Specification · Superseded

Open Model Context Protocol
Version 2025-11-25

Built from `7596f66` · 2026-09-23

> **SUPERSEDED**
>
> This version has been superseded by 2026-07-28.

Contents

- 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, 2025-06-18.

Major changes

1. Enhance authorization server discovery with support for OpenID Connect Discovery 1.0. (PR #797)
2. Allow servers to expose icons as additional metadata for tools, resources, resource templates, and prompts (SEP-973).
3. Enhance authorization flows with incremental scope consent via `WWW-Authenticate` (SEP-835)
4. Provide guidance on tool names (SEP-986)
5. Update `ElicitResult` and `EnumSchema` to use a more standards-based approach and support titled, untitled, single-select, and multi-select enums (SEP-1330).
6. Added support for URL mode elicitation (SEP-1036)
7. Add tool calling support to sampling via `tools` and `toolChoice` parameters (SEP-1577)
8. Add support for OAuth Client ID Metadata Documents as a recommended client registration mechanism (SEP-991, PR #1296)
9. Add experimental support for tasks to enable tracking durable requests with polling and deferred result retrieval (SEP-1686).

Minor changes

1. Clarify that servers using stdio transport may use stderr for all types of logging, not just error messages (PR #670).
2. Add optional `description` field to `Implementation` interface to align with MCP registry server.json format and provide human-readable context during initialization.
3. Clarify that servers must respond with HTTP 403 Forbidden for invalid Origin headers in Streamable HTTP transport. (PR #1439)
4. Updated the Security Best Practices guidance.
5. Clarify that input validation errors should be returned as Tool Execution Errors rather than Protocol Errors to enable model self-correction (SEP-1303).
6. Support polling SSE streams by allowing servers to disconnect at will (SEP-1699).
7. Clarify SEP-1699: GET streams support polling, resumption always via GET regardless of stream origin, event IDs should encode stream identity, disconnection includes server-initiated closure (Issue #1847).
8. Align OAuth 2.0 Protected Resource Metadata discovery with RFC 9728, making `WWW-Authenticate` header optional with fallback to `.well-known` endpoint (SEP-985).
9. Add support for default values in all primitive types (string, number, enum) for elicitation schemas (SEP-1034).

10. Establish JSON Schema 2020-12 as the default dialect for MCP schema definitions (SEP-1613).

Other schema changes

1. Decouple request payloads from RPC method definitions into standalone parameter schemas. (SEP-1319, PR #1284)

Governance and process updates

1. Formalize Model Context Protocol governance structure (SEP-932).
2. Establish shared communication practices and guidelines for the MCP community (SEP-994).
3. Formalize Working Groups and Interest Groups in MCP governance (SEP-1302).
4. Establish SDK tiering system with clear requirements for feature support and maintenance commitments (SEP-1730).

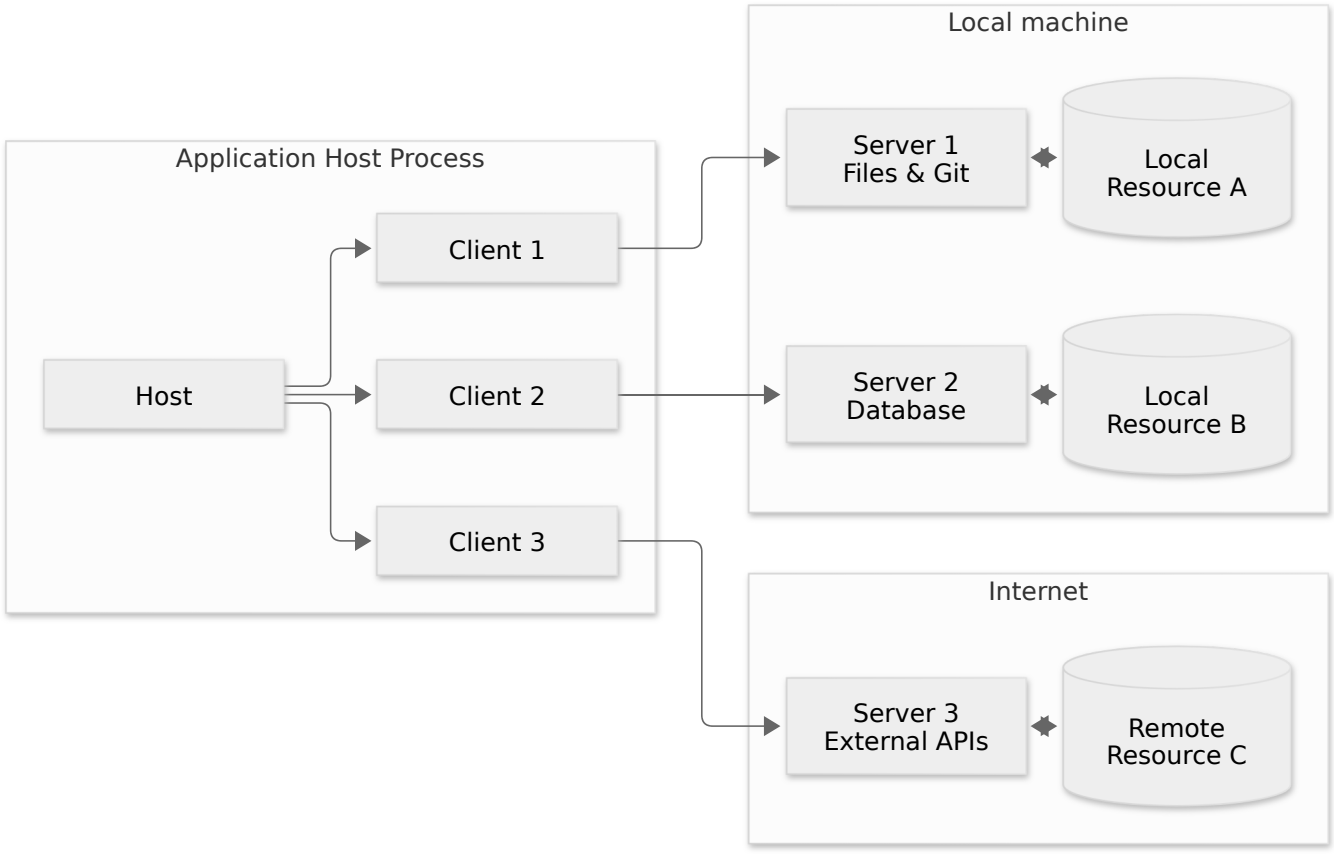
Full changelog

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

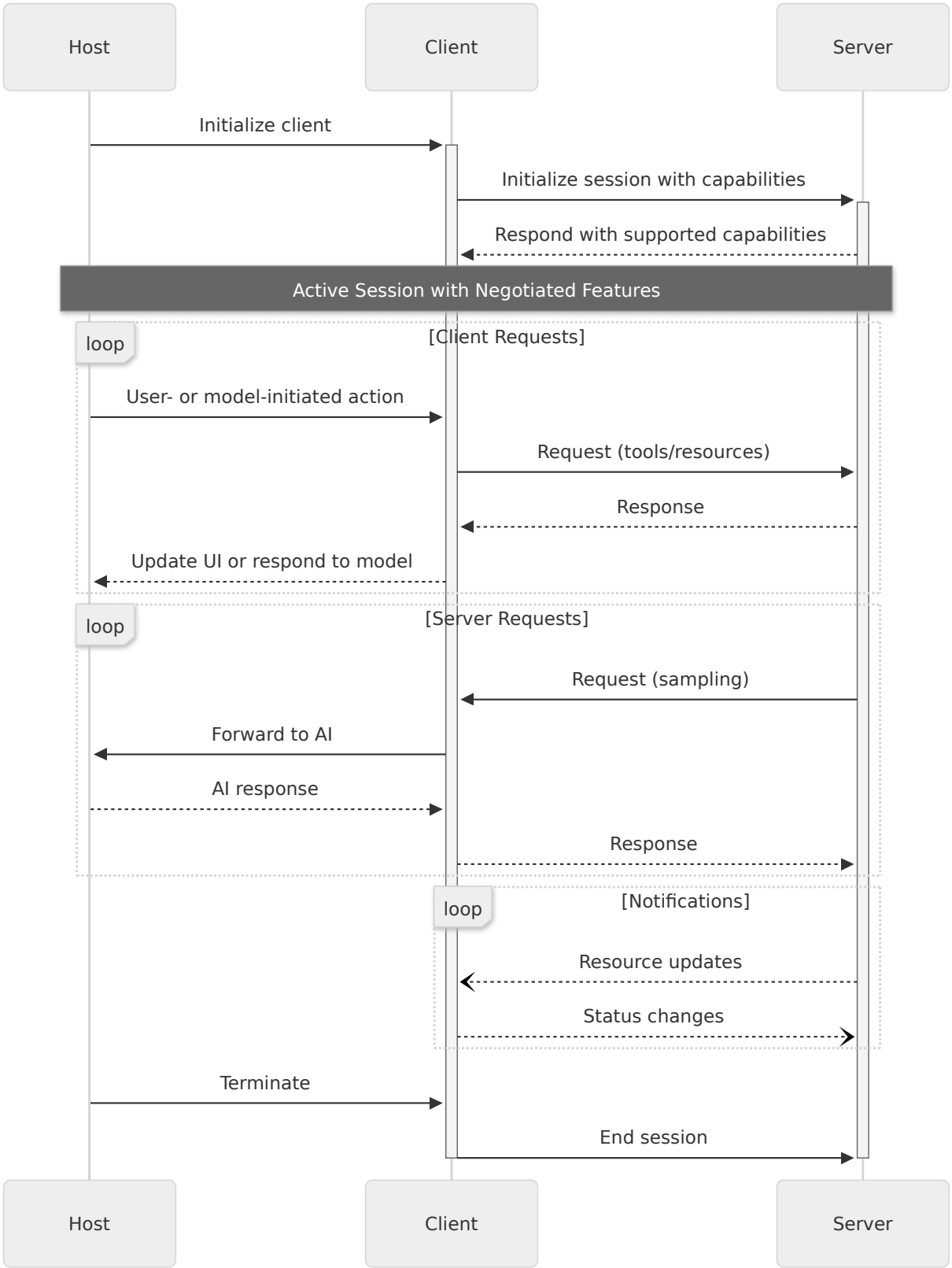
4. **Features can be added to servers and clients progressively**

