitation **MUST** declare the `elicitation` capability during initialization:

```
{
  "capabilities": {
    "elicitation": {}
  }
}
```

Protocol Messages

Creating Elicitation Requests

To request information from a user, servers send an `elicitation/create` request:

Simple Text Request

Request:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "elicitation/create",
  "params": {
    "message": "Please provide your GitHub username",
    "requestedSchema": {
      "type": "object",
      "properties": {
        "name": {
          "type": "string"
        }
      },
      "required": ["name"]
    }
  }
}
```

Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "action": "accept",
    "content": {
      "name": "octocat"
    }
  }
}
```

Structured Data Request

Request:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "elicitation/create",
  "params": {
    "message": "Please provide your contact information",
    "requestedSchema": {
      "type": "object",
      "properties": {
        "name": {
          "type": "string",
          "description": "Your full name"
        },
        "email": {
          "type": "string",
          "format": "email",
          "description": "Your email address"
        },
        "age": {
          "type": "number",
          "minimum": 18,
          "description": "Your age"
        }
      },
      "required": ["name", "email"]
    }
  }
}
```

Response:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "action": "accept",
    "content": {
      "name": "Monalisa Octocat",
      "email": "octocat@github.com",
      "age": 30
    }
  }
}
```

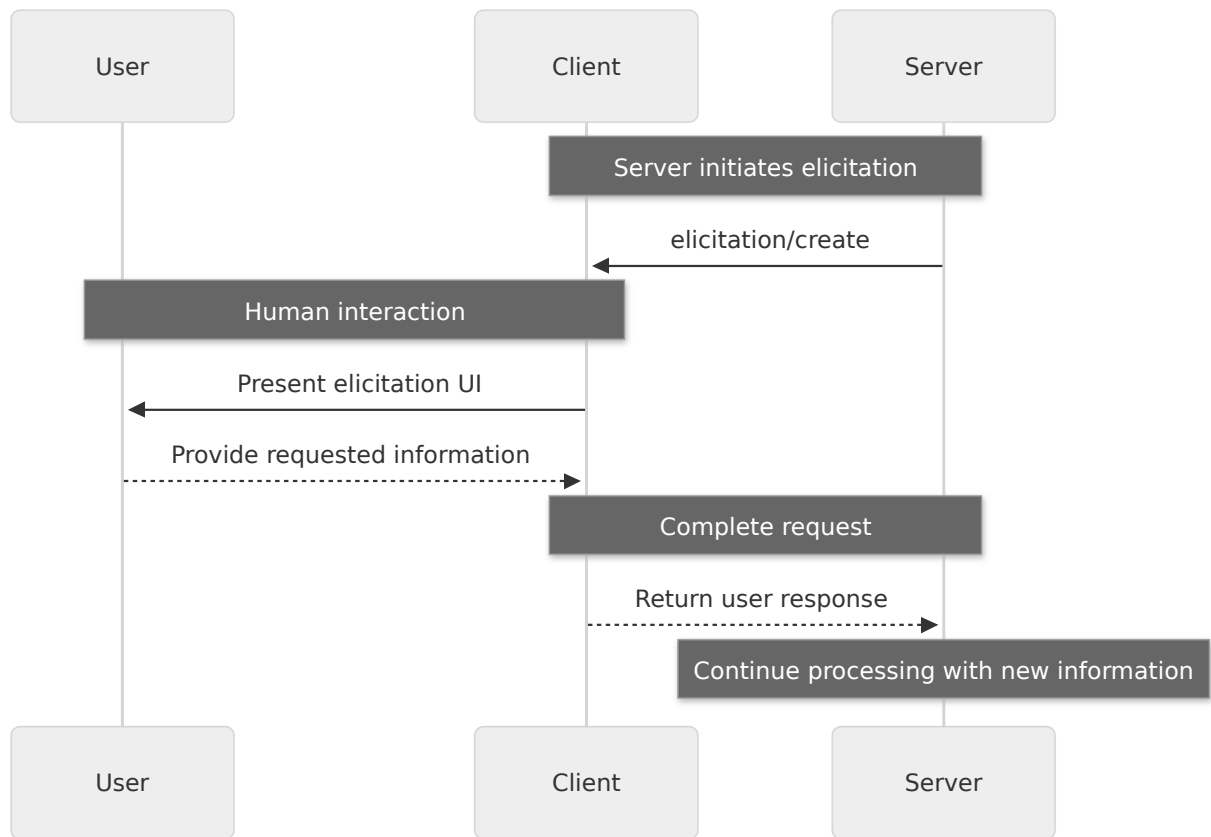
Reject Response Example:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "action": "decline"
  }
}
```

Cancel Response Example:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "action": "cancel"
  }
}
```

Message Flow



Request Schema

The `requestedSchema` field allows servers to define the structure of the expected response using a restricted subset of JSON Schema. To simplify implementation for clients, elicitation schemas are limited to flat objects with primitive properties only:

```

"requestedSchema": {
  "type": "object",
  "properties": {
    "propertyName": {
      "type": "string",
      "title": "Display Name",
      "description": "Description of the property"
    },
    "anotherProperty": {
      "type": "number",
      "minimum": 0,
      "maximum": 100
    }
  },
  "required": ["propertyName"]
}

```

Supported Schema Types

The schema is restricted to these primitive types:

1. String Schema

```

{
  "type": "string",
  "title": "Display Name",
  "description": "Description text",
  "minLength": 3,
  "maxLength": 50,
  "format": "email" // Supported: "email", "uri", "date", "date-time"
}

```

Supported formats: email, uri, date, date-time

2. Number Schema

```

{
  "type": "number", // or "integer"
  "title": "Display Name",
  "description": "Description text",
  "minimum": 0,
  "maximum": 100
}

```

3. Boolean Schema

```
{
  "type": "boolean",
  "title": "Display Name",
  "description": "Description text",
  "default": false
}
```

4. Enum Schema

```
{
  "type": "string",
  "title": "Display Name",
  "description": "Description text",
  "enum": ["option1", "option2", "option3"],
  "enumNames": ["Option 1", "Option 2", "Option 3"]
}
```

Clients can use this schema to:

1. Generate appropriate input forms
2. Validate user input before sending
3. Provide better guidance to users

Note that complex nested structures, arrays of objects, and other advanced JSON Schema features are intentionally not supported to simplify client implementation.

Response Actions

Elicitation responses use a three-action model to clearly distinguish between different user actions:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "action": "accept", // or "decline" or "cancel"
    "content": {
      "propertyName": "value",
      "anotherProperty": 42
    }
  }
}
```

The three response actions are:

1. **Accept** (action: "accept"): User explicitly approved and submitted with data

- The `content` field contains the submitted data matching the requested schema
- Example: User clicked "Submit", "OK", "Confirm", etc.

2. **Decline** (`action: "decline"`): User explicitly declined the request

- The `content` field is typically omitted
- Example: User clicked "Reject", "Decline", "No", etc.

3. **Cancel** (`action: "cancel"`): User dismissed without making an explicit choice

- The `content` field is typically omitted
- Example: User closed the dialog, clicked outside, pressed Escape, etc.

Servers should handle each state appropriately:

- **Accept:** Process the submitted data
- **Decline:** Handle explicit decline (e.g., offer alternatives)
- **Cancel:** Handle dismissal (e.g., prompt again later)

Security Considerations

1. Servers **MUST NOT** request sensitive information through elicitation
2. Clients **SHOULD** implement user approval controls
3. Both parties **SHOULD** validate elicitation content against the provided schema
4. Clients **SHOULD** provide clear indication of which server is requesting information
5. Clients **SHOULD** allow users to decline elicitation requests at any time
6. Clients **SHOULD** implement rate limiting
7. Clients **SHOULD** present elicitation requests in a way that makes it clear what information is being requested and why

6 Server Features

6.1 Overview

Servers provide the fundamental building blocks for adding context to language models via MCP. These primitives enable rich interactions between clients, servers, and language models:

- **Prompts:** Pre-defined templates or instructions that guide language model interactions
- **Resources:** Structured data or content that provides additional context to the model
- **Tools:** Executable functions that allow models to perform actions or retrieve information

Each primitive can be summarized in the following control hierarchy:

Primitive	Control	Description	Example
Prompts	User-controlled	Interactive templates invoked by user choice	Slash commands, menu options
Resources	Application-controlled	Contextual data attached and managed by the client	File contents, git history
Tools	Model-controlled	Functions exposed to the LLM to take actions	API POST requests, file writing

Explore these key primitives in more detail below:

Prompts

Resources

Tools

6.2 Prompts

The Model Context Protocol (MCP) provides a standardized way for servers to expose prompt templates to clients. Prompts allow servers to provide structured messages and instructions for interacting with language models. Clients can discover available prompts, retrieve their contents, and provide arguments to customize them.

User Interaction Model

Prompts are designed to be **user-controlled**, meaning they are exposed from servers to clients with the intention of the user being able to explicitly select them for use.

Typically, prompts would be triggered through user-initiated commands in the user interface, which allows users to naturally discover and invoke available prompts.

For example, as slash commands:



However, implementors are free to expose prompts through any interface pattern that suits their needs—the protocol itself does not mandate any specific user interaction model.

Capabilities

Servers that support prompts **MUST** declare the `prompts` capability during initialization:

```
{
  "capabilities": {
    "prompts": {
      "listChanged": true
    }
  }
}
```

`listChanged` indicates whether the server will emit notifications when the list of available prompts changes.

Protocol Messages

Listing Prompts

To retrieve available prompts, clients send a `prompts/list` request. This operation supports pagination.

Request:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "prompts/list",
  "params": {
    "cursor": "optional-cursor-value"
  }
}
```

Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "prompts": [
      {
        "name": "code_review",
        "title": "Request Code Review",
        "description": "Asks the LLM to analyze code quality and suggest improvements",
        "arguments": [
          {
            "name": "code",
            "description": "The code to review",
            "required": true
          }
        ]
      }
    ]
  },
  "nextCursor": "next-page-cursor"
}
```

Getting a Prompt

To retrieve a specific prompt, clients send a `prompts/get` request. Arguments may be auto-completed through the completion API.

Request:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "prompts/get",
  "params": {
    "name": "code_review",
    "arguments": {
      "code": "def hello():\n    print('world')"
```

Response:

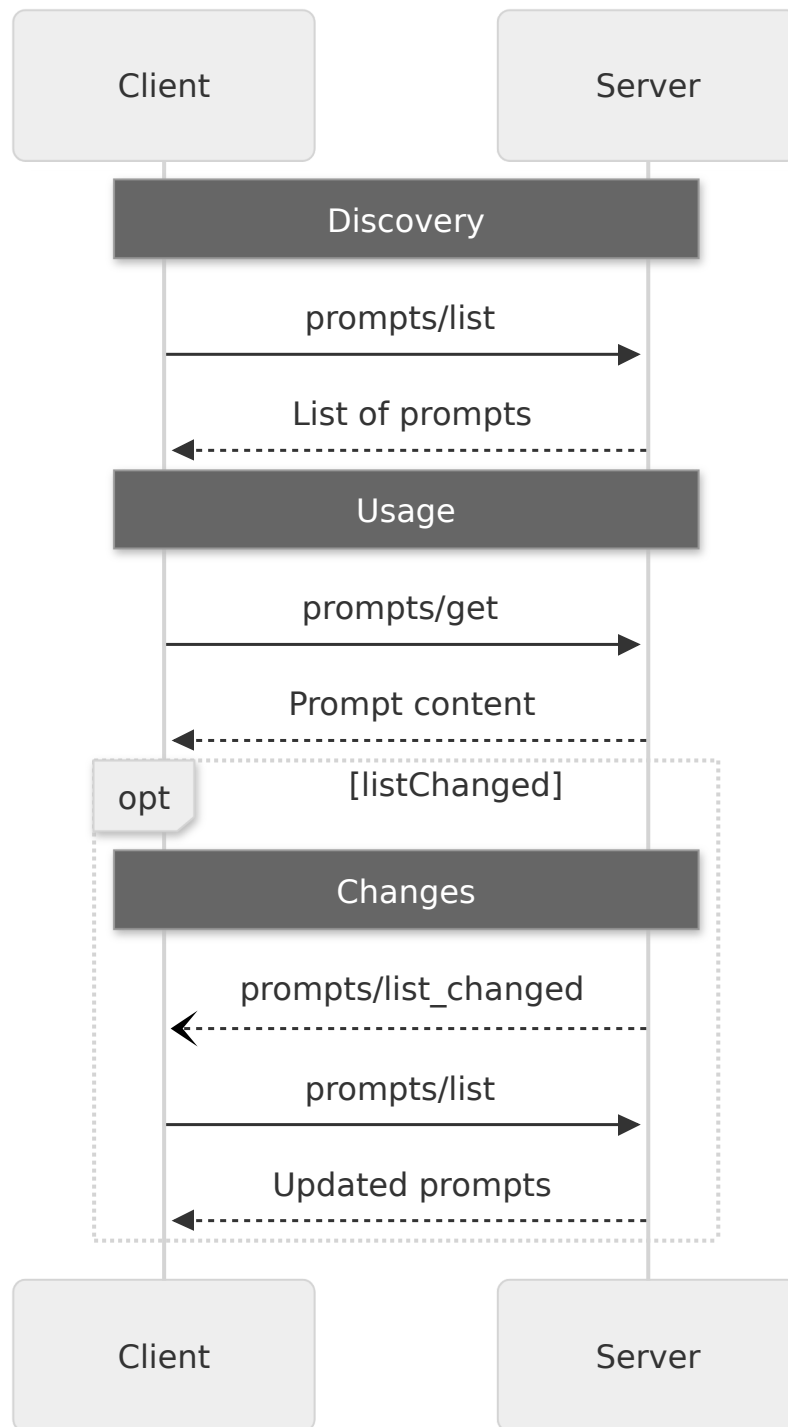
```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "description": "Code review prompt",
    "messages": [
      {
        "role": "user",
        "content": {
          "type": "text",
          "text": "Please review this Python code:\ndef hello():\n    print('world')"
```

List Changed Notification

When the list of available prompts changes, servers that declared the `listChanged` capability **SHOULD** send a notification:

```
{
  "jsonrpc": "2.0",
  "method": "notifications/prompts/list_changed"
}
```

Message Flow



Data Types

Prompt

A prompt definition includes:

- `name` : Unique identifier for the prompt
- `title` : Optional human-readable name of the prompt for display purposes.
- `description` : Optional human-readable description
- `arguments` : Optional list of arguments for customization

PromptMessage

Messages in a prompt can contain:

- `role` : Either "user" or "assistant" to indicate the speaker
- `content` : One of the following content types:

NOTE

All content types in prompt messages support optional annotations for metadata about audience, priority, and modification times.