    - 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 either the result or error of the operation.

Result Responses

Result responses are sent when the operation completes successfully.

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

- Result responses **MUST** include the same ID as the request they correspond to.
- Result responses **MUST** include a `result` field.
- The `result` **MAY** follow any JSON object structure.

Error Responses

Error responses are sent when the operation fails or encounters an error.

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

- Error responses **MUST** include the same ID as the request they correspond to (except in error cases where the ID could not be read due a malformed request).
- Error responses **MUST** include an `error` field with a `code` and `message`.
- 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.

JSON Schema Usage

The Model Context Protocol uses JSON Schema for validation throughout the protocol. This section clarifies how JSON Schema should be used within MCP messages.

Schema Dialect

MCP supports JSON Schema with the following rules:

1. **Default dialect**: When a schema does not include a `$schema` field, it defaults to JSON Schema 2020-12
2. **Explicit dialect**: Schemas MAY include a `$schema` field to specify a different dialect
3. **Supported dialects**: Implementations MUST support at least 2020-12 and SHOULD document which additional dialects they support
4. **Recommendation**: Implementors are RECOMMENDED to use JSON Schema 2020-12.

Example Usage

Default dialect (2020-12):

```
{
  "type": "object",
  "properties": {
    "name": { "type": "string" },
    "age": { "type": "integer", "minimum": 0 }
  },
  "required": ["name"]
}
```

Explicit dialect (draft-07):

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "object",
  "properties": {
    "name": { "type": "string" },
    "age": { "type": "integer", "minimum": 0 }
  },
  "required": ["name"]
}
```

Implementation Requirements

- Clients and servers **MUST** support JSON Schema 2020-12 for schemas without an explicit `$schema` field
- Clients and servers **MUST** validate schemas according to their declared or default dialect. They **MUST** handle unsupported dialects gracefully by returning an appropriate error indicating the dialect is not supported.
- Clients and servers **SHOULD** document which schema dialects they support

Schema Validation

- Schemas **MUST** be valid according to their declared or default dialect

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 (`-`).
  - Implementations SHOULD use reverse DNS notation (e.g., `com.example/` rather than `example.com/`).
- Any prefix where the second label is `modelcontextprotocol` or `mcp` is **reserved** for MCP use.
  - For example: `io.modelcontextprotocol/`, `dev.mcp/`, `org.modelcontextprotocol.api/`, and `com.mcp.tools/` are all reserved.
  - However, `com.example.mcp/` is NOT reserved, as the second label is `example`.

Name:

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

`icons`

The `icons` property provides a standardized way for servers to expose visual identifiers for their resources, tools, prompts, and implementations. Icons enhance user interfaces by providing visual context and improving the discoverability of available functionality.

Icons are represented as an array of `Icon` objects, where each icon includes:

- `src`: A URI pointing to the icon resource (required). This can be:
  - An HTTP/HTTPS URL pointing to an image file
  - A data URI with base64-encoded image data
- `mimeType`: Optional MIME type if the server's type is missing or generic
- `sizes`: Optional array of size specifications (e.g., `["48x48"]`, `["any"]` for scalable formats like SVG, or `["48x48", "96x96"]` for multiple sizes)
- `theme`: Optional theme preference (`light` or `dark`) for the icon background

Required MIME type support:

Clients that support rendering icons **MUST** support at least the following MIME types:

- `image/png` - PNG images (safe, universal compatibility)
- `image/jpeg` (and `image/jpg`) - JPEG images (safe, universal compatibility)

Clients that support rendering icons **SHOULD** also support:

- `image/svg+xml` - SVG images (scalable but requires security precautions as noted below)
- `image/webp` - WebP images (modern, efficient format)

Security considerations:

Consumers of icon metadata **MUST** take appropriate security precautions when handling icons to prevent compromise:

- Treat icon metadata and icon bytes as untrusted inputs and defend against network, privacy, and parsing risks.
- Ensure that the icon URI is either a HTTPS or `data:` URI. Clients **MUST** reject icon URIs that use unsafe schemes and redirects, such as `javascript:`, `file:`, `ftp:`, `ws:`, or local app URI schemes.
    - Disallow scheme changes and redirects to hosts on different origins.
- Be resilient against resource exhaustion attacks stemming from oversized images, large dimensions, or excessive frames (e.g., in GIFs).
    - Consumers **MAY** set limits for image and content size.
- Fetch icons without credentials. Do not send cookies, `Authorization` headers, or client credentials.
- Verify that icon URIs are from the same origin as the server. This minimizes the risk of exposing data or tracking information to third-parties.
- Exercise caution when fetching and rendering icons as the payload **MAY** contain executable content (e.g., SVG with embedded JavaScript or extended capabilities).
    - Consumers **MAY** choose to disallow specific file types or otherwise sanitize icon files before rendering.
- Validate MIME types and file contents before rendering. Treat the MIME type information as advisory. Detect content type via magic bytes; reject on mismatch or unknown types.
    - Maintain a strict allowlist of image types.

Usage:

Icons can be attached to:

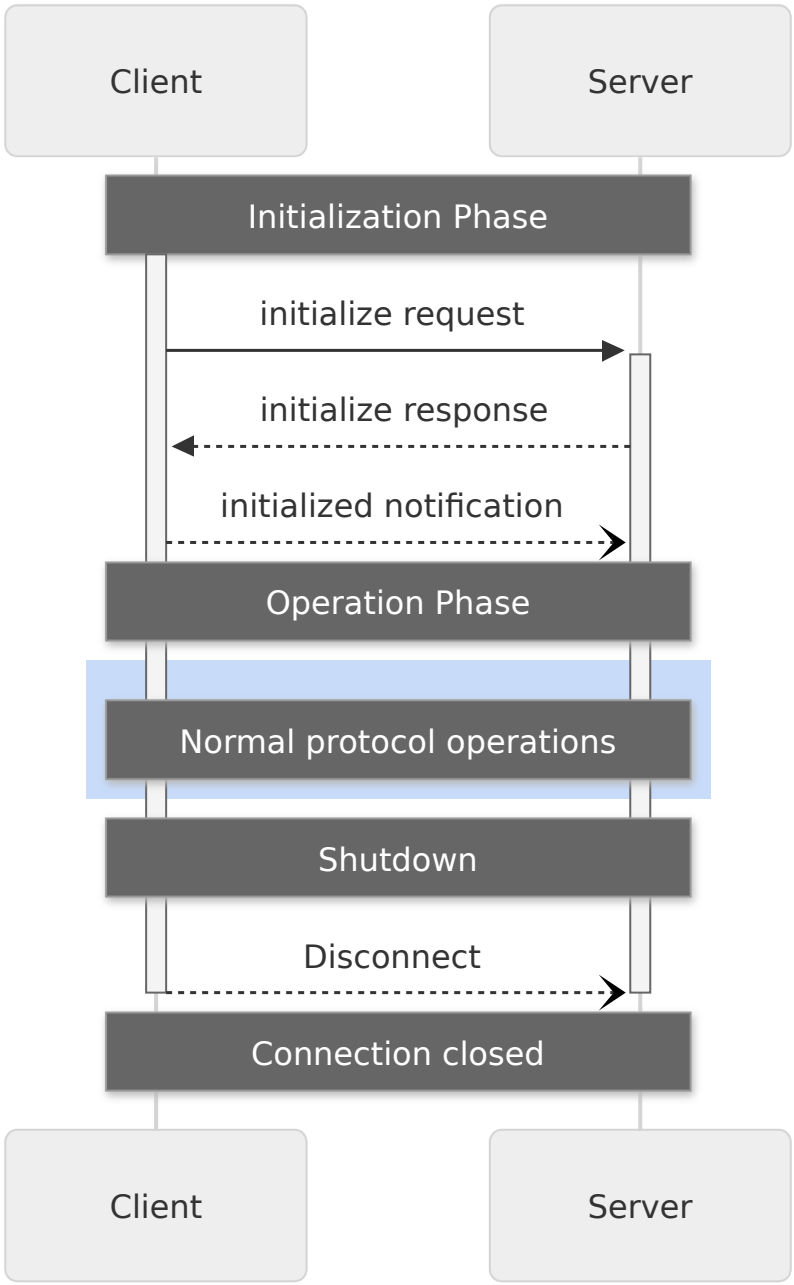
- `Implementation`: Visual identifier for the MCP server/client implementation
- `Tool`: Visual representation of the tool's functionality
- `Prompt`: Icon to display alongside prompt templates
- `Resource`: Visual indicator for different resource types

Multiple icons can be provided to support different display contexts and resolutions. Clients should select the most appropriate icon based on their UI requirements.

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": "2025-11-25",
    "capabilities": {
      "roots": {
        "listChanged": true
      },
      "sampling": {},
      "elicitation": {
        "form": {},
        "url": {}
      },
      "tasks": {
        "requests": {
          "elicitation": {
            "create": {}
          },
          "sampling": {
            "createMessage": {}
          }
        }
      }
    },
    "clientInfo": {
      "name": "ExampleClient",
      "title": "Example Client Display Name",
      "version": "1.0.0",
      "description": "An example MCP client application",
      "icons": [
        {
          "src": "https://example.com/icon.png",
          "mimeType": "image/png",
          "sizes": ["48x48"]
        }
      ],
      "websiteUrl": "https://example.com"
    }
  }
}
```

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

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "protocolVersion": "2025-11-25",
    "capabilities": {
      "logging": {},
      "prompts": {
        "listChanged": true
      },
      "resources": {
        "subscribe": true,
        "listChanged": true
      },
      "tools": {
        "listChanged": true
      },
      "tasks": {
        "list": {},
        "cancel": {},
        "requests": {
          "tools": {
            "call": {}
          }
        }
      }
    },
    "serverInfo": {
      "name": "ExampleServer",
      "title": "Example Server Display Name",
      "version": "1.0.0",
      "description": "An example MCP server providing tools and resources",
      "icons": [
        {
          "src": "https://example.com/server-icon.svg",
          "mimeType": "image/svg+xml",
          "sizes": ["any"]
        }
      ],
      "websiteUrl": "https://example.com/server"
    },
    "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	`tasks`	Support for task-augmented client 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	`tasks`	Support for task-augmented server requests
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

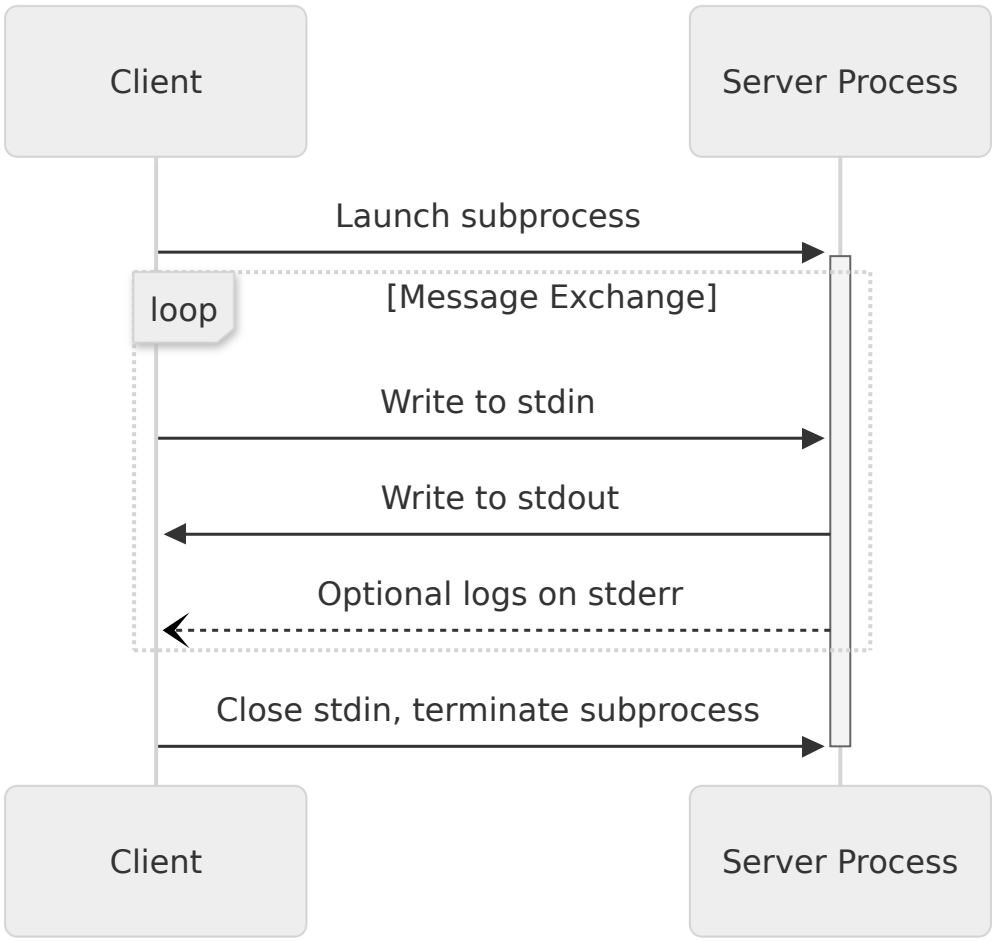
Clients **SHOULD** support stdio whenever possible.

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 any logging purposes including informational, debug, and error messages.
- The client **MAY** capture, forward, or ignore the server's `stderr` output and **SHOULD NOT** assume `stderr` output indicates error conditions.
- 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
    - If the `Origin` header is present and invalid, servers **MUST** respond with HTTP 403 Forbidden. The HTTP response body **MAY** comprise a JSON-RPC *error response* that has no `id`
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 server **SHOULD** immediately send an SSE event consisting of an event ID and an empty `data` field in order to prime the client to reconnect (using that event ID as `Last-Event-ID`).
    - After the server has sent an SSE event with an event ID to the client, the server **MAY** close the *connection* (without terminating the *SSE stream*) at any time in order to avoid holding a long-lived connection. The client **SHOULD** then "poll" the SSE stream by attempting to reconnect.
    - If the server does close the *connection* prior to terminating the *SSE stream*, it **SHOULD** send an SSE event with a standard `retry` field before closing the connection. The client **MUST** respect the `retry` field, waiting the given number of milliseconds before attempting to reconnect.
    - The SSE stream **SHOULD** eventually include a 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 **MAY** terminate the SSE stream if the session expires.
    - After the JSON-RPC *response* has been sent, the server **SHOULD** terminate 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.
   - If the server closes the *connection* without terminating the *stream*, it **SHOULD** follow the same polling behavior as described for POST requests: sending a `retry` field and allowing the client to reconnect.
   - 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.
   - Event IDs **SHOULD** encode sufficient information to identify the originating stream, enabling the server to correlate a `Last-Event-ID` to the correct stream.
2. If the client wishes to resume after a disconnection (whether due to network failure or server-initiated closure), 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.
    - This mechanism applies regardless of how the original stream was initiated (via POST or GET). Resumption is always via HTTP GET with `Last-Event-ID`.