Text Content

Text content represents plain text messages:

```
{
  "type": "text",
  "text": "The text content of the message"
}
```

This is the most common content type used for natural language interactions.

Image Content

Image content allows including visual information in messages:

```
{
  "type": "image",
  "data": "base64-encoded-image-data",
  "mimeType": "image/png"
}
```

The image data **MUST** be base64-encoded and include a valid MIME type. This enables multi-modal interactions where visual context is important.

Audio Content

Audio content allows including audio information in messages:

```
{
  "type": "audio",
  "data": "base64-encoded-audio-data",
  "mimeType": "audio/wav"
}
```

The audio data **MUST** be base64-encoded and include a valid MIME type. This enables multi-modal interactions where audio context is important.

Embedded Resources

Embedded resources allow referencing server-side resources directly in messages:

```
{
  "type": "resource",
  "resource": {
    "uri": "resource://example",
    "mimeType": "text/plain",
    "text": "Resource content"
  }
}
```

Resources can contain either text or binary (blob) data and **MUST** include:

- A valid resource URI
- The appropriate MIME type
- Either text content or base64-encoded blob data

Embedded resources enable prompts to seamlessly incorporate server-managed content like documentation, code samples, or other reference materials directly into the conversation flow.

Error Handling

Servers **SHOULD** return standard JSON-RPC errors for common failure cases:

- Invalid prompt name: `-32602` (Invalid params)
- Missing required arguments: `-32602` (Invalid params)
- Internal errors: `-32603` (Internal error)

Implementation Considerations

1. Servers **SHOULD** validate prompt arguments before processing
2. Clients **SHOULD** handle pagination for large prompt lists

3. Both parties **SHOULD** respect capability negotiation

Security

Implementations **MUST** carefully validate all prompt inputs and outputs to prevent injection attacks or unauthorized access to resources.

6.3 Resources

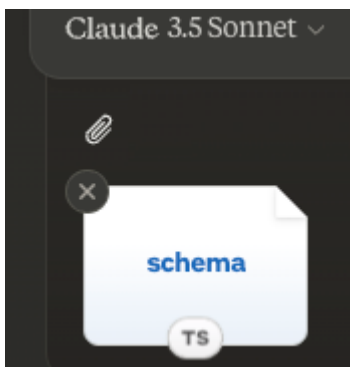
The Model Context Protocol (MCP) provides a standardized way for servers to expose resources to clients. Resources allow servers to share data that provides context to language models, such as files, database schemas, or application-specific information. Each resource is uniquely identified by a URI.

User Interaction Model

Resources in MCP are designed to be **application-driven**, with host applications determining how to incorporate context based on their needs.

For example, applications could:

- Expose resources through UI elements for explicit selection, in a tree or list view
- Allow the user to search through and filter available resources
- Implement automatic context inclusion, based on heuristics or the AI model's selection



However, implementations are free to expose resources through any interface pattern that suits their needs—the protocol itself does not mandate any specific user interaction model.

Capabilities

Servers that support resources **MUST** declare the `resources` capability:

```
{
  "capabilities": {
    "resources": {
      "subscribe": true,
      "listChanged": true
    }
  }
}
```

The capability supports two optional features:

- `subscribe` : whether the client can subscribe to be notified of changes to individual resources.
- `listChanged` : whether the server will emit notifications when the list of available resources changes.

Both `subscribe` and `listChanged` are optional—servers can support neither, either, or both:

```
{
  "capabilities": {
    "resources": {} // Neither feature supported
  }
}
```

```
{
  "capabilities": {
    "resources": {
      "subscribe": true // Only subscriptions supported
    }
  }
}
```

```
{
  "capabilities": {
    "resources": {
      "listChanged": true // Only list change notifications supported
    }
  }
}
```

Protocol Messages

Listing Resources

To discover available resources, clients send a `resources/list` request. This operation supports pagination.

Request:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "resources/list",
  "params": {
    "cursor": "optional-cursor-value"
  }
}
```

Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "resources": [
      {
        "uri": "file:///project/src/main.rs",
        "name": "main.rs",
        "title": "Rust Software Application Main File",
        "description": "Primary application entry point",
        "mimeType": "text/x-rust"
      }
    ],
    "nextCursor": "next-page-cursor"
  }
}
```

Reading Resources

To retrieve resource contents, clients send a `resources/read` request:

Request:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "resources/read",
  "params": {
    "uri": "file:///project/src/main.rs"
  }
}
```

Response:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "contents": [
      {
        "uri": "file:///project/src/main.rs",
        "mimeType": "text/x-rust",
        "text": "fn main() {\n    println!(\"Hello world!\");\n}"
      }
    ]
  }
}
```

Resource Templates

Resource templates allow servers to expose parameterized resources using [URI templates](#). Arguments may be auto-completed through the completion API. This operation supports pagination.

Request:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "resources/templates/list",
  "params": {
    "cursor": "optional-cursor-value"
  }
}
```

Response:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "result": {
    "resourceTemplates": [
      {
        "uriTemplate": "file:///path",
        "name": "Project Files",
        "title": "📁 Project Files",
        "description": "Access files in the project directory",
        "mimeType": "application/octet-stream"
      }
    ],
    "nextCursor": "next-page-cursor"
  }
}
```

List Changed Notification

When the list of available resources changes, servers that declared the `listChanged` capability **SHOULD** send a notification:

```
{
  "jsonrpc": "2.0",
  "method": "notifications/resources/list_changed"
}
```

Subscriptions

The protocol supports optional subscriptions to resource changes. Clients can subscribe to specific resources and receive notifications when they change:

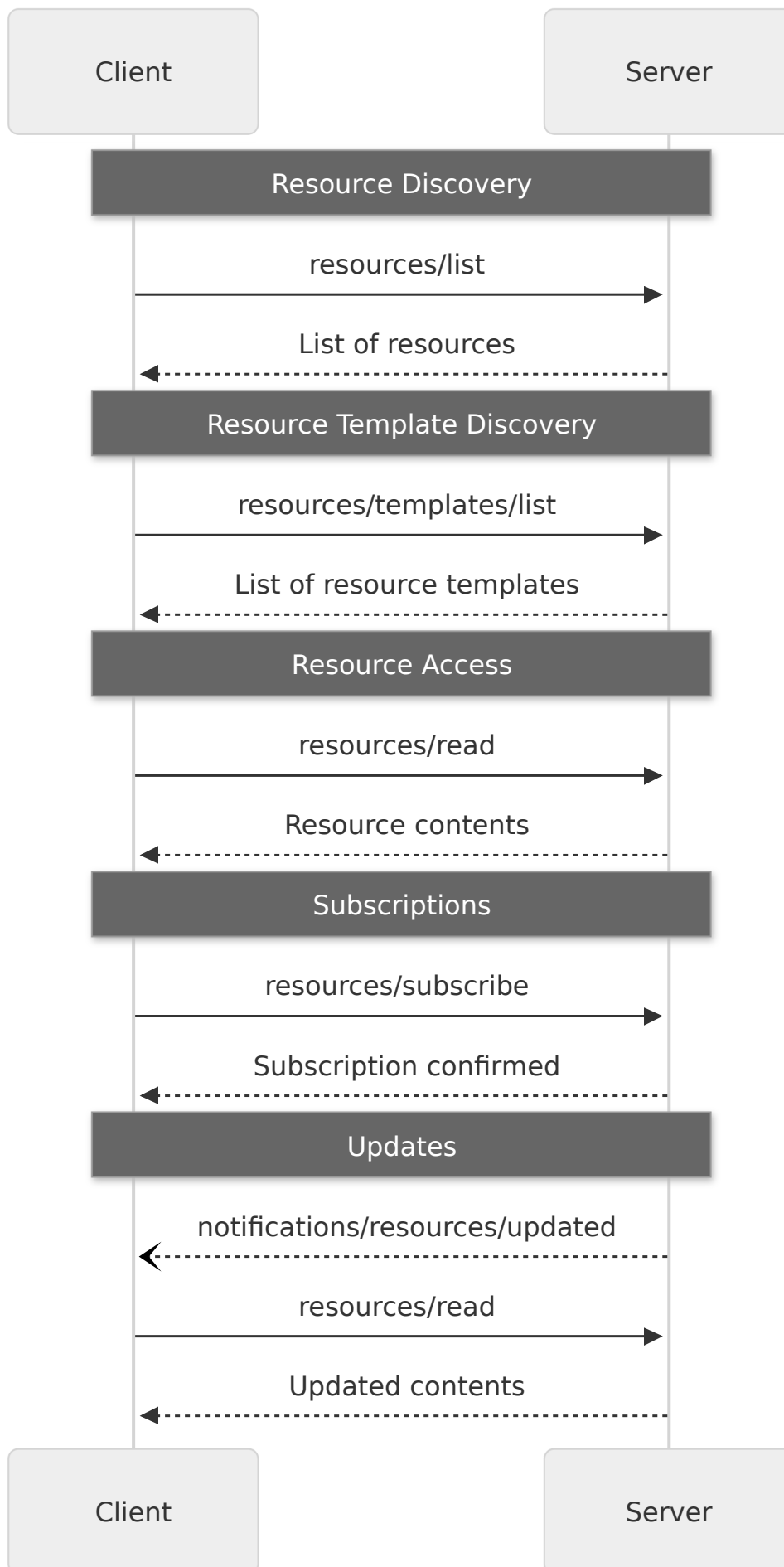
Subscribe Request:

```
{
  "jsonrpc": "2.0",
  "id": 4,
  "method": "resources/subscribe",
  "params": {
    "uri": "file:///project/src/main.rs"
  }
}
```

Update Notification:

```
{  
  "jsonrpc": "2.0",  
  "method": "notifications/resources/updated",  
  "params": {  
    "uri": "file:///project/src/main.rs"  
  }  
}
```

Message Flow



Data Types

Resource

A resource definition includes:

- **uri** : Unique identifier for the resource
- **name** : The name of the resource.
- **title** : Optional human-readable name of the resource for display purposes.
- **description** : Optional description
- **mimeType** : Optional MIME type
- **size** : Optional size in bytes

Resource Contents

Resources can contain either text or binary data:

Text Content

```
{
  "uri": "file:///example.txt",
  "mimeType": "text/plain",
  "text": "Resource content"
}
```

Binary Content

```
{
  "uri": "file:///example.png",
  "mimeType": "image/png",
  "blob": "base64-encoded-data"
}
```

Annotations

Resources, resource templates and content blocks support optional annotations that provide hints to clients about how to use or display the resource:

- **audience** : An array indicating the intended audience(s) for this resource. Valid values are **"user"** and **"assistant"**. For example, **["user", "assistant"]** indicates content useful for both.
- **priority** : A number from 0.0 to 1.0 indicating the importance of this resource. A value of 1 means "most important" (effectively required), while 0 means "least important" (entirely optional).

- **lastModified**: An ISO 8601 formatted timestamp indicating when the resource was last modified (e.g., "2025-01-12T15:00:58Z").