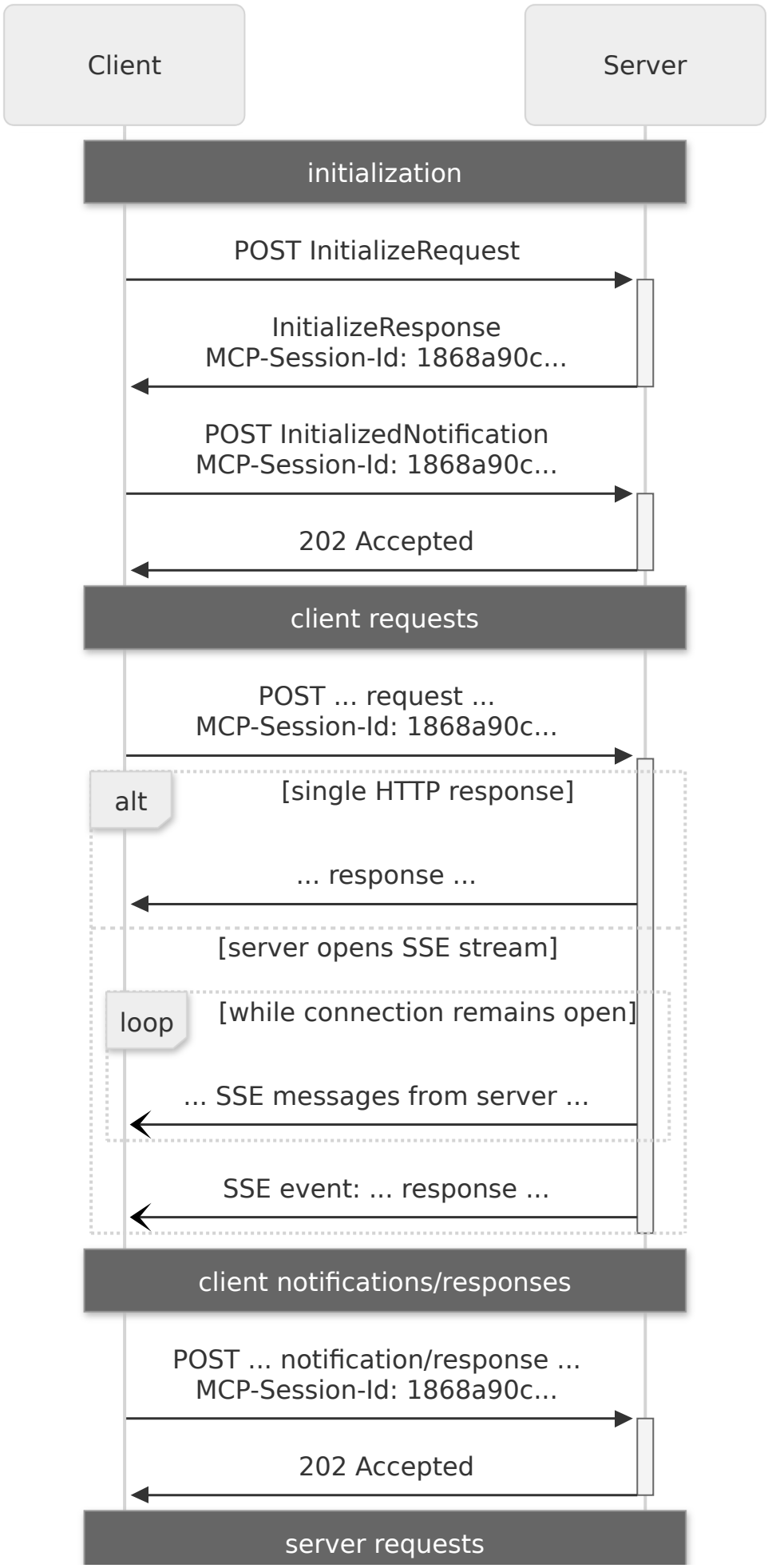
In other words, these event IDs should be assigned by servers on a *per-stream* basis, to act as a cursor within that particular stream.

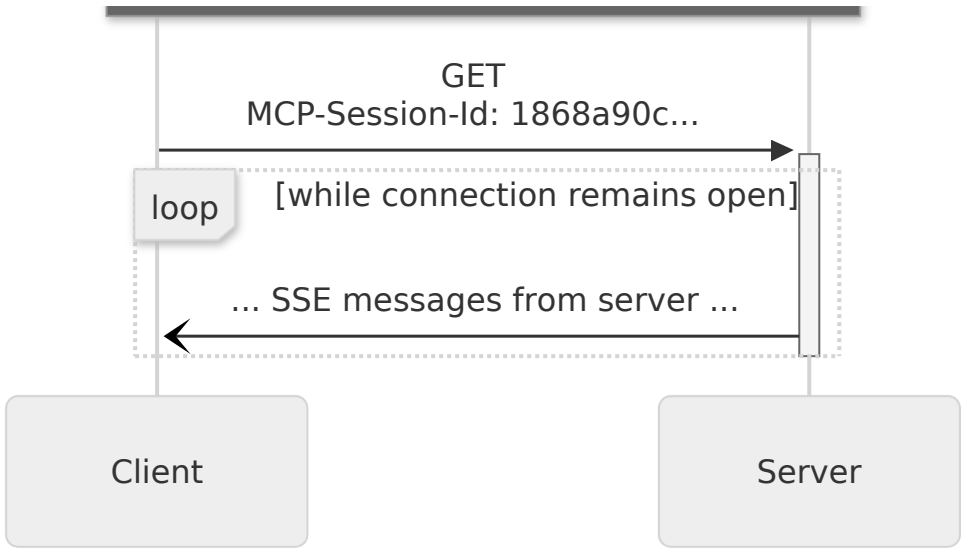
Session Management

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).
    - The client **MUST** handle the session ID in a secure manner, see Session Hijacking mitigations for more details.
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


Client
Server
initialization
POST InitializeRequest
InitializeResponse
MCP-Session-Id: 1868a90c...
POST InitializedNotification
MCP-Session-Id: 1868a90c...
202 Accepted
client requests
POST ... request ...
MCP-Session-Id: 1868a90c...
alt
[single HTTP response]
... response ...
[server opens SSE stream]
loop
[while connection remains open]
... SSE messages from server ...
SSE event: ... response ...
client notifications/responses
POST ... notification/response ...
MCP-Session-Id: 1868a90c...
202 Accepted
server requests



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: 2025-11-25`

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 the following HTTP status codes "400 Bad Request", "404 Not Found" or "405 Method Not Allowed":
        - 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)
- OAuth Client ID Metadata Documents (draft-ietf-oauth-client-id-metadata-document-00)

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 OAuth Client ID Metadata Documents (draft-ietf-oauth-client-id-metadata-document-00).
3. Authorization servers and MCP clients **MAY** support the OAuth 2.0 Dynamic Client Registration Protocol (RFC7591).
4. 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.
5. MCP authorization servers **MUST** provide at least one of the following discovery mechanisms:
   - OAuth 2.0 Authorization Server Metadata (RFC8414)
   - OpenID Connect Discovery 1.0

   MCP clients **MUST** support both discovery mechanisms to obtain the information required to interact with the authorization server.

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".

Protected Resource Metadata Discovery Requirements

MCP servers **MUST** implement one of the following discovery mechanisms to provide authorization server location information to MCP clients:

1. **WWW-Authenticate Header**: Include the resource metadata URL in the `WWW-Authenticate` HTTP header under `resource_metadata` when returning `401 Unauthorized` responses, as described in RFC9728 Section 5.1.
2. **Well-Known URI**: Serve metadata at a well-known URI as specified in RFC9728. This can be either:
    - At the path of the server's MCP endpoint: `https://example.com/public/mcp` could host metadata at `https://example.com/.well-known/oauth-protected-resource/public/mcp`
    - At the root: `https://example.com/.well-known/oauth-protected-resource`

MCP clients **MUST** support both discovery mechanisms and use the resource metadata URL from the parsed `WWW-Authenticate` headers when present; otherwise, they **MUST** fall back to constructing and requesting the well-known URIs in the order listed above.

MCP servers **SHOULD** include a `scope` parameter in the `WWW-Authenticate` header as defined in RFC 6750 Section 3 to indicate the scopes required for accessing the resource. This provides clients with immediate guidance on the appropriate scopes to request during authorization, following the principle of least privilege and preventing clients from requesting excessive permissions.

The scopes included in the `WWW-Authenticate` challenge **MAY** match `scopes_supported`, be a subset or superset of it, or an alternative collection that is neither a strict subset nor superset. Clients **MUST NOT** assume any particular set relationship between the challenged scope set and `scopes_supported`. Clients **MUST** treat the scopes provided in the challenge as authoritative for satisfying the current request. Servers **SHOULD** strive for consistency in how they construct scope sets but they are not required to surface every dynamically issued scope through `scopes_supported`.

Example 401 response with scope guidance:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer resource_metadata="https://mcp.example.com/.well-
known/oauth-protected-resource",
                        scope="files:read"
```

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

If the `scope` parameter is absent, clients **SHOULD** apply the fallback behavior defined in the Scope Selection Strategy section.

Authorization Server Metadata Discovery

To handle different issuer URL formats and ensure interoperability with both OAuth 2.0 Authorization Server Metadata and OpenID Connect Discovery 1.0 specifications, MCP clients **MUST** attempt multiple well-known endpoints when discovering authorization server metadata.

The discovery approach is based on RFC8414 Section 3.1 "Authorization Server Metadata Request" for OAuth 2.0 Authorization Server Metadata discovery and RFC8414 Section 5 "Compatibility Notes" for OpenID Connect Discovery 1.0 interoperability.

For issuer URLs with path components (e.g., `https://auth.example.com/tenant1`), clients **MUST** try endpoints in the following priority order:

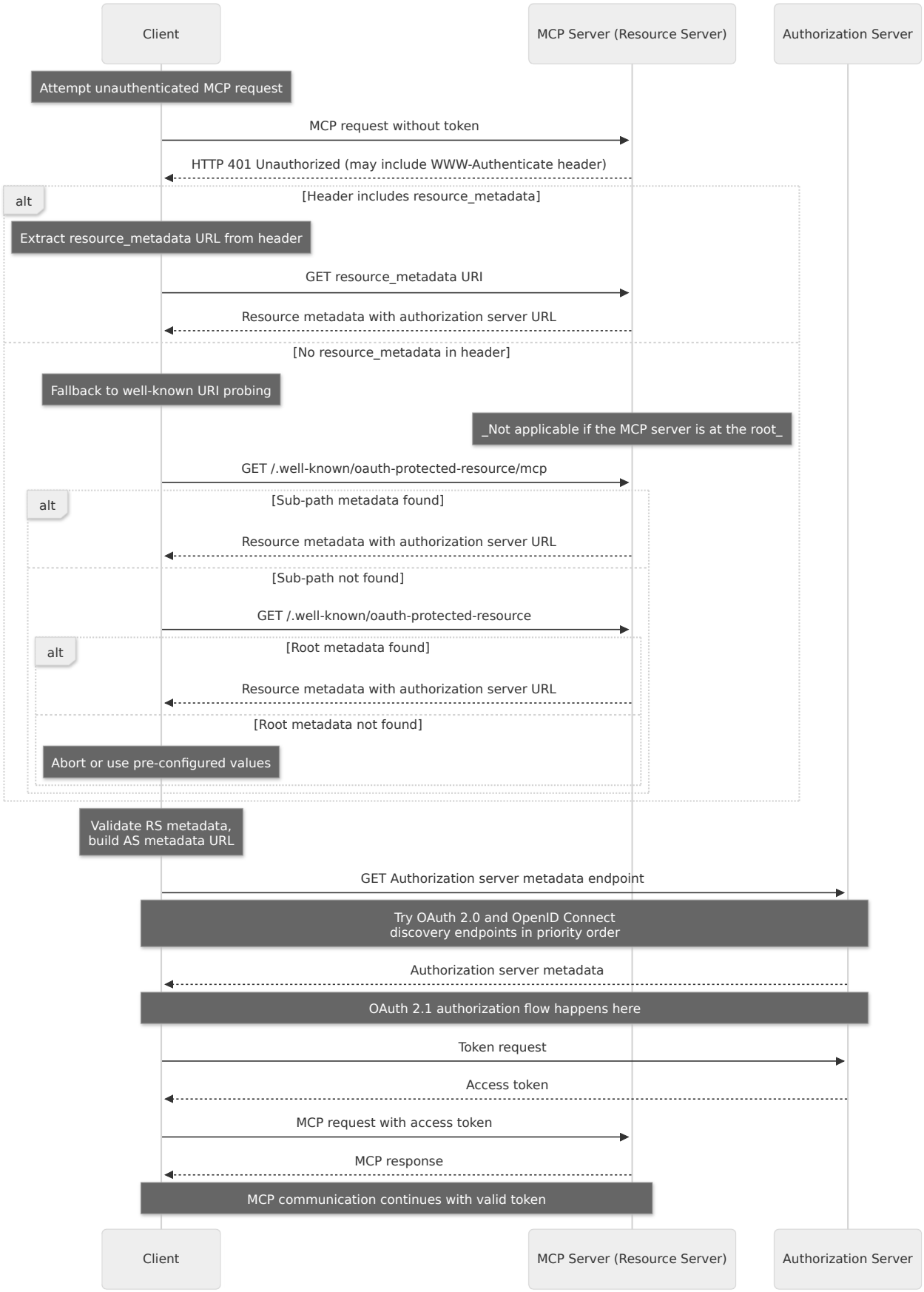
1. OAuth 2.0 Authorization Server Metadata with path insertion: `https://auth.example.com/.well-known/oauth-authorization-server/tenant1`
2. OpenID Connect Discovery 1.0 with path insertion: `https://auth.example.com/.well-known/openid-configuration/tenant1`
3. OpenID Connect Discovery 1.0 path appending: `https://auth.example.com/tenant1/.well-known/openid-configuration`

For issuer URLs without path components (e.g., `https://auth.example.com`), clients **MUST** try:

1. OAuth 2.0 Authorization Server Metadata: `https://auth.example.com/.well-known/oauth-authorization-server`
2. OpenID Connect Discovery 1.0: `https://auth.example.com/.well-known/openid-configuration`

Authorization Server Discovery Sequence Diagram

The following diagram outlines an example flow:


Client
MCP Server (Resource Server)
Authorization Server
Attempt unauthenticated MCP request
MCP request without token
HTTP 401 Unauthorized (may include WWW-Authenticate header)
alt
[Header includes resource_metadata]
Extract resource_metadata URL from header
GET resource_metadata URI
Resource metadata with authorization server URL
[No resource_metadata in header]
Fallback to well-known URI probing
Not applicable if the MCP server is at the root
GET /.well-known/oauth-protected-resource/mcp
alt
[Sub-path metadata found]
Resource metadata with authorization server URL
[Sub-path not found]
GET /.well-known/oauth-protected-resource
alt
[Root metadata found]
Resource metadata with authorization server URL
[Root metadata not found]
Abort or use pre-configured values
Validate RS metadata, build AS metadata URL
GET Authorization server metadata endpoint
Try OAuth 2.0 and OpenID Connect discovery endpoints in priority order
Authorization server metadata
OAuth 2.1 authorization flow happens here
Token request
Access token
MCP request with access token
MCP response
MCP communication continues with valid token
Client
MCP Server (Resource Server)
Authorization Server

Client Registration Approaches

MCP supports three client registration mechanisms. Choose based on your scenario:

- **Client ID Metadata Documents**: When client and server have no prior relationship (most common)
- **Pre-registration**: When client and server have an existing relationship
- **Dynamic Client Registration**: For backwards compatibility or specific requirements

Clients supporting all options **SHOULD** follow the following priority order:

1. Use pre-registered client information for the server if the client has it available
2. Use Client ID Metadata Documents if the Authorization Server indicates if the server supports it (via `client_id_metadata_document_supported` in OAuth Authorization Server Metadata)
3. Use Dynamic Client Registration as a fallback if the Authorization Server supports it (via `registration_endpoint` in OAuth Authorization Server Metadata)
4. Prompt the user to enter the client information if no other option is available

Client ID Metadata Documents

MCP clients and authorization servers **SHOULD** support OAuth Client ID Metadata Documents as specified in OAuth Client ID Metadata Document. This approach enables clients to use HTTPS URLs as client identifiers, where the URL points to a JSON document containing client metadata. This addresses the common MCP scenario where servers and clients have no pre-existing relationship.

Implementation Requirements

MCP implementations supporting Client ID Metadata Documents **MUST** follow the requirements specified in OAuth Client ID Metadata Document. Key requirements include:

For MCP Clients:

- Clients **MUST** host their metadata document at an HTTPS URL following RFC requirements
- The `client_id` URL **MUST** use the "https" scheme and contain a path component, e.g. `https://example.com/client.json`
- The metadata document **MUST** include at least the following properties: `client_id`, `client_name`, `redirect_uris`
- Clients **MUST** ensure the `client_id` value in the metadata matches the document URL exactly
- Clients **MAY** use `private_key_jwt` for client authentication (e.g., for requests to the token endpoint) with appropriate JWKS configuration as described in Section 6.2 of Client ID Metadata Document

For Authorization Servers:

- **SHOULD** fetch metadata documents when encountering URL-formatted client_ids
- **MUST** validate that the fetched document's `client_id` matches the URL exactly
- **SHOULD** cache metadata respecting HTTP cache headers
- **MUST** validate redirect URIs presented in an authorization request against those in the metadata document
- **MUST** validate the document structure is valid JSON and contains required fields

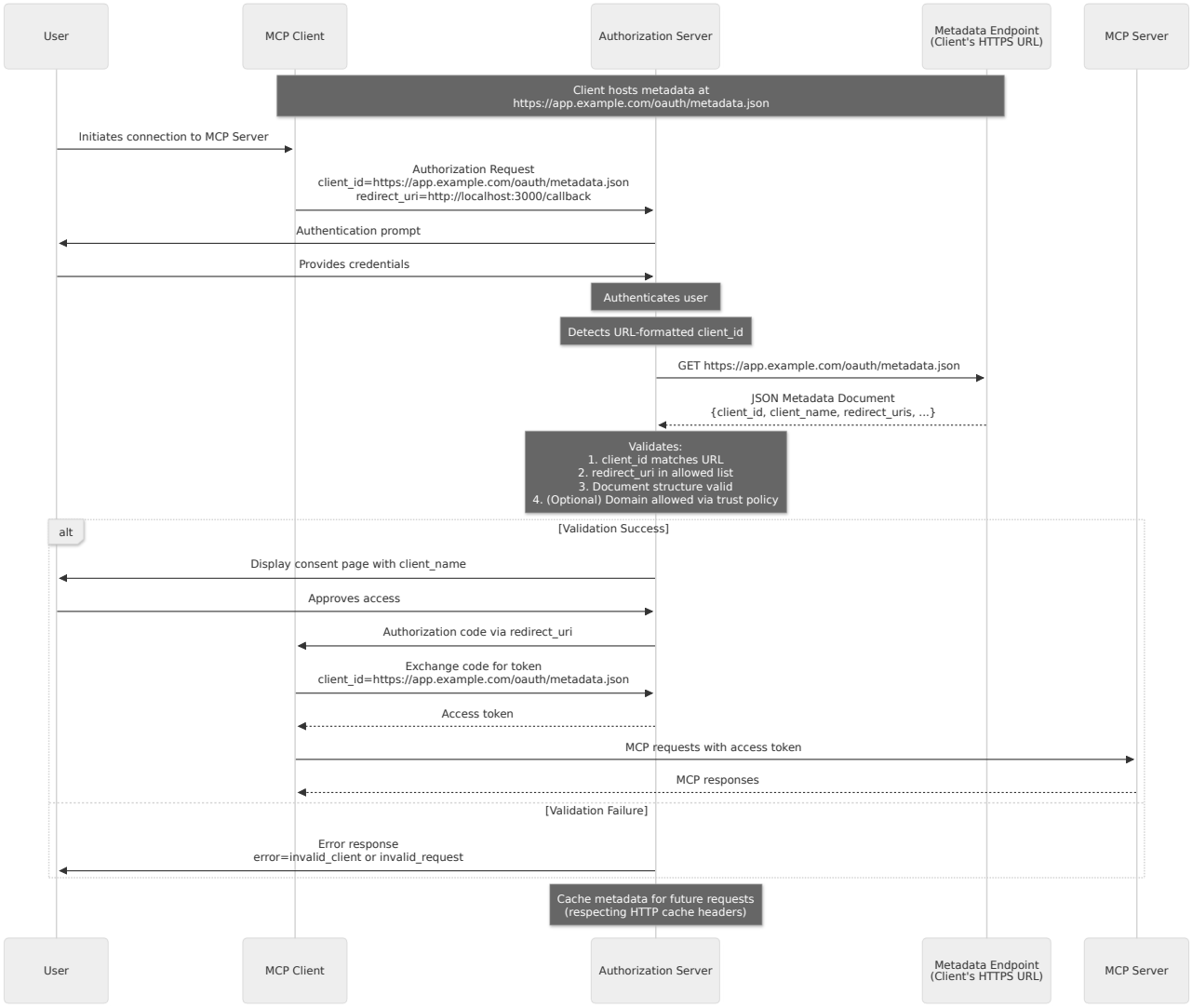
- **SHOULD** follow the security considerations in Section 6 of Client ID Metadata Document

Example Metadata Document