Example resource with annotations:

```
{
  "uri": "file:///project/README.md",
  "name": "README.md",
  "title": "Project Documentation",
  "mimeType": "text/markdown",
  "annotations": {
    "audience": ["user"],
    "priority": 0.8,
    "lastModified": "2025-01-12T15:00:58Z"
  }
}
```

Clients can use these annotations to:

- Filter resources based on their intended audience
- Prioritize which resources to include in context
- Display modification times or sort by recency

Common URI Schemes

The protocol defines several standard URI schemes. This list not exhaustive—implementations are always free to use additional, custom URI schemes.

https://

Used to represent a resource available on the web.

Servers **SHOULD** use this scheme only when the client is able to fetch and load the resource directly from the web on its own—that is, it doesn't need to read the resource via the MCP server.

For other use cases, servers **SHOULD** prefer to use another URI scheme, or define a custom one, even if the server will itself be downloading resource contents over the internet.

file://

Used to identify resources that behave like a filesystem. However, the resources do not need to map to an actual physical filesystem.

MCP servers **MAY** identify file:// resources with an XDG MIME type, like `inode/directory`, to represent non-regular files (such as directories) that don't otherwise have a standard MIME type.

git://

Git version control integration.

Custom URI Schemes

Custom URI schemes **MUST** be in accordance with [RFC3986](#), taking the above guidance in to account.

Error Handling

Servers **SHOULD** return standard JSON-RPC errors for common failure cases:

- Resource not found: -32002
- Internal errors: -32603

Example error:

```
{
  "jsonrpc": "2.0",
  "id": 5,
  "error": {
    "code": -32002,
    "message": "Resource not found",
    "data": {
      "uri": "file:///nonexistent.txt"
    }
  }
}
```

Security Considerations

1. Servers **MUST** validate all resource URIs
2. Access controls **SHOULD** be implemented for sensitive resources
3. Binary data **MUST** be properly encoded
4. Resource permissions **SHOULD** be checked before operations

6.4 Tools

The Model Context Protocol (MCP) allows servers to expose tools that can be invoked by language models. Tools enable models to interact with external systems, such as querying databases, calling APIs, or performing computations. Each tool is uniquely identified by a name and includes metadata describing its schema.

User Interaction Model

Tools in MCP are designed to be **model-controlled**, meaning that the language model can discover and invoke tools automatically based on its contextual understanding and the user's prompts.

However, implementations are free to expose tools through any interface pattern that suits their needs—the protocol itself does not mandate any specific user interaction model.

WARNING

For trust & safety and security, there **SHOULD** always be a human in the loop with the ability to deny tool invocations.

Applications **SHOULD**:

- Provide UI that makes clear which tools are being exposed to the AI model
- Insert clear visual indicators when tools are invoked
- Present confirmation prompts to the user for operations, to ensure a human is in the loop

Capabilities

Servers that support tools **MUST** declare the `tools` capability:

```
{
  "capabilities": {
    "tools": {
      "listChanged": true
    }
  }
}
```

`listChanged` indicates whether the server will emit notifications when the list of available tools changes.

Protocol Messages

Listing Tools

To discover available tools, clients send a `tools/list` request. This operation supports pagination.

Request:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "tools/list",
  "params": {
    "cursor": "optional-cursor-value"
  }
}
```

Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "tools": [
      {
        "name": "get_weather",
        "title": "Weather Information Provider",
        "description": "Get current weather information for a location",
        "inputSchema": {
          "type": "object",
          "properties": {
            "location": {
              "type": "string",
              "description": "City name or zip code"
            }
          },
          "required": ["location"]
        }
      }
    ],
    "nextCursor": "next-page-cursor"
  }
}
```

Calling Tools

To invoke a tool, clients send a `tools/call` request:

Request:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "tools/call",
  "params": {
    "name": "get_weather",
    "arguments": {
      "location": "New York"
    }
  }
}
```

Response:

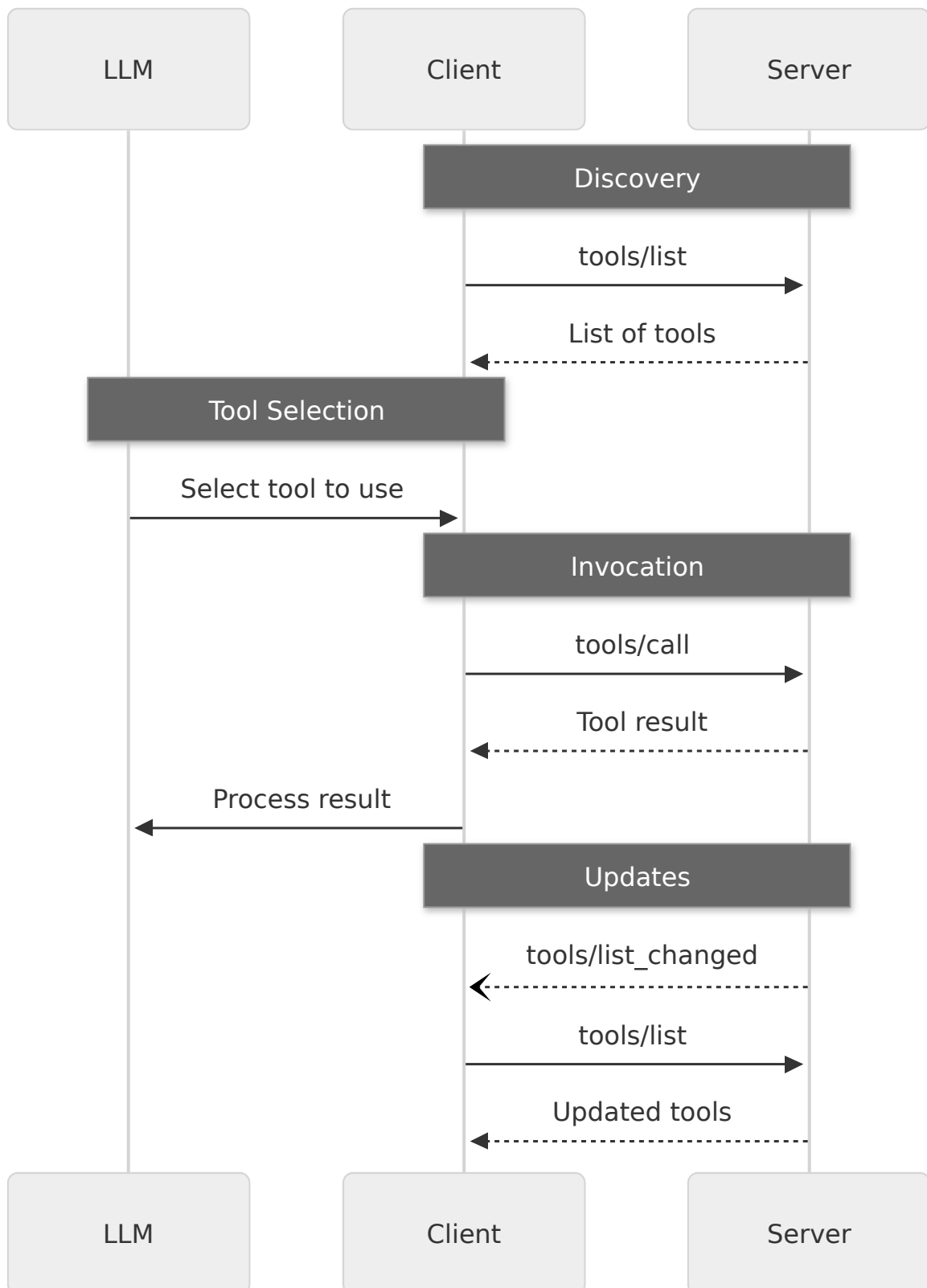
```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "content": [
      {
        "type": "text",
        "text": "Current weather in New York:\nTemperature: 72°F\nConditions: Partly cloudy"
      }
    ],
    "isError": false
  }
}
```

List Changed Notification

When the list of available tools changes, servers that declared the `listChanged` capability **SHOULD** send a notification:

```
{
  "jsonrpc": "2.0",
  "method": "notifications/tools/list_changed"
}
```

Message Flow



Data Types

Tool

A tool definition includes:

- `name` : Unique identifier for the tool
- `title` : Optional human-readable name of the tool for display purposes.
- `description` : Human-readable description of functionality
- `inputSchema` : JSON Schema defining expected parameters
- `outputSchema` : Optional JSON Schema defining expected output structure
- `annotations` : optional properties describing tool behavior

WARNING

For trust & safety and security, clients **MUST** consider tool annotations to be untrusted unless they come from trusted servers.

Tool Result

Tool results may contain **structured** or **unstructured** content.

Unstructured content is returned in the `content` field of a result, and can contain multiple content items of different types:

NOTE

All content types (text, image, audio, resource links, and embedded resources) support optional annotations that provide metadata about audience, priority, and modification times. This is the same annotation format used by resources and prompts.

Text Content

```
{
  "type": "text",
  "text": "Tool result text"
}
```

Image Content

```
{
  "type": "image",
  "data": "base64-encoded-data",
  "mimeType": "image/png",
  "annotations": {
    "audience": ["user"],
    "priority": 0.9
  }
}
```

This example demonstrates the use of an optional Annotation.

Audio Content

```
{
  "type": "audio",
  "data": "base64-encoded-audio-data",
  "mimeType": "audio/wav"
}
```

Resource Links

A tool **MAY** return links to Resources, to provide additional context or data. In this case, the tool will return a URI that can be subscribed to or fetched by the client:

```
{
  "type": "resource_link",
  "uri": "file:///project/src/main.rs",
  "name": "main.rs",
  "description": "Primary application entry point",
  "mimeType": "text/x-rust",
  "annotations": {
    "audience": ["assistant"],
    "priority": 0.9
  }
}
```

Resource links support the same Resource annotations as regular resources to help clients understand how to use them.

INFO

Resource links returned by tools are not guaranteed to appear in the results of a `resources/list` request.

Embedded Resources

Resources **MAY** be embedded to provide additional context or data using a suitable URI scheme. Servers that use embedded resources **SHOULD** implement the `resources` capability:

```
{
  "type": "resource",
  "resource": {
    "uri": "file:///project/src/main.rs",
    "mimeType": "text/x-rust",
    "text": "fn main() {\n    println!(\"Hello world!\");\n}",
    "annotations": {
      "audience": ["user", "assistant"],
      "priority": 0.7,
      "lastModified": "2025-05-03T14:30:00Z"
    }
  }
}
```

Embedded resources support the same Resource annotations as regular resources to help clients understand how to use them.

Structured Content

Structured content is returned as a JSON object in the `structuredContent` field of a result.

For backwards compatibility, a tool that returns structured content **SHOULD** also return the serialized JSON in a `TextContent` block.

NOTE

`structuredContent` is server-produced result data and is unrelated to LLM "structured outputs" (schema-constrained model generation).

Output Schema

Tools may also provide an output schema for validation of structured results. If an output schema is provided:

- Servers **MUST** provide structured results that conform to this schema.
- Clients **SHOULD** validate structured results against this schema.

Example tool with output schema:

```
{
  "name": "get_weather_data",
  "title": "Weather Data Retriever",
  "description": "Get current weather data for a location",
  "inputSchema": {
    "type": "object",
    "properties": {
      "location": {
        "type": "string",
        "description": "City name or zip code"
      }
    },
    "required": ["location"]
  },
  "outputSchema": {
    "type": "object",
    "properties": {
      "temperature": {
        "type": "number",
        "description": "Temperature in celsius"
      },
      "conditions": {
        "type": "string",
        "description": "Weather conditions description"
      },
      "humidity": {
        "type": "number",
        "description": "Humidity percentage"
      }
    },
    "required": ["temperature", "conditions", "humidity"]
  }
}
```

Example valid response for this tool:

```

{
  "jsonrpc": "2.0",
  "id": 5,
  "result": {
    "content": [
      {
        "type": "text",
        "text": "{\"temperature\": 22.5, \"conditions\": \"Partly cloudy\", \"humidity\": 65}"
      }
    ],
    "structuredContent": {
      "temperature": 22.5,
      "conditions": "Partly cloudy",
      "humidity": 65
    }
  }
}

```

Providing an output schema helps clients and LLMs understand and properly handle structured tool outputs by:

- Enabling strict schema validation of responses
- Providing type information for better integration with programming languages
- Guiding clients and LLMs to properly parse and utilize the returned data
- Supporting better documentation and developer experience

Error Handling

Tools use two error reporting mechanisms:

1. **Protocol Errors:** Standard JSON-RPC errors for issues like:

- Unknown tools
- Invalid arguments
- Server errors

2. **Tool Execution Errors:** Reported in tool results with `isError: true`:

- API failures
- Invalid input data
- Business logic errors

Example protocol error:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "error": {
    "code": -32602,
    "message": "Unknown tool: invalid_tool_name"
  }
}
```

Example tool execution error:

```
{
  "jsonrpc": "2.0",
  "id": 4,
  "result": {
    "content": [
      {
        "type": "text",
        "text": "Failed to fetch weather data: API rate limit exceeded"
      }
    ],
    "isError": true
  }
}
```

Security Considerations

1. Servers **MUST**:

- Validate all tool inputs
- Implement proper access controls
- Rate limit tool invocations
- Sanitize tool outputs

2. Clients **SHOULD**:

- Prompt for user confirmation on sensitive operations
- Show tool inputs to the user before calling the server, to avoid malicious or accidental data exfiltration
- Validate tool results before passing to LLM
- Implement timeouts for tool calls
- Log tool usage for audit purposes

6.5 Utilities

6.5.1 Completion

The Model Context Protocol (MCP) provides a standardized way for servers to offer argument autocompletion suggestions for prompts and resource URIs. This enables rich, IDE-like experiences where users receive contextual suggestions while entering argument values.