```
{
  "client_id": "https://app.example.com/oauth/client-metadata.json",
  "client_name": "Example MCP Client",
  "client_uri": "https://app.example.com",
  "logo_uri": "https://app.example.com/logo.png",
  "redirect_uris": [
    "http://127.0.0.1:3000/callback",
    "http://localhost:3000/callback"
  ],
  "grant_types": ["authorization_code"],
  "response_types": ["code"],
  "token_endpoint_auth_method": "none"
}
```

Client ID Metadata Documents Flow

The following diagram illustrates the complete flow when using Client ID Metadata Documents:



Discovery

Authorization servers advertise that they support clients using Client ID Metadata Documents by including the following property in their OAuth Authorization Server metadata:

```
{
  "client_id_metadata_document_supported": true
}
```

MCP clients **SHOULD** check for this capability and **MAY** fall back to Dynamic Client Registration or pre-registration if unavailable.

Preregistration

MCP clients **SHOULD** support an option for static client credentials such as those supplied by a preregistration flow. This could be:

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).

Dynamic Client Registration

MCP clients and authorization servers **MAY** support the OAuth 2.0 Dynamic Client Registration Protocol RFC7591 to allow MCP clients to obtain OAuth client IDs without user interaction. This option is included for backwards compatibility with earlier versions of the MCP authorization spec.

Scope Selection Strategy

When implementing authorization flows, MCP clients **SHOULD** follow the principle of least privilege by requesting only the scopes necessary for their intended operations. During the initial authorization handshake, MCP clients **SHOULD** follow this priority order for scope selection:

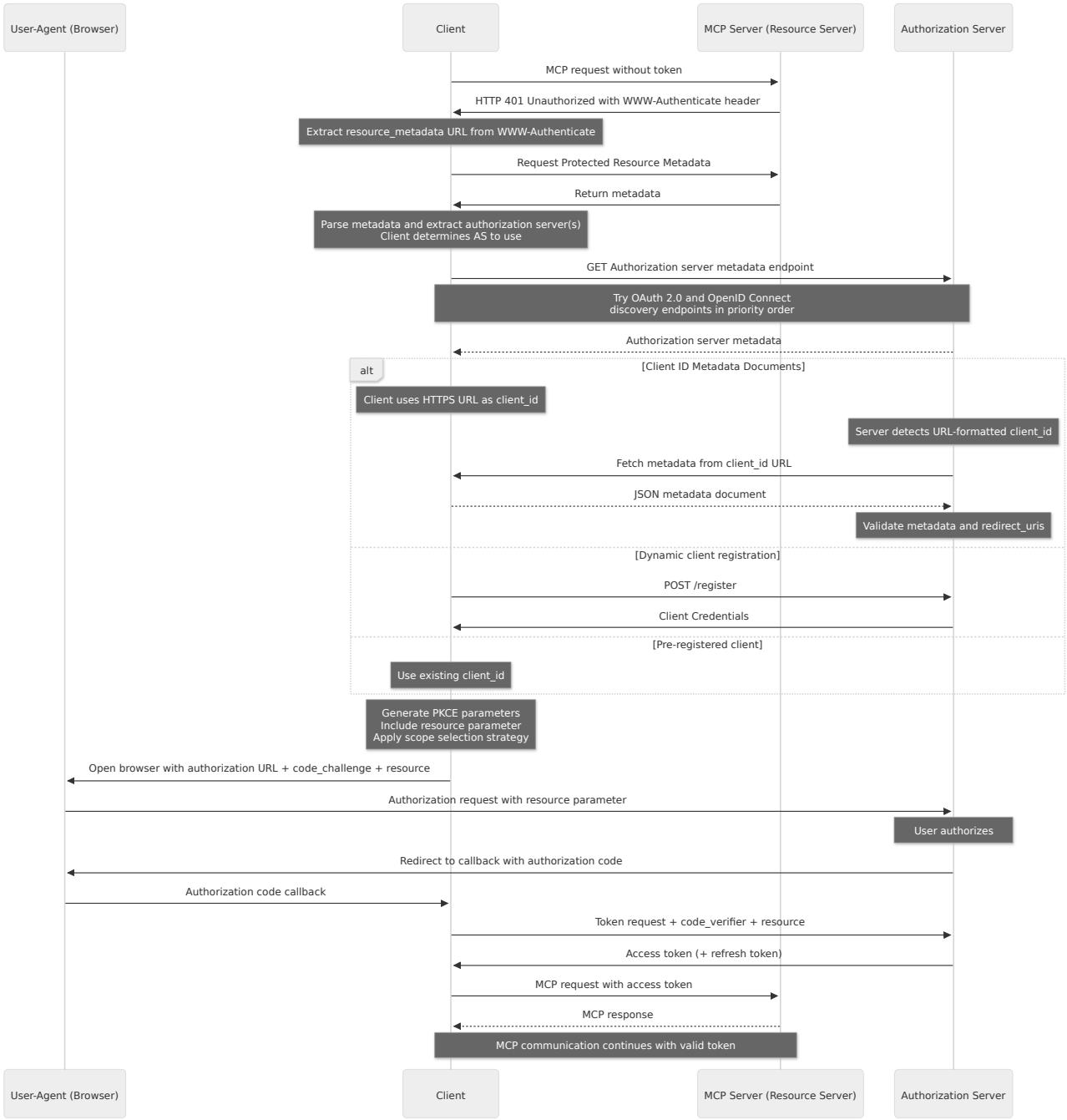
1. **Use `scope` parameter** from the initial `WWW-Authenticate` header in the 401 response, if provided
2. **If `scope` is not available**, use all scopes defined in `scopes_supported` from the Protected Resource Metadata document, omitting the `scope` parameter if `scopes_supported` is undefined.

This approach accommodates the general-purpose nature of MCP clients, which typically lack domain-specific knowledge to make informed decisions about individual scope selection. Requesting all available scopes allows the authorization server and end-user to determine appropriate permissions during the consent process.

This approach minimizes user friction while following the principle of least privilege. The `scopes_supported` field is intended to represent the minimal set of scopes necessary for basic functionality (see Scope Minimization), with additional scopes requested incrementally through the step-up authorization flow steps described in the Scope Challenge Handling section.

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.

MCP 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

Scope Challenge Handling

This section covers handling insufficient scope errors during runtime operations when a client already has a token but needs additional permissions. This follows the error handling patterns defined in OAuth 2.1 Section 5 and leverages the metadata fields from RFC 9728 (OAuth 2.0 Protected Resource Metadata).

Runtime Insufficient Scope Errors

When a client makes a request with an access token with insufficient scope during runtime operations, the server **SHOULD** respond with:

- `HTTP 403 Forbidden` status code (per RFC 6750 Section 3.1)
- `WWW-Authenticate` header with the `Bearer` scheme and additional parameters:

- `error="insufficient_scope"` - indicating the specific type of authorization failure
- `scope="required_scope1 required_scope2"` - specifying the minimum scopes needed for the operation
- `resource_metadata` - the URI of the Protected Resource Metadata document (for consistency with 401 responses)
- `error_description` (optional) - human-readable description of the error

Server Scope Management: When responding with insufficient scope errors, servers **SHOULD** include the scopes needed to satisfy the current request in the `scope` parameter.

Servers have flexibility in determining which scopes to include:

- **Minimum approach**: Include the newly-required scopes for the specific operation. Include any existing granted scopes as well, if they are required, to prevent clients from losing previously granted permissions.
- **Recommended approach**: Include both existing relevant scopes and newly required scopes to prevent clients from losing previously granted permissions
- **Extended approach**: Include existing scopes, newly required scopes, and related scopes that commonly work together

The choice depends on the server's assessment of user experience impact and authorization friction.

Servers **SHOULD** be consistent in their scope inclusion strategy to provide predictable behavior for clients.

Servers **SHOULD** consider the user experience impact when determining which scopes to include in the response, as misconfigured scopes may require frequent user interaction.

Example insufficient scope response:

```
HTTP/1.1 403 Forbidden
WWW-Authenticate: Bearer error="insufficient_scope",
                        scope="files:read files:write user:profile",
                        resource_metadata="https://mcp.example.com/.well-
known/oauth-protected-resource",
                        error_description="Additional file write permission
required"
```

Step-Up Authorization Flow

Clients will receive scope-related errors during initial authorization or at runtime (`insufficient_scope`). Clients **SHOULD** respond to these errors by requesting a new access token with an increased set of scopes via a step-up authorization flow or handle the errors in other, appropriate ways. Clients acting on behalf of a user **SHOULD** attempt the step-up authorization flow. Clients acting on their own behalf (`client_credentials` clients) **MAY** attempt the step-up authorization flow or abort the request immediately.

The flow is as follows:

1. **Parse error information** from the authorization server response or `WWW-Authenticate` header
2. **Determine required scopes** as outlined in Scope Selection Strategy.
3. **Initiate (re-)authorization** with the determined scope set
4. **Retry the original request** with the new authorization no more than a few times and treat this as a permanent authorization failure

Clients **SHOULD** implement retry limits and **SHOULD** track scope upgrade attempts to avoid repeated failures for the same resource and operation combination.

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 and **MUST** verify PKCE support before proceeding with authorization. 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.

MCP clients **MUST** use the `S256` code challenge method when technically capable, as required by OAuth 2.1 Section 4.1.1.

Since OAuth 2.1 and PKCE specifications do not define a mechanism for clients to discover PKCE support, MCP clients **MUST** rely on authorization server metadata to verify this capability:

- **OAuth 2.0 Authorization Server Metadata**: If `code_challenge_methods_supported` is absent, the authorization server does not support PKCE and MCP clients **MUST** refuse to proceed.
- **OpenID Connect Discovery 1.0**: While the OpenID Provider Metadata does not define `code_challenge_methods_supported`, this field is commonly included by OpenID providers. MCP clients **MUST** verify the presence of `code_challenge_methods_supported` in the provider metadata response. If the field is absent, MCP clients **MUST** refuse to proceed.

Authorization servers providing OpenID Connect Discovery 1.0 **MUST** include `code_challenge_methods_supported` in their metadata to ensure MCP compatibility.

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.

Client ID Metadata Document Security

When implementing Client ID Metadata Documents, authorization servers **MUST** consider the security implications detailed in OAuth Client ID Metadata Document, Section 6. Key considerations

include:

Authorization Server Abuse Protection

The authorization server takes a URL as input from an unknown client and fetches that URL. A malicious client could use this to trigger the authorization server to make requests to arbitrary URLs, such as requests to private administration endpoints the authorization server has access to.

Authorization servers fetching metadata documents **SHOULD** consider Server-Side Request Forgery (SSRF) risks, as described in OAuth Client ID Metadata Document: Server Side Request Forgery (SSRF) Attacks.

Localhost Redirect URI Risks

Client ID Metadata Documents cannot prevent `localhost` URL impersonation by themselves. An attacker can claim to be any client by:

1. Providing the legitimate client's metadata URL as their `client_id`
2. Binding to any `localhost` port, and providing that address as the redirect_uri
3. Receiving the authorization code via the redirect when the user approves

The server will see the legitimate client's metadata document and the user will see the legitimate client's name, making attack detection difficult.

Authorization servers:

- **SHOULD** display additional warnings for `localhost`-only redirect URIs
- **MAY** require additional attestation mechanisms for enhanced security
- **MUST** clearly display the redirect URI hostname during authorization

Trust Policies

Authorization servers **MAY** implement domain-based trust policies:

- Allowlists for trusted domains (for protected servers)
- Accept any HTTPS `client_id` (for open servers)
- Reputation checks for unknown domains
- Restrictions based on domain age or certificate validation
- Display the CIMD and other associated client hostnames prominently to prevent phishing

Servers maintain full control over their access policies.

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.

MCP Authorization Extensions

There are several authorization extensions to the core protocol that define additional authorization mechanisms. These extensions are:

- **Optional** - Implementations can choose to adopt these extensions
- **Additive** - Extensions do not modify or break core protocol functionality; they add new capabilities while preserving core protocol behavior
- **Composable** - Extensions are modular and designed to work together without conflicts, allowing implementations to adopt multiple extensions simultaneously
- **Versioned independently** - Extensions follow the core MCP versioning cycle but may adopt independent versioning as needed

A list of supported extensions can be found in the MCP Authorization Extensions repository.

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

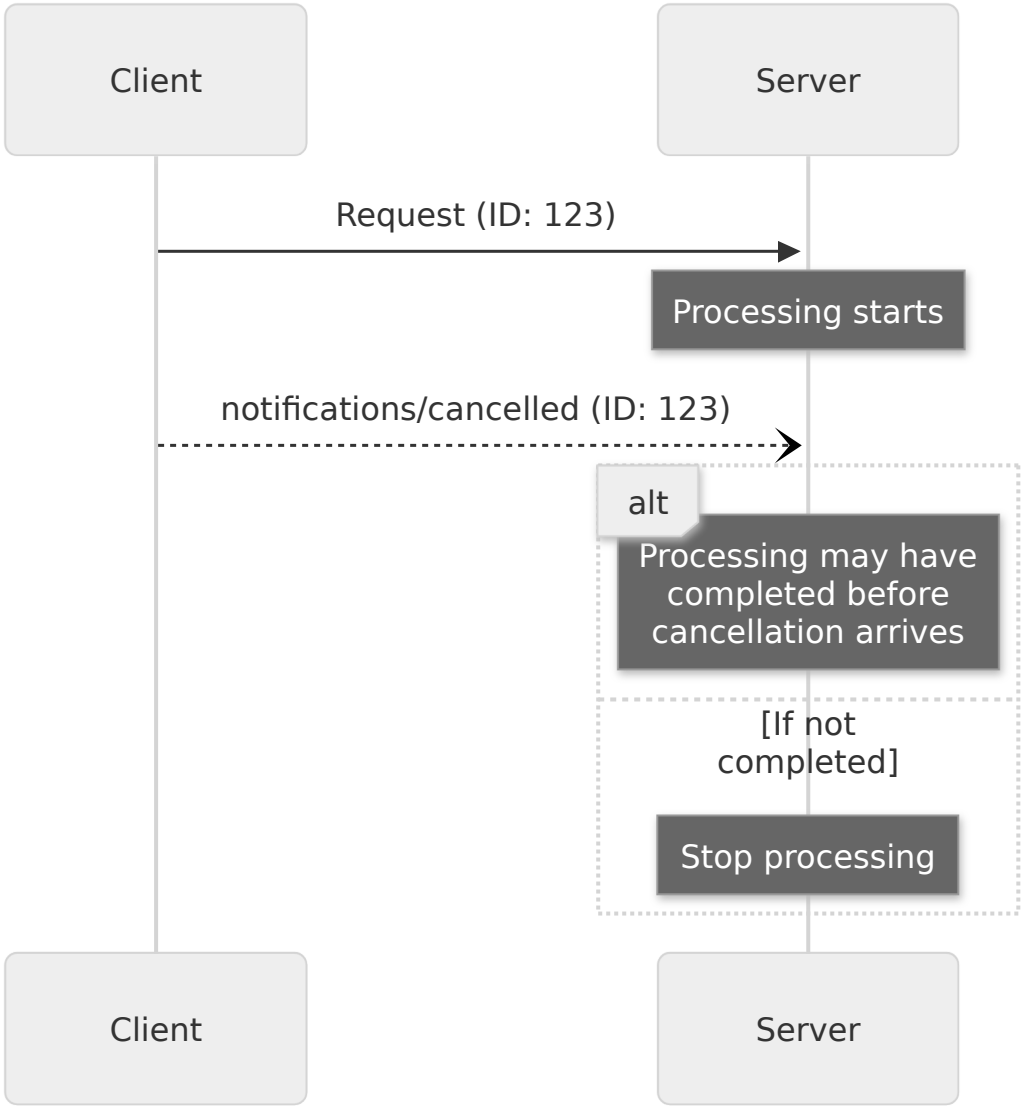
1. Cancellation notifications **MUST** only reference requests that:
    - Were previously issued in the same direction
    - Are believed to still be in-progress
2. The `initialize` request **MUST NOT** be cancelled by clients
3. For task-augmented requests, the `tasks/cancel` request **MUST** be used instead of the `notifications/cancelled` notification. Tasks have their own dedicated cancellation mechanism that returns the final task state.
4. Receivers of cancellation notifications **SHOULD**:
    - Stop processing the cancelled request
    - Free associated resources
    - Not send a response for the cancelled request
5. Receivers **MAY** ignore cancellation notifications if:
    - The referenced request is unknown
    - Processing has already completed
    - The request cannot be cancelled

6. 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"
}
```