User Interaction Model

Completion in MCP is designed to support interactive user experiences similar to IDE code completion.

For example, applications may show completion suggestions in a dropdown or popup menu as users type, with the ability to filter and select from available options.

However, implementations are free to expose completion through any interface pattern that suits their needs—the protocol itself does not mandate any specific user interaction model.

Capabilities

Servers that support completions **MUST** declare the `completions` capability:

```
{
  "capabilities": {
    "completions": {}
  }
}
```

Protocol Messages

Requesting Completions

To get completion suggestions, clients send a `completion/complete` request specifying what is being completed through a reference type:

Request:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "completion/complete",
  "params": {
    "ref": {
      "type": "ref/prompt",
      "name": "code_review"
    },
    "argument": {
      "name": "language",
      "value": "py"
    }
  }
}
```

Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "completion": {
      "values": ["python", "pytorch", "pyside"],
      "total": 10,
      "hasMore": true
    }
  }
}
```

For prompts or URI templates with multiple arguments, clients should include previous completions in the `context.arguments` object to provide context for subsequent requests.

Request:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "completion/complete",
  "params": {
    "ref": {
      "type": "ref/prompt",
      "name": "code_review"
    },
    "argument": {
      "name": "framework",
      "value": "fla"
    },
    "context": {
      "arguments": {
        "language": "python"
      }
    }
  }
}
```

Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "completion": {
      "values": ["flask"],
      "total": 1,
      "hasMore": false
    }
  }
}
```

Reference Types

The protocol supports two types of completion references:

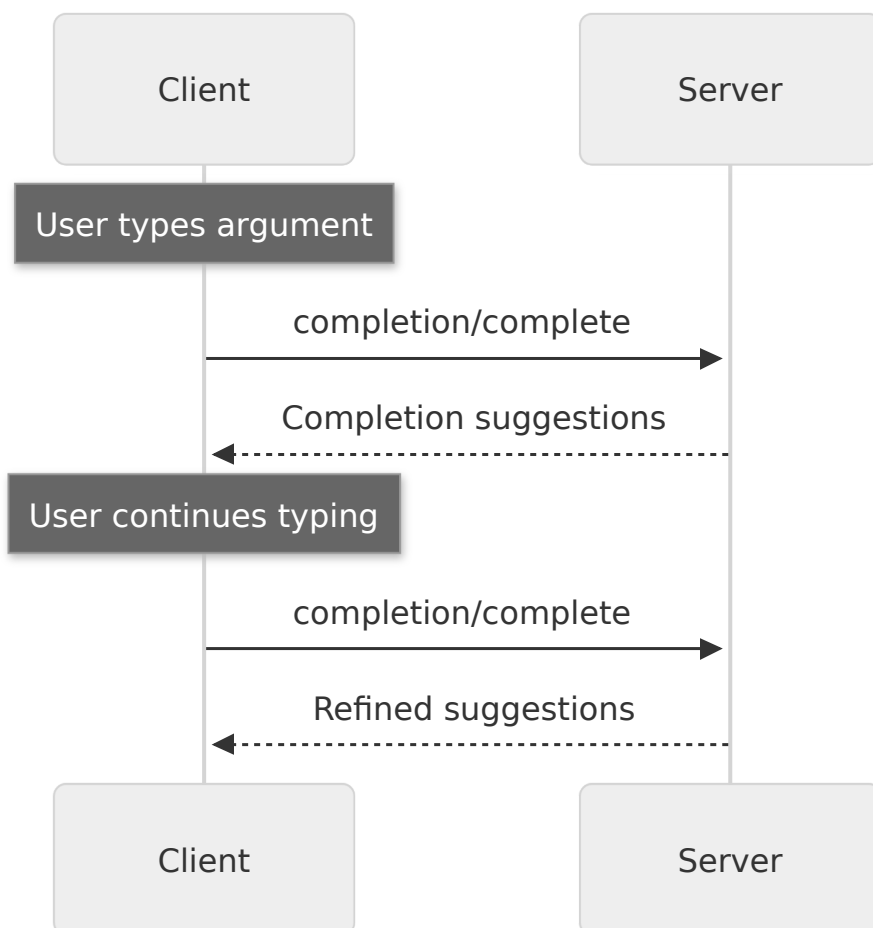
Type	Description	Example
<code>ref/prompt</code>	References a prompt by name	<code>{"type": "ref/prompt", "name": "code_review"}</code>
<code>ref/resource</code>	References a resource URI	<code>{"type": "ref/resource", "uri": "file:///path"}</code>

Completion Results

Servers return an array of completion values ranked by relevance, with:

- Maximum 100 items per response
- Optional total number of available matches
- Boolean indicating if additional results exist

Message Flow



Data Types

CompleteRequest

- `ref` : A `PromptReference` or `ResourceReference`
- `argument` : Object containing:
 - `name` : Argument name
 - `value` : Current value
- `context` : Object containing:
 - `arguments` : A mapping of already-resolved argument names to their values.

CompleteResult

- `completion` : Object containing:
 - `values` : Array of suggestions (max 100)
 - `total` : Optional total matches
 - `hasMore` : Additional results flag

Error Handling

Servers **SHOULD** return standard JSON-RPC errors for common failure cases:

- Method not found: `-32601` (Capability not supported)
- Invalid prompt name: `-32602` (Invalid params)
- Missing required arguments: `-32602` (Invalid params)
- Internal errors: `-32603` (Internal error)

Implementation Considerations

1. Servers **SHOULD**:

- Return suggestions sorted by relevance
- Implement fuzzy matching where appropriate
- Rate limit completion requests
- Validate all inputs

2. Clients **SHOULD**:

- Debounce rapid completion requests
- Cache completion results where appropriate
- Handle missing or partial results gracefully

Security

Implementations **MUST**:

- Validate all completion inputs
- Implement appropriate rate limiting
- Control access to sensitive suggestions

- Prevent completion-based information disclosure

6.5.2 Logging

The Model Context Protocol (MCP) provides a standardized way for servers to send structured log messages to clients. Clients can control logging verbosity by setting minimum log levels, with servers sending notifications containing severity levels, optional logger names, and arbitrary JSON-serializable data.

User Interaction Model

Implementations are free to expose logging through any interface pattern that suits their needs—the protocol itself does not mandate any specific user interaction model.

Capabilities

Servers that emit log message notifications **MUST** declare the `logging` capability:

```
{
  "capabilities": {
    "logging": {}
  }
}
```

Log Levels

The protocol follows the standard syslog severity levels specified in [RFC 5424](#):

Level	Description	Example Use Case
debug	Detailed debugging information	Function entry/exit points
info	General informational messages	Operation progress updates
notice	Normal but significant events	Configuration changes
warning	Warning conditions	Deprecated feature usage
error	Error conditions	Operation failures
critical	Critical conditions	System component failures
alert	Action must be taken immediately	Data corruption detected
emergency	System is unusable	Complete system failure

Protocol Messages

Setting Log Level

To configure the minimum log level, clients **MAY** send a `logging/setLevel` request:

Request:

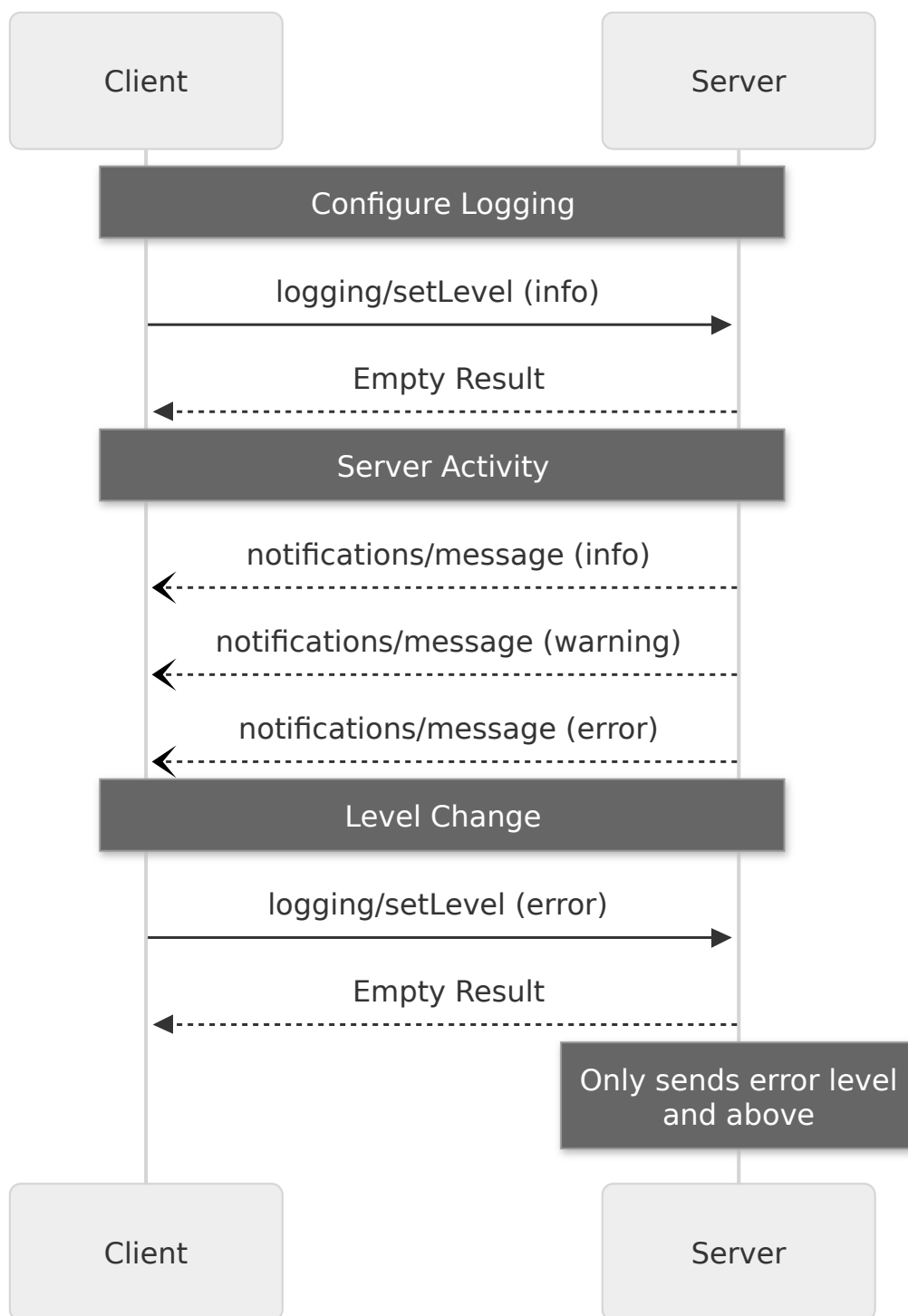
```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "logging/setLevel",
  "params": {
    "level": "info"
  }
}
```

Log Message Notifications

Servers send log messages using `notifications/message` notifications:

```
{
  "jsonrpc": "2.0",
  "method": "notifications/message",
  "params": {
    "level": "error",
    "logger": "database",
    "data": {
      "error": "Connection failed",
      "details": {
        "host": "localhost",
        "port": 5432
      }
    }
  }
}
```

Message Flow



Error Handling

Servers **SHOULD** return standard JSON-RPC errors for common failure cases:

- Invalid log level: `-32602` (Invalid params)
- Configuration errors: `-32603` (Internal error)

Implementation Considerations

1. Servers **SHOULD**:

- Rate limit log messages
- Include relevant context in data field
- Use consistent logger names
- Remove sensitive information

2. Clients **MAY**:

- Present log messages in the UI
- Implement log filtering/search
- Display severity visually
- Persist log messages

Security

1. Log messages **MUST NOT** contain:

- Credentials or secrets
- Personal identifying information
- Internal system details that could aid attacks

2. Implementations **SHOULD**:

- Rate limit messages
- Validate all data fields
- Control log access
- Monitor for sensitive content

6.5.3 Pagination

The Model Context Protocol (MCP) supports paginating list operations that may return large result sets. Pagination allows servers to yield results in smaller chunks rather than all at once.

Pagination is especially important when connecting to external services over the internet, but also useful for local integrations to avoid performance issues with large data sets.

Pagination Model

Pagination in MCP uses an opaque cursor-based approach, instead of numbered pages.

- The **cursor** is an opaque string token, representing a position in the result set
- **Page size** is determined by the server, and clients **MUST NOT** assume a fixed page size

Response Format

Pagination starts when the server sends a **response** that includes:

- The current page of results
- An optional `nextCursor` field if more results exist

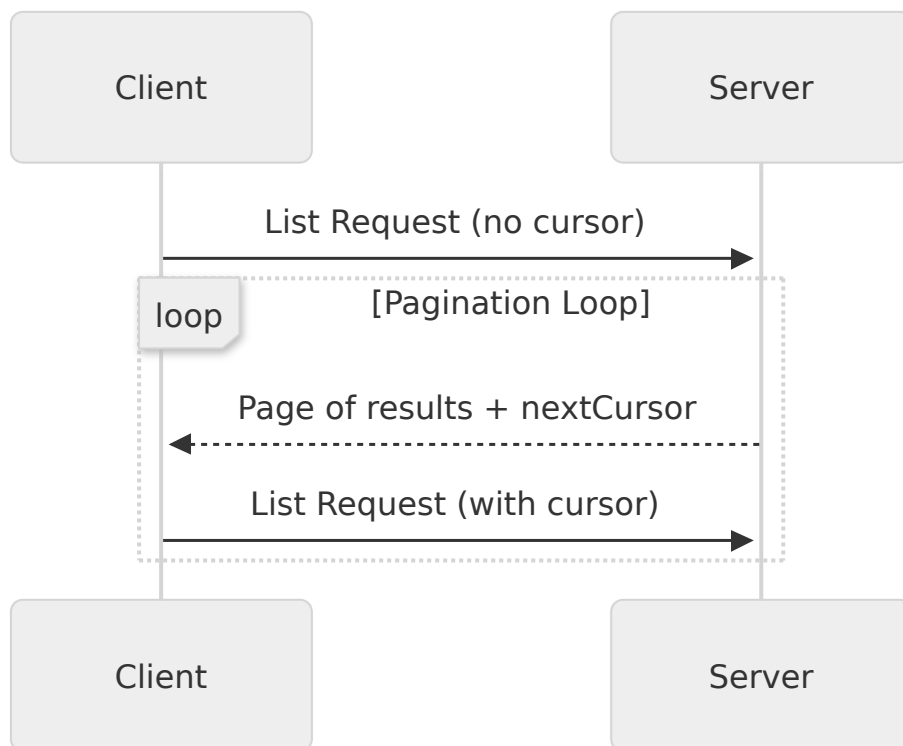
```
{
  "jsonrpc": "2.0",
  "id": "123",
  "result": {
    "resources": [...],
    "nextCursor": "eyJwYWdlIjogM30="
  }
}
```

Request Format

After receiving a cursor, the client can *continue* paginating by issuing a request including that cursor:

```
{
  "jsonrpc": "2.0",
  "id": "124",
  "method": "resources/list",
  "params": {
    "cursor": "eyJwYWdlIjogMn0="
  }
}
```

Pagination Flow



Operations Supporting Pagination

The following MCP operations support pagination:

- `resources/list` - List available resources
- `resources/templates/list` - List resource templates
- `prompts/list` - List available prompts
- `tools/list` - List available tools

Implementation Guidelines

1. Servers **SHOULD**:

- Provide stable cursors
- Handle invalid cursors gracefully

2. Clients **SHOULD**:

- Treat a missing `nextCursor` as the end of results
- Support both paginated and non-paginated flows

3. Clients **MUST** treat cursors as opaque tokens:

- Don't make assumptions about cursor format
- Don't attempt to parse or modify cursors
- Don't persist cursors across sessions

Error Handling

Invalid cursors **SHOULD** result in an error with code -32602 (Invalid params).

7 Schema Reference

JSON-RPC

JSONRPCError

```
interface JSONRPCError {
  jsonrpc: "2.0";
  id: RequestId;
  error: { code: number; message: string; data?: unknown };
}
```

A response to a request that indicates an error occurred.

jsonrpc: "2.0"

id: RequestId

error: { code: number; message: string; data?: unknown }

Type Declaration

- code: number

The error type that occurred.

- message: string

A short description of the error. The message SHOULD be limited to a concise single sentence.

- Optional data?: unknown

Additional information about the error. The value of this member is defined by the sender (e.g. detailed error information, nested errors etc.).

JSONRPCMessage

```
JSONRPCMessage:
| JSONRPCRequest
| JSONRPCNotification
| JSONRPCResponse
| JSONRPCError
```

Refers to any valid JSON-RPC object that can be decoded off the wire, or encoded to be sent.

JSONRPCNotification

```
interface JSONRPCNotification {
  method: string;
  params?: { _meta?: { [key: string]: unknown }; [key: string]: unknown };
  jsonrpc: "2.0";
}
```

A notification which does not expect a response.

method: string

Inherited from Notification.method

params?: { _meta?: { [key: string]: unknown }; [key: string]: unknown }

Type Declaration

- [key: string]: unknown
- Optional _meta?: { [key: string]: unknown }

See General fields: `_meta` for notes on `_meta` usage.

Inherited from Notification.params

jsonrpc: "2.0"

JSONRPCRequest

```
interface JSONRPCRequest {
  method: string;
  params?: {
    _meta?: { progressToken?: ProgressToken; [key: string]: unknown };
    [key: string]: unknown;
  };
  jsonrpc: "2.0";
  id: RequestId;
}
```

A request that expects a response.

method: string

Inherited from Request.method

params?: { _meta?: { progressToken?: ProgressToken; [key: string]: unknown }; [key: string]: unknown; }

Type Declaration

- [key: string]: unknown
- Optional _meta?: { progressToken?: ProgressToken; [key: string]: unknown }

See General fields: `_meta` for notes on `_meta` usage.

- Optional `progressToken?: ProgressToken`

If specified, the caller is requesting out-of-band progress notifications for this request (as represented by notifications/progress). The value of this parameter is an opaque token that will be attached to any subsequent notifications. The receiver is not obligated to provide these notifications.

Inherited from `Request.params`

jsonrpc: "2.0"

id: `RequestId`

JSONRPCResponse

```
interface JSONRPCResponse {
  jsonrpc: "2.0";
  id: RequestId;
  result: Result;
}
```

A successful (non-error) response to a request.

jsonrpc: "2.0"

id: `RequestId`

result: `Result`

Common Types

Annotations

```
interface Annotations {
  audience?: Role[];
  priority?: number;
  lastModified?: string;
}
```

Optional annotations for the client. The client can use annotations to inform how objects are used or displayed

audience?: Role[]

Describes who the intended customer of this object or data is.

It can include multiple entries to indicate content useful for multiple audiences (e.g., `["user", "assistant"]`).

priority?: number

Describes how important this data is for operating the server.

A value of 1 means "most important," and indicates that the data is effectively required, while 0 means "least important," and indicates that the data is entirely optional.

lastModified?: string

The moment the resource was last modified, as an ISO 8601 formatted string.

Should be an ISO 8601 formatted string (e.g., "2025-01-12T15:00:58Z").

Examples: last activity timestamp in an open file, timestamp when the resource was attached, etc.

Cursor

Cursor: string

An opaque token used to represent a cursor for pagination.

EmptyResult

EmptyResult: Result

A response that indicates success but carries no data.

LogLevel

```
LogLevel:  
| "debug"  
| "info"  
| "notice"  
| "warning"  
| "error"  
| "critical"  
| "alert"  
| "emergency"
```

The severity of a log message.

These map to syslog message severities, as specified in RFC-5424:

<https://datatracker.ietf.org/doc/html/rfc5424#section-6.2.1>

ProgressToken

```
ProgressToken: string | number
```

A progress token, used to associate progress notifications with the original request.

RequestId

```
RequestId: string | number
```

A uniquely identifying ID for a request in JSON-RPC.

Result

```
interface Result {  
  _meta?: { [key: string]: unknown };  
  [key: string]: unknown;  
}
```

```
_meta?: { [key: string]: unknown }
```

See General fields: `_meta` for notes on `_meta` usage.

Role

```
Role: "user" | "assistant"
```

The sender or recipient of messages and data in a conversation.

Content

AudioContent

```
interface AudioContent {  
  type: "audio";  
  data: string;  
  mimeType: string;  
  annotations?: Annotations;  
  _meta?: { [key: string]: unknown };  
}
```

Audio provided to or from an LLM.

type: "audio"

data: string

The base64-encoded audio data.

contentType: string

The MIME type of the audio. Different providers may support different audio types.

annotations?: Annotations

Optional annotations for the client.

_meta?: { [key: string]: unknown }

See General fields: `_meta` for notes on `_meta` usage.

BlobResourceContents

```
interface BlobResourceContents {
  uri: string;
  mimeType?: string;
  _meta?: { [key: string]: unknown };
  blob: string;
}
```

uri: string

The URI of this resource.

Inherited from ResourceContents.uri

mimeType?: string

The MIME type of this resource, if known.

Inherited from ResourceContents.mimeType

_meta?: { [key: string]: unknown }

See General fields: `_meta` for notes on `_meta` usage.

Inherited from ResourceContents._meta

blob: string

A base64-encoded string representing the binary data of the item.

ContentBlock

ContentBlock:

- | `TextContent`
- | `ImageContent`
- | `AudioContent`
- | `ResourceLink`
- | `EmbeddedResource`

EmbeddedResource

```
interface EmbeddedResource {
  type: "resource";
  resource: TextResourceContents | BlobResourceContents;
  annotations?: Annotations;
  _meta?: { [key: string]: unknown };
}
```

The contents of a resource, embedded into a prompt or tool call result.

It is up to the client how best to render embedded resources for the benefit of the LLM and/or the user.

type: "resource"

resource: TextResourceContents | BlobResourceContents

annotations?: Annotations

Optional annotations for the client.

_meta?: { [key: string]: unknown }

See General fields: `_meta` for notes on `_meta` usage.

ImageContent

```
interface ImageContent {
  type: "image";
  data: string;
  mimeType: string;
  annotations?: Annotations;
  _meta?: { [key: string]: unknown };
}
```

An image provided to or from an LLM.

type: "image"

data: string

The base64-encoded image data.

mimeType: string

The MIME type of the image. Different providers may support different image types.

annotations?: Annotations

Optional annotations for the client.

_meta?: { [key: string]: unknown }

See General fields: `_meta` for notes on `_meta` usage.

ResourceLink

```
interface ResourceLink {
  name: string;
  title?: string;
  uri: string;
  description?: string;
  mimeType?: string;
  annotations?: Annotations;
  size?: number;
  _meta?: { [key: string]: unknown };
  type: "resource_link";
}
```

A resource that the server is capable of reading, included in a prompt or tool call result.

Note: resource links returned by tools are not guaranteed to appear in the results of `resources/list` requests.

name: string

Intended for programmatic or logical use, but used as a display name in past specs or fallback (if title isn't present).

Inherited from `Resource.name`

title?: string

Intended for UI and end-user contexts — optimized to be human-readable and easily understood, even by those unfamiliar with domain-specific terminology.

If not provided, the name should be used for display (except for Tool, where `annotations.title` should be given precedence over using `name`, if present).

Inherited from `Resource.title`

uri: string

The URI of this resource.

Inherited from Resource.uri

description?: string

A description of what this resource represents.

This can be used by clients to improve the LLM's understanding of available resources. It can be thought of like a "hint" to the model.

Inherited from Resource.description

mimeType?: string

The MIME type of this resource, if known.

Inherited from Resource.mimeType

annotations?: Annotations

Optional annotations for the client.

Inherited from Resource.annotations

size?: number

The size of the raw resource content, in bytes (i.e., before base64 encoding or any tokenization), if known.

This can be used by Hosts to display file sizes and estimate context window usage.

Inherited from Resource.size

_meta?: { [key: string]: unknown }

See General fields: `_meta` for notes on `_meta` usage.

Inherited from Resource._meta

type: "resource_link"**TextContent**

```
interface TextContent {
  type: "text";
  text: string;
  annotations?: Annotations;
  _meta?: { [key: string]: unknown };
}
```

Text provided to or from an LLM.

type: "text"

text: string

The text content of the message.

annotations?: Annotations

Optional annotations for the client.

_meta?: { [key: string]: unknown }

See General fields: `_meta` for notes on `_meta` usage.

TextResourceContents

```
interface TextResourceContents {
  uri: string;
  mimeType?: string;
  _meta?: { [key: string]: unknown };
  text: string;
}
```

uri: string

The URI of this resource.

Inherited from ResourceContents.uri

mimeType?: string

The MIME type of this resource, if known.

Inherited from ResourceContents.mimeType

_meta?: { [key: string]: unknown }

See General fields: `_meta` for notes on `_meta` usage.