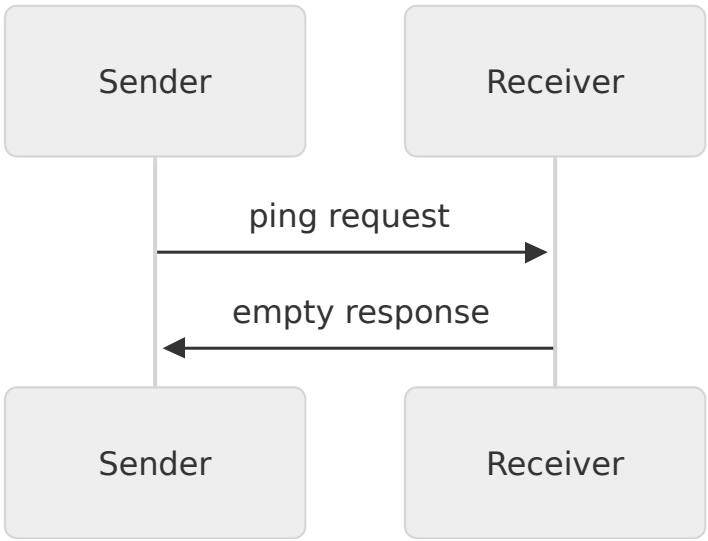
Behavior Requirements

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:

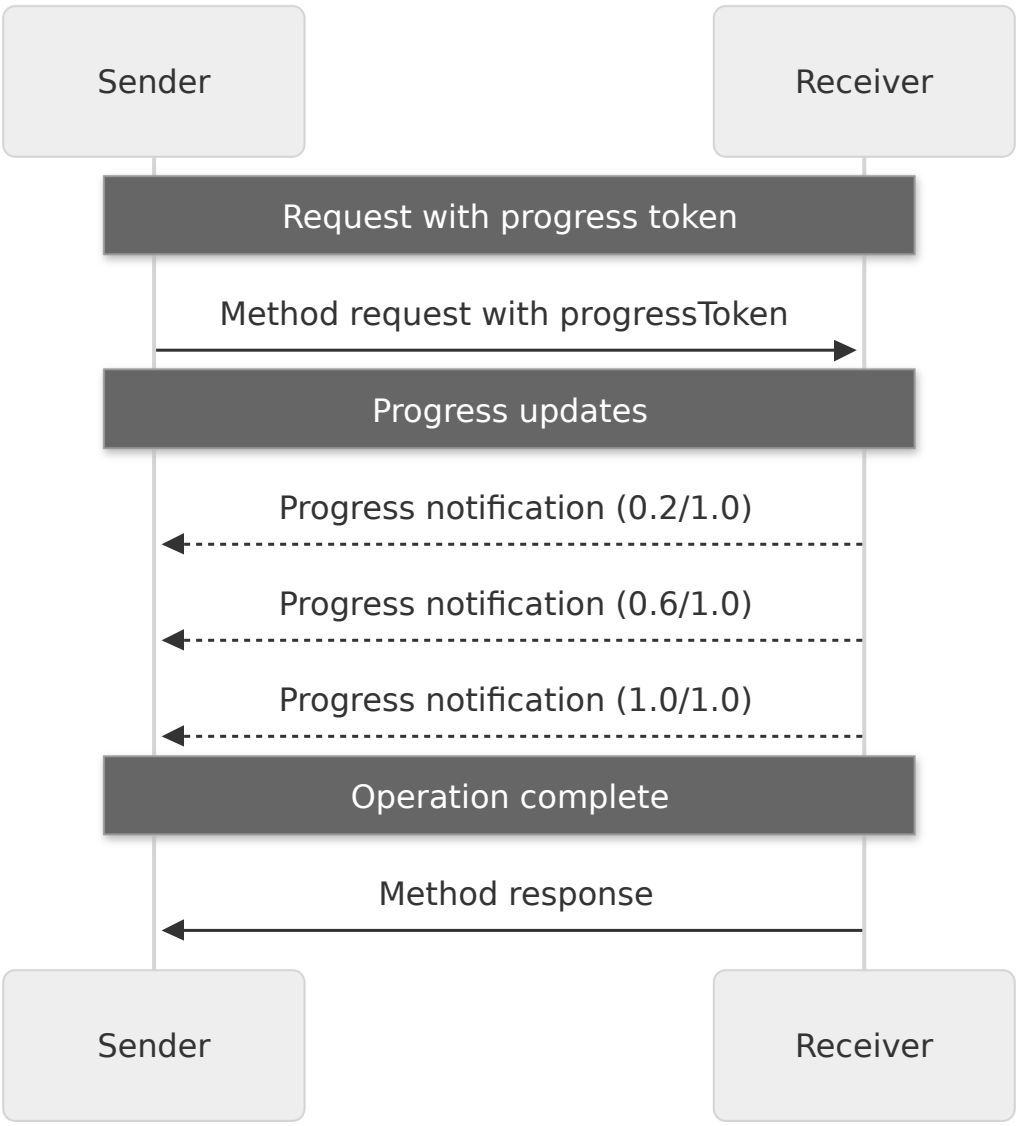
- The original progress token
- The current progress value so far
- An optional "total" value
- An optional "message" value

```
{
  "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
3. For task-augmented requests, the `progressToken` provided in the original request **MUST** continue to be used for progress notifications throughout the task's lifetime, even after the `CreateTaskResult` has been returned. The progress token remains valid and associated