Inherited from ResourceContents._meta

text: string

The text of the item. This must only be set if the item can actually be represented as text (not binary data).

completion/complete

CompleteRequest

```
interface CompleteRequest {
  method: "completion/complete";
  params: {
    ref: PromptReference | ResourceTemplateReference;
    argument: { name: string; value: string };
    context?: { arguments?: { [key: string]: string } };
  };
}
```

A request from the client to the server, to ask for completion options.

method: "completion/complete"

Overrides Request.method

params: { ref: PromptReference | ResourceTemplateReference; argument: { name: string; value: string }; context?: { arguments?: { [key: string]: string } }; }

Type Declaration

- ref: `PromptReference | ResourceTemplateReference`
- argument: { name: `string`; value: `string` }

The argument's information

- name: `string`

The name of the argument

- value: `string`

The value of the argument to use for completion matching.

- Optional context?: { arguments?: { [key: `string`]: `string` } }

Additional, optional context for completions

- Optional arguments?: { [key: `string`]: `string` }

Previously-resolved variables in a URI template or prompt.

Overrides Request.params

CompleteResult

```
interface CompleteResult {
  _meta?: { [key: string]: unknown };
  completion: { values: string[]; total?: number; hasMore?: boolean };
  [key: string]: unknown;
}
```

The server's response to a completion/complete request

_meta?: { [key: string]: unknown }

See General fields: `_meta` for notes on `_meta` usage.

Inherited from Result.`_meta`

completion: { values: string[]; total?: number; hasMore?: boolean }

Type Declaration

- values: `string[]`

An array of completion values. Must not exceed 100 items.

- Optional total?: `number`

The total number of completion options available. This can exceed the number of values actually sent in the response.

- Optional hasMore?: `boolean`

Indicates whether there are additional completion options beyond those provided in the current response, even if the exact total is unknown.

PromptReference

```
interface PromptReference {
  name: string;
  title?: string;
  type: "ref/prompt";
}
```

Identifies a prompt.

name: string

Intended for programmatic or logical use, but used as a display name in past specs or fallback (if title isn't present).

Inherited from BaseMetadata.name

title?: string

Intended for UI and end-user contexts — optimized to be human-readable and easily understood, even by those unfamiliar with domain-specific terminology.

If not provided, the name should be used for display (except for Tool, where `annotations.title` should be given precedence over using `name`, if present).

Inherited from BaseMetadata.title

type: "ref/prompt"

ResourceTemplateReference

```
interface ResourceTemplateReference {
  type: "ref/resource";
  uri: string;
}
```

A reference to a resource or resource template definition.

type: "ref/resource"

uri: string

The URI or URI template of the resource.

elicitation/create

ElicitRequest

```
interface ElicitRequest {
  method: "elicitation/create";
  params: {
    message: string;
    requestedSchema: {
      type: "object";
      properties: { [key: string]: PrimitiveSchemaDefinition };
      required?: string[];
    };
  };
}
```

A request from the server to elicit additional information from the user via the client.

method: "elicitation/create"

Overrides Request.method

params: { message: string; requestedSchema: { type: "object"; properties: { [key: string]: PrimitiveSchemaDefinition }; required?: string[]; }; }

Type Declaration

- message: string
The message to present to the user.
- requestedSchema: {
 type: "object";
 properties: { [key: string]: PrimitiveSchemaDefinition };

```
    required?: string[];
  }
```

A restricted subset of JSON Schema. Only top-level properties are allowed, without nesting.

Overrides Request.params

ElicitResult

```
interface ElicitResult {
  _meta?: { [key: string]: unknown };
  action: "accept" | "decline" | "cancel";
  content?: { [key: string]: string | number | boolean };
  [key: string]: unknown;
}
```

The client's response to an elicitation request.

_meta?: { [key: string]: unknown }

See General fields: `_meta` for notes on `_meta` usage.

Inherited from Result._meta

action: "accept" | "decline" | "cancel"

The user action in response to the elicitation.

- "accept": User submitted the form/confirmed the action
- "decline": User explicitly declined the action
- "cancel": User dismissed without making an explicit choice

content?: { [key: string]: string | number | boolean }

The submitted form data, only present when action is "accept". Contains values matching the requested schema.

BooleanSchema

```
interface BooleanSchema {
  type: "boolean";
  title?: string;
  description?: string;
  default?: boolean;
}
```

type: "boolean"

title?: string

description?: string

default?: boolean

EnumSchema

```
interface EnumSchema {  
  type: "string";  
  title?: string;  
  description?: string;  
  enum: string[];  
  enumNames?: string[];  
}
```

type: "string"

title?: string

description?: string

enum: string[]

enumNames?: string[]

NumberSchema

```
interface NumberSchema {  
  type: "number" | "integer";  
  title?: string;  
  description?: string;  
  minimum?: number;  
  maximum?: number;  
}
```

type: "number" | "integer"

title?: string

description?: string

minimum?: number

maximum?: number

PrimitiveSchemaDefinition

```
PrimitiveSchemaDefinition:
| StringSchema
| NumberSchema
| BooleanSchema
| EnumSchema
```

Restricted schema definitions that only allow primitive types without nested objects or arrays.

StringSchema

```
interface StringSchema {
  type: "string";
  title?: string;
  description?: string;
  minLength?: number;
  maxLength?: number;
  format?: "uri" | "email" | "date" | "date-time";
}
```

```
| type: "string"
```

```
| title?: string
```

```
| description?: string
```

```
| minLength?: number
```

```
| maxLength?: number
```

```
| format?: "uri" | "email" | "date" | "date-time"
```

initialize

InitializeRequest

```
interface InitializeRequest {
  method: "initialize";
  params: {
    protocolVersion: string;
    capabilities: ClientCapabilities;
    clientInfo: Implementation;
  };
}
```

This request is sent from the client to the server when it first connects, asking it to begin initialization.

method: "initialize"

Overrides Request.method

params: { protocolVersion: string; capabilities: ClientCapabilities; clientInfo: Implementation; }

Type Declaration

- protocolVersion: [string](#)

The latest version of the Model Context Protocol that the client supports. The client MAY decide to support older versions as well.

- capabilities: [ClientCapabilities](#)
- clientInfo: [Implementation](#)

Overrides Request.params

InitializeResult

```
interface InitializeResult {
  _meta?: { [key: string]: unknown };
  protocolVersion: string;
  capabilities: ServerCapabilities;
  serverInfo: Implementation;
  instructions?: string;
  [key: string]: unknown;
}
```

After receiving an initialize request from the client, the server sends this response.

_meta?: { [key: string]: unknown }

See General fields: [_meta](#) for notes on [_meta](#) usage.

Inherited from Result.[_meta](#)

protocolVersion: string

The version of the Model Context Protocol that the server wants to use. This may not match the version that the client requested. If the client cannot support this version, it MUST disconnect.

capabilities: ServerCapabilities

serverInfo: Implementation

instructions?: string

Instructions describing how to use the server and its features.

This can be used by clients to improve the LLM's understanding of available tools, resources, etc. It can be thought of like a "hint" to the model. For example, this information MAY be added to the system prompt.

ClientCapabilities

```
interface ClientCapabilities {
  experimental?: { [key: string]: object };
  roots?: { listChanged?: boolean };
  sampling?: object;
  elicitation?: object;
}
```

Capabilities a client may support. Known capabilities are defined here, in this schema, but this is not a closed set: any client can define its own, additional capabilities.

experimental?: { [key: string]: object }

Experimental, non-standard capabilities that the client supports.

roots?: { listChanged?: boolean }

Present if the client supports listing roots.

Type Declaration

- Optional listChanged?: boolean

Whether the client supports notifications for changes to the roots list.

sampling?: object

Present if the client supports sampling from an LLM.

elicitation?: object

Present if the client supports elicitation from the server.

Implementation

```
interface Implementation {
  name: string;
  title?: string;
  version: string;
}
```

Describes the name and version of an MCP implementation, with an optional title for UI representation.

name: string

Intended for programmatic or logical use, but used as a display name in past specs or fallback (if title isn't present).

Inherited from BaseMetadata.name

title?: string

Intended for UI and end-user contexts — optimized to be human-readable and easily understood, even by those unfamiliar with domain-specific terminology.

If not provided, the name should be used for display (except for Tool, where `annotations.title` should be given precedence over using `name`, if present).

Inherited from BaseMetadata.title

version: string**ServerCapabilities**

```
interface ServerCapabilities {
  experimental?: { [key: string]: object };
  logging?: object;
  completions?: object;
  prompts?: { listChanged?: boolean };
  resources?: { subscribe?: boolean; listChanged?: boolean };
  tools?: { listChanged?: boolean };
}
```

Capabilities that a server may support. Known capabilities are defined here, in this schema, but this is not a closed set: any server can define its own, additional capabilities.

experimental?: { [key: string]: object }

Experimental, non-standard capabilities that the server supports.

logging?: object

Present if the server supports sending log messages to the client.

completions?: object

Present if the server supports argument autocompletion suggestions.

prompts?: { listChanged?: boolean }

Present if the server offers any prompt templates.

Type Declaration

- Optional listChanged?: boolean

Whether this server supports notifications for changes to the prompt list.

resources?: { subscribe?: boolean; listChanged?: boolean }

Present if the server offers any resources to read.

Type Declaration

- Optional subscribe?: [boolean](#)

Whether this server supports subscribing to resource updates.

- Optional listChanged?: [boolean](#)

Whether this server supports notifications for changes to the resource list.

tools?: { listChanged?: boolean }

Present if the server offers any tools to call.

Type Declaration

- Optional listChanged?: [boolean](#)

Whether this server supports notifications for changes to the tool list.

logging/setLevel

SetLevelRequest

```
interface SetLevelRequest {
  method: "logging/setLevel";
  params: { level: LoggingLevel };
}
```

A request from the client to the server, to enable or adjust logging.

method: "logging/setLevel"

Overrides Request.method

params: { level: [LoggingLevel](#) }

Type Declaration

- level: [LoggingLevel](#)

The level of logging that the client wants to receive from the server. The server should send all logs at this level and higher (i.e., more severe) to the client as notifications/message.

Overrides Request.params

notifications/cancelled

CancelledNotification

```
interface CancelledNotification {
  method: "notifications/cancelled";
  params: { requestId: RequestId; reason?: string };
}
```

This notification can be sent by either side to indicate that it is cancelling a previously-issued request.

The request SHOULD still be in-flight, but due to communication latency, it is always possible that this notification MAY arrive after the request has already finished.

This notification indicates that the result will be unused, so any associated processing SHOULD cease.

A client MUST NOT attempt to cancel its `initialize` request.

method: "notifications/cancelled"

Overrides Notification.method

params: { requestId: RequestId; reason?: string }

Type Declaration

- requestId: RequestId

The ID of the request to cancel.

This MUST correspond to the ID of a request previously issued in the same direction.

- Optional reason?: string

An optional string describing the reason for the cancellation. This MAY be logged or presented to the user.

Overrides Notification.params

notifications/initialized

InitializedNotification

```
interface InitializedNotification {
  params?: { _meta?: { [key: string]: unknown }; [key: string]: unknown };
  method: "notifications/initialized";
}
```

This notification is sent from the client to the server after initialization has finished.

params?: { **_meta?:** { [key: string]: unknown }; [key: string]: unknown }

Type Declaration

- [key: string]: unknown
- Optional **_meta?:** { [key: string]: unknown }

See General fields: **_meta** for notes on **_meta** usage.

Inherited from Notification.params

method: "notifications/initialized"

Overrides Notification.method

notifications/message

LoggingMessageNotification

```
interface LoggingMessageNotification {
  method: "notifications/message";
  params: { level: LoggingLevel; logger?: string; data: unknown };
}
```

Notification of a log message passed from server to client. If no logging/setLevel request has been sent from the client, the server MAY decide which messages to send automatically.

method: "notifications/message"

Overrides Notification.method

params: { **level:** LoggingLevel; **logger?:** string; **data:** unknown }

Type Declaration

- level: LoggingLevel

The severity of this log message.

- Optional logger?: string

An optional name of the logger issuing this message.

- data: unknown

The data to be logged, such as a string message or an object. Any JSON serializable type is allowed here.

Overrides Notification.params

notifications/progress

ProgressNotification

```
interface ProgressNotification {
  method: "notifications/progress";
  params: {
    progressToken: ProgressToken;
    progress: number;
    total?: number;
    message?: string;
  };
}
```

An out-of-band notification used to inform the receiver of a progress update for a long-running request.

method: "notifications/progress"

Overrides Notification.method

params: { progressToken: ProgressToken; progress: number; total?: number; message?: string; }

Type Declaration

- progressToken: [ProgressToken](#)

The progress token which was given in the initial request, used to associate this notification with the request that is proceeding.

- progress: [number](#)

The progress thus far. This should increase every time progress is made, even if the total is unknown.

- Optional total?: [number](#)

Total number of items to process (or total progress required), if known.

- Optional message?: [string](#)

An optional message describing the current progress.

Overrides Notification.params

notifications/prompts/list_changed

PromptListChangedNotification

```
interface PromptListChangedNotification {
  params?: { _meta?: { [key: string]: unknown }; [key: string]: unknown };
  method: "notifications/prompts/list_changed";
}
```

An optional notification from the server to the client, informing it that the list of prompts it offers has changed. This may be issued by servers without any previous subscription from the client.

params?: { _meta?: { [key: string]: unknown }; [key: string]: unknown }

Type Declaration

- [key: string]: unknown
- Optional _meta?: { [key: string]: unknown }

See General fields: `_meta` for notes on `_meta` usage.

Inherited from Notification.params

method: "notifications/prompts/list_changed"

Overrides Notification.method

notifications/resources/list_changed

ResourceListChangedNotification

```
interface ResourceListChangedNotification {
  params?: { _meta?: { [key: string]: unknown }; [key: string]: unknown };
  method: "notifications/resources/list_changed";
}
```

An optional notification from the server to the client, informing it that the list of resources it can read from has changed. This may be issued by servers without any previous subscription from the client.

params?: { _meta?: { [key: string]: unknown }; [key: string]: unknown }

Type Declaration

- [key: string]: unknown
- Optional _meta?: { [key: string]: unknown }

See General fields: `_meta` for notes on `_meta` usage.

Inherited from Notification.params

method: "notifications/resources/list_changed"

Overrides Notification.method

notifications/resources/updated

ResourceUpdatedNotification

```
interface ResourceUpdatedNotification {
  method: "notifications/resources/updated";
  params: { uri: string };
}
```

A notification from the server to the client, informing it that a resource has changed and may need to be read again. This should only be sent if the client previously sent a resources/subscribe request.

method: "notifications/resources/updated"

Overrides Notification.method

params: { uri: string }

Type Declaration

- uri: `string`

The URI of the resource that has been updated. This might be a sub-resource of the one that the client actually subscribed to.

Overrides Notification.params

notifications/roots/list_changed

RootsListChangedNotification

```
interface RootsListChangedNotification {
  params?: { _meta?: { [key: string]: unknown }; [key: string]: unknown };
  method: "notifications/roots/list_changed";
}
```

A notification from the client to the server, informing it that the list of roots has changed. This notification should be sent whenever the client adds, removes, or modifies any root. The server should then request an updated list of roots using the ListRootsRequest.

params?: { _meta?: { [key: string]: unknown }; [key: string]: unknown }

Type Declaration

- [key: `string`]: `unknown`
- Optional `_meta?: { [key: string]: unknown }`

See General fields: `_meta` for notes on `_meta` usage.

Inherited from Notification.params

method: "notifications/roots/list_changed"

Overrides Notification.method

notifications/tools/list_changed

ToolListChangedNotification

```
interface ToolListChangedNotification {
  params?: { _meta?: { [key: string]: unknown }; [key: string]: unknown };
  method: "notifications/tools/list_changed";
}
```

An optional notification from the server to the client, informing it that the list of tools it offers has changed. This may be issued by servers without any previous subscription from the client.

params?: { _meta?: { [key: string]: unknown }; [key: string]: unknown }

Type Declaration

- [key: string]: unknown
- Optional _meta?: { [key: string]: unknown }

See General fields: `_meta` for notes on `_meta` usage.

Inherited from Notification.params

method: "notifications/tools/list_changed"

Overrides Notification.method

ping

PingRequest

```
interface PingRequest {
  params?: {
    _meta?: { progressToken?: ProgressToken; [key: string]: unknown };
    [key: string]: unknown;
  };
  method: "ping";
}
```

A ping, issued by either the server or the client, to check that the other party is still alive. The receiver must promptly respond, or else may be disconnected.

```
params?: { _meta?: { progressToken?: ProgressToken; [key: string]: unknown }; [key: string]: unknown; }
```

Type Declaration

- [key: [string](#)]: [unknown](#)
- Optional `_meta?: { progressToken?: ProgressToken; [key: string]: unknown }`

See General fields: `_meta` for notes on `_meta` usage.

- Optional `progressToken?: ProgressToken`

If specified, the caller is requesting out-of-band progress notifications for this request (as represented by notifications/progress). The value of this parameter is an opaque token that will be attached to any subsequent notifications. The receiver is not obligated to provide these notifications.

Inherited from `Request.params`

method: "ping"

Overrides `Request.method`

prompts/get

GetPromptRequest

```
interface GetPromptRequest {
  method: "prompts/get";
  params: { name: string; arguments?: { [key: string]: string } };
}
```

Used by the client to get a prompt provided by the server.

method: "prompts/get"

Overrides `Request.method`

```
params: { name: string; arguments?: { [key: string]: string } }
```

Type Declaration

- name: [string](#)
The name of the prompt or prompt template.
- Optional `arguments?: { [key: string]: string }`
Arguments to use for templating the prompt.

Overrides `Request.params`

GetPromptResult

```
interface GetPromptResult {
  _meta?: { [key: string]: unknown };
  description?: string;
  messages: PromptMessage[];
  [key: string]: unknown;
}
```

The server's response to a prompts/get request from the client.

_meta?: { [key: string]: unknown }

See General fields: `_meta` for notes on `_meta` usage.

Inherited from Result._meta

description?: string

An optional description for the prompt.

messages: PromptMessage[]

PromptMessage

```
interface PromptMessage {
  role: Role;
  content: ContentBlock;
}
```

Describes a message returned as part of a prompt.

This is similar to `SamplingMessage`, but also supports the embedding of resources from the MCP server.

role: Role

content: ContentBlock

prompts/list

ListPromptsRequest

```
interface ListPromptsRequest {
  params?: { cursor?: string };
  method: "prompts/list";
}
```

Sent from the client to request a list of prompts and prompt templates the server has.

params?: { cursor?: string }

Type Declaration

- Optional cursor?: string

An opaque token representing the current pagination position. If provided, the server should return results starting after this cursor.

Inherited from `PaginatedRequest.params`

method: "prompts/list"

Overrides `PaginatedRequest.method`

ListPromptsResult

```
interface ListPromptsResult {
  _meta?: { [key: string]: unknown };
  nextCursor?: string;
  prompts: Prompt[];
  [key: string]: unknown;
}
```

The server's response to a prompts/list request from the client.

_meta?: { [key: string]: unknown }

See General fields: `_meta` for notes on `_meta` usage.

Inherited from `PaginatedResult._meta`

nextCursor?: string

An opaque token representing the pagination position after the last returned result. If present, there may be more results available.

Inherited from `PaginatedResult.nextCursor`

prompts: Prompt[]

Prompt

```
interface Prompt {
  name: string;
  title?: string;
  description?: string;
  arguments?: PromptArgument[];
  _meta?: { [key: string]: unknown };
}
```

A prompt or prompt template that the server offers.

name: string

Intended for programmatic or logical use, but used as a display name in past specs or fallback (if title isn't present).

Inherited from BaseMetadata.name

title?: string

Intended for UI and end-user contexts — optimized to be human-readable and easily understood, even by those unfamiliar with domain-specific terminology.

If not provided, the name should be used for display (except for Tool, where `annotations.title` should be given precedence over using `name`, if present).

Inherited from BaseMetadata.title

description?: string

An optional description of what this prompt provides

arguments?: PromptArgument[]

A list of arguments to use for templating the prompt.

_meta?: { [key: string]: unknown }

See General fields: `_meta` for notes on `_meta` usage.

PromptArgument

```
interface PromptArgument {
  name: string;
  title?: string;
  description?: string;
  required?: boolean;
}
```

Describes an argument that a prompt can accept.

name: string

Intended for programmatic or logical use, but used as a display name in past specs or fallback (if title isn't present).

Inherited from `BaseMetadata.name`

title?: string

Intended for UI and end-user contexts — optimized to be human-readable and easily understood, even by those unfamiliar with domain-specific terminology.

If not provided, the name should be used for display (except for Tool, where `annotations.title` should be given precedence over using `name`, if present).

Inherited from `BaseMetadata.title`

description?: string

A human-readable description of the argument.

required?: boolean

Whether this argument must be provided.

resources/list

ListResourcesRequest

```
interface ListResourcesRequest {
  params?: { cursor?: string };
  method: "resources/list";
}
```

Sent from the client to request a list of resources the server has.

params?: { cursor?: string }

Type Declaration

- Optional `cursor?: string`

An opaque token representing the current pagination position. If provided, the server should return results starting after this cursor.

Inherited from `PaginatedRequest.params`

method: "resources/list"

Overrides `PaginatedRequest.method`

ListResourcesResult

```
interface ListResourcesResult {
  _meta?: { [key: string]: unknown };
  nextCursor?: string;
  resources: Resource[];
  [key: string]: unknown;
}
```

The server's response to a `resources/list` request from the client.

`_meta?: { [key: string]: unknown }`

See General fields: `_meta` for notes on `_meta` usage.

Inherited from `PaginatedResult._meta`

`nextCursor?: string`

An opaque token representing the pagination position after the last returned result. If present, there may be more results available.

Inherited from `PaginatedResult.nextCursor`

`resources: Resource[]`

Resource

```
interface Resource {
  name: string;
  title?: string;
  uri: string;
  description?: string;
  mimeType?: string;
  annotations?: Annotations;
  size?: number;
  _meta?: { [key: string]: unknown };
}
```

A known resource that the server is capable of reading.

`name: string`

Intended for programmatic or logical use, but used as a display name in past specs or fallback (if title isn't present).

Inherited from `BaseMetadata.name`

`title?: string`

Intended for UI and end-user contexts — optimized to be human-readable and easily understood, even by those unfamiliar with domain-specific terminology.

If not provided, the name should be used for display (except for Tool, where `annotations.title` should be given precedence over using `name`, if present).

Inherited from `BaseMetadata.title`

uri: string

The URI of this resource.

description?: string

A description of what this resource represents.

This can be used by clients to improve the LLM's understanding of available resources. It can be thought of like a "hint" to the model.

mimeType?: string

The MIME type of this resource, if known.

annotations?: Annotations

Optional annotations for the client.

size?: number

The size of the raw resource content, in bytes (i.e., before base64 encoding or any tokenization), if known.

This can be used by Hosts to display file sizes and estimate context window usage.

_meta?: { [key: string]: unknown }

See General fields: `_meta` for notes on `_meta` usage.

resources/read

ReadResourceRequest

```
interface ReadResourceRequest {
  method: "resources/read";
  params: { uri: string };
}
```

Sent from the client to the server, to read a specific resource URI.

method: "resources/read"

Overrides `Request.method`

params: { uri: string }

Type Declaration

- uri: [string](#)

The URI of the resource to read. The URI can use any protocol; it is up to the server how to interpret it.

Overrides Request.params

ReadResourceResult

```
interface ReadResourceResult {
  _meta?: { [key: string]: unknown };
  contents: (TextResourceContents | BlobResourceContents)[];
  [key: string]: unknown;
}
```

The server's response to a resources/read request from the client.

_meta?: { [key: string]: unknown }

See General fields: `_meta` for notes on `_meta` usage.

Inherited from Result._meta

contents: (TextResourceContents | BlobResourceContents)[]

resources/subscribe

SubscribeRequest

```
interface SubscribeRequest {
  method: "resources/subscribe";
  params: { uri: string };
}
```

Sent from the client to request resources/updated notifications from the server whenever a particular resource changes.

method: "resources/subscribe"

Overrides Request.method

params: { uri: string }

Type Declaration

- uri: [string](#)

The URI of the resource to subscribe to. The URI can use any protocol; it is up to the server how to interpret it.

Overrides Request.params

resources/templates/list

ListResourceTemplatesRequest

```
interface ListResourceTemplatesRequest {
  params?: { cursor?: string };
  method: "resources/templates/list";
}
```

Sent from the client to request a list of resource templates the server has.

params?: { cursor?: string }

Type Declaration

- Optional cursor?: string

An opaque token representing the current pagination position. If provided, the server should return results starting after this cursor.

Inherited from PaginatedRequest.params

method: "resources/templates/list"

Overrides PaginatedRequest.method

ListResourceTemplatesResult

```
interface ListResourceTemplatesResult {
  _meta?: { [key: string]: unknown };
  nextCursor?: string;
  resourceTemplates: ResourceTemplate[];
  [key: string]: unknown;
}
```

The server's response to a resources/templates/list request from the client.

_meta?: { [key: string]: unknown }

See General fields: `_meta` for notes on `_meta` usage.

Inherited from PaginatedResult._meta

nextCursor?: string

An opaque token representing the pagination position after the last returned result. If present, there may be more results available.

Inherited from `PaginatedResult.nextCursor`

resourceTemplates: ResourceTemplate[]

ResourceTemplate

```
interface ResourceTemplate {
  name: string;
  title?: string;
  uriTemplate: string;
  description?: string;
  mimeType?: string;
  annotations?: Annotations;
  _meta?: { [key: string]: unknown };
}
```

A template description for resources available on the server.

name: string

Intended for programmatic or logical use, but used as a display name in past specs or fallback (if title isn't present).

Inherited from `BaseMetadata.name`

title?: string

Intended for UI and end-user contexts — optimized to be human-readable and easily understood, even by those unfamiliar with domain-specific terminology.

If not provided, the name should be used for display (except for Tool, where `annotations.title` should be given precedence over using `name`, if present).

Inherited from `BaseMetadata.title`

uriTemplate: string

A URI template (according to RFC 6570) that can be used to construct resource URIs.

description?: string

A description of what this template is for.

This can be used by clients to improve the LLM's understanding of available resources. It can be thought of like a "hint" to the model.

mimeType?: string

The MIME type for all resources that match this template. This should only be included if all resources matching this template have the same type.

annotations?: Annotations

Optional annotations for the client.

```
_meta?: { [key: string]: unknown }
```

See General fields: `_meta` for notes on `_meta` usage.

resources/unsubscribe

UnsubscribeRequest

```
interface UnsubscribeRequest {
  method: "resources/unsubscribe";
  params: { uri: string };
}
```

Sent from the client to request cancellation of resources/updated notifications from the server. This should follow a previous resources/subscribe request.

method: "resources/unsubscribe"

Overrides Request.method

params: { uri: string }

Type Declaration

- uri: `string`

The URI of the resource to unsubscribe from.

Overrides Request.params

roots/list

ListRootsRequest

```
interface ListRootsRequest {
  params?: {
    _meta?: { progressToken?: ProgressToken; [key: string]: unknown };
    [key: string]: unknown;
  };
  method: "roots/list";
}
```

Sent from the server to request a list of root URIs from the client. Roots allow servers to ask for specific directories or files to operate on. A common example for roots is providing a set of repositories or directories a server should operate on.

This request is typically used when the server needs to understand the file system structure or access specific locations that the client has permission to read from.

```
params?: { _meta?: { progressToken?: ProgressToken; [key: string]: unknown }; [key: string]: unknown; }
```

Type Declaration

- [key: [string](#)]: [unknown](#)
- Optional `_meta?: { progressToken?: ProgressToken; [key: string]: unknown }`

See General fields: `_meta` for notes on `_meta` usage.

- Optional `progressToken?: ProgressToken`

If specified, the caller is requesting out-of-band progress notifications for this request (as represented by notifications/progress). The value of this parameter is an opaque token that will be attached to any subsequent notifications. The receiver is not obligated to provide these notifications.

Inherited from `Request.params`

method: "roots/list"

Overrides `Request.method`

ListRootsResult

```
interface ListRootsResult {
  _meta?: { [key: string]: unknown };
  roots: Root[];
  [key: string]: unknown;
}
```

The client's response to a roots/list request from the server. This result contains an array of `Root` objects, each representing a root directory or file that the server can operate on.

```
_meta?: { [key: string]: unknown }
```

See General fields: `_meta` for notes on `_meta` usage.

Inherited from `Result._meta`

roots: Root[]

Root

```
interface Root {
  uri: string;
  name?: string;
  _meta?: { [key: string]: unknown };
}
```

Represents a root directory or file that the server can operate on.

uri: string

The URI identifying the root. This *must* start with file:// for now. This restriction may be relaxed in future versions of the protocol to allow other URI schemes.

name?: string

An optional name for the root. This can be used to provide a human-readable identifier for the root, which may be useful for display purposes or for referencing the root in other parts of the application.

_meta?: { [key: string]: unknown }

See General fields: `_meta` for notes on `_meta` usage.

sampling/createMessage

CreateMessageRequest

```
interface CreateMessageRequest {
  method: "sampling/createMessage";
  params: {
    messages: SamplingMessage[];
    modelPreferences?: ModelPreferences;
    systemPrompt?: string;
    includeContext?: "none" | "thisServer" | "allServers";
    temperature?: number;
    maxTokens: number;
    stopSequences?: string[];
    metadata?: object;
  };
}
```

A request from the server to sample an LLM via the client. The client has full discretion over which model to select. The client should also inform the user before beginning sampling, to allow them to inspect the request (human in the loop) and decide whether to approve it.

method: "sampling/createMessage"

Overrides Request.method

```
params: { messages: SamplingMessage[]; modelPreferences?: ModelPreferences;
systemPrompt?: string; includeContext?: "none" | "thisServer" |
"allServers"; temperature?: number; maxTokens: number; stopSequences?:
string[]; metadata?: object; }
```

Type Declaration

- messages: [SamplingMessage\[\]](#)
- Optional modelPreferences?: [ModelPreferences](#)

The server's preferences for which model to select. The client MAY ignore these preferences.

- Optional systemPrompt?: [string](#)

An optional system prompt the server wants to use for sampling. The client MAY modify or omit this prompt.

- Optional includeContext?: ["none"](#) | ["thisServer"](#) | ["allServers"](#)

A request to include context from one or more MCP servers (including the caller), to be attached to the prompt. The client MAY ignore this request.

- Optional temperature?: [number](#)
- maxTokens: [number](#)

The requested maximum number of tokens to sample (to prevent runaway completions).

The client MAY choose to sample fewer tokens than the requested maximum.

- Optional stopSequences?: [string\[\]](#)
- Optional metadata?: [object](#)

Optional metadata to pass through to the LLM provider. The format of this metadata is provider-specific.

Overrides Request.params

CreateMessageResult

```
interface CreateMessageResult {
  _meta?: { [key: string]: unknown };
  model: string;
  stopReason?: string;
  role: Role;
  content: TextContent | ImageContent | AudioContent;
  [key: string]: unknown;
}
```

The client's response to a `sampling/create_message` request from the server. The client should inform the user before returning the sampled message, to allow them to inspect the response (human in the loop) and decide whether to allow the server to see it.

`_meta?: { [key: string]: unknown }`

See General fields: `_meta` for notes on `_meta` usage.

Inherited from `Result._meta`

`model: string`

The name of the model that generated the message.

`stopReason?: string`

The reason why sampling stopped, if known.

`role: Role`

Inherited from `SamplingMessage.role`

`content: TextContent | ImageContent | AudioContent`

Inherited from `SamplingMessage.content`

ModelHint

```
interface ModelHint {
  name?: string;
}
```

Hints to use for model selection.

Keys not declared here are currently left unspecified by the spec and are up to the client to interpret.

`name?: string`

A hint for a model name.

The client SHOULD treat this as a substring of a model name; for example:

- `claude-3-5-sonnet` should match `claude-3-5-sonnet-20241022`
- `sonnet` should match `claude-3-5-sonnet-20241022`, `claude-3-sonnet-20240229`, etc.
- `claude` should match any Claude model

The client MAY also map the string to a different provider's model name or a different model family, as long as it fills a similar niche; for example:

- `gemini-1.5-flash` could match `claude-3-haiku-20240307`

ModelPreferences

```
interface ModelPreferences {
  hints?: ModelHint[];
  costPriority?: number;
  speedPriority?: number;
  intelligencePriority?: number;
}
```

The server's preferences for model selection, requested of the client during sampling.

Because LLMs can vary along multiple dimensions, choosing the "best" model is rarely straightforward. Different models excel in different areas—some are faster but less capable, others are more capable but more expensive, and so on. This interface allows servers to express their priorities across multiple dimensions to help clients make an appropriate selection for their use case.

These preferences are always advisory. The client MAY ignore them. It is also up to the client to decide how to interpret these preferences and how to balance them against other considerations.

hints?: ModelHint[]

Optional hints to use for model selection.

If multiple hints are specified, the client MUST evaluate them in order (such that the first match is taken).

The client SHOULD prioritize these hints over the numeric priorities, but MAY still use the priorities to select from ambiguous matches.

costPriority?: number

How much to prioritize cost when selecting a model. A value of 0 means cost is not important, while a value of 1 means cost is the most important factor.

speedPriority?: number

How much to prioritize sampling speed (latency) when selecting a model. A value of 0 means speed is not important, while a value of 1 means speed is the most important factor.

intelligencePriority?: number

How much to prioritize intelligence and capabilities when selecting a model. A value of 0 means intelligence is not important, while a value of 1 means intelligence is the most important factor.

SamplingMessage

```
interface SamplingMessage {
  role: Role;
  content: TextContent | ImageContent | AudioContent;
}
```

Describes a message issued to or received from an LLM API.

role: Role

content: TextContent | ImageContent | AudioContent

tools/call

CallToolRequest

```
interface CallToolRequest {
  method: "tools/call";
  params: { name: string; arguments?: { [key: string]: unknown } };
}
```

Used by the client to invoke a tool provided by the server.

method: "tools/call"

Overrides Request.method

params: { name: string; arguments?: { [key: string]: unknown } }

Overrides Request.params

CallToolResult

```
interface CallToolResult {
  _meta?: { [key: string]: unknown };
  content: ContentBlock[];
  structuredContent?: { [key: string]: unknown };
  isError?: boolean;
  [key: string]: unknown;
}
```

The server's response to a tool call.

_meta?: { [key: string]: unknown }

See General fields: `_meta` for notes on `_meta` usage.

Inherited from Result._meta

content: ContentBlock[]

A list of content objects that represent the unstructured result of the tool call.

structuredContent?: { [key: string]: unknown }

An optional JSON object that represents the structured result of the tool call.

isError?: boolean

Whether the tool call ended in an error.

If not set, this is assumed to be false (the call was successful).

Any errors that originate from the tool **SHOULD** be reported inside the result object, with `isError` set to true, *not* as an MCP protocol-level error response. Otherwise, the LLM would not be able to see that an error occurred and self-correct.

However, any errors in *finding* the tool, an error indicating that the server does not support tool calls, or any other exceptional conditions, should be reported as an MCP error response.

tools/list**ListToolsRequest**

```
interface ListToolsRequest {
  params?: { cursor?: string };
  method: "tools/list";
}
```

Sent from the client to request a list of tools the server has.

params?: { cursor?: string }

Type Declaration

- Optional cursor?: string

An opaque token representing the current pagination position. If provided, the server should return results starting after this cursor.

Inherited from `PaginatedRequest.params`

method: "tools/list"

Overrides `PaginatedRequest.method`

ListToolsResult

```
interface ListToolsResult {
  _meta?: { [key: string]: unknown };
  nextCursor?: string;
  tools: Tool[];
  [key: string]: unknown;
}
```

The server's response to a tools/list request from the client.

_meta?: { [key: string]: unknown }

See General fields: `_meta` for notes on `_meta` usage.

Inherited from `PaginatedResult._meta`

nextCursor?: string

An opaque token representing the pagination position after the last returned result. If present, there may be more results available.

Inherited from `PaginatedResult.nextCursor`

tools: Tool[]

Tool

```
interface Tool {
  name: string;
  title?: string;
  description?: string;
  inputSchema: {
    type: "object";
    properties?: { [key: string]: object };
    required?: string[];
  };
  outputSchema?: {
    type: "object";
    properties?: { [key: string]: object };
    required?: string[];
  };
  annotations?: ToolAnnotations;
  _meta?: { [key: string]: unknown };
}
```

Definition for a tool the client can call.

name: string

Intended for programmatic or logical use, but used as a display name in past specs or fallback (if title isn't present).

Inherited from `BaseMetadata.name`

title?: string

Intended for UI and end-user contexts — optimized to be human-readable and easily understood, even by those unfamiliar with domain-specific terminology.

If not provided, the name should be used for display (except for `Tool`, where `annotations.title` should be given precedence over using `name`, if present).

Inherited from `BaseMetadata.title`

description?: string

A human-readable description of the tool.

This can be used by clients to improve the LLM's understanding of available tools. It can be thought of like a "hint" to the model.

inputSchema: { type: "object"; properties?: { [key: string]: object }; required?: string[]; }

A JSON Schema object defining the expected parameters for the tool.

outputSchema?: { type: "object"; properties?: { [key: string]: object }; required?: string[]; }

An optional JSON Schema object defining the structure of the tool's output returned in the `structuredContent` field of a `CallToolResult`.

annotations?: ToolAnnotations

Optional additional tool information.

Display name precedence order is: title, annotations.title, then name.

_meta?: { [key: string]: unknown }

See General fields: `_meta` for notes on `_meta` usage.

ToolAnnotations

```
interface ToolAnnotations {
  title?: string;
  readOnlyHint?: boolean;
  destructiveHint?: boolean;
  idempotentHint?: boolean;
  openWorldHint?: boolean;
}
```

Additional properties describing a Tool to clients.

NOTE: all properties in `ToolAnnotations` are **hints**. They are not guaranteed to provide a faithful description of tool behavior (including descriptive properties like `title`).

Clients should never make tool use decisions based on `ToolAnnotations` received from untrusted servers.

title?: string

A human-readable title for the tool.

readOnlyHint?: boolean

If true, the tool does not modify its environment.

Default: false

destructiveHint?: boolean

If true, the tool may perform destructive updates to its environment. If false, the tool performs only additive updates.

(This property is meaningful only when `readOnlyHint == false`)

Default: true

idempotentHint?: boolean

If true, calling the tool repeatedly with the same arguments will have no additional effect on the its environment.

(This property is meaningful only when `readOnlyHint == false`)

Default: false

openWorldHint?: boolean

If true, this tool may interact with an "open world" of external entities. If false, the tool's domain of interaction is closed. For example, the world of a web search tool is open, whereas that of a memory tool is not.

Default: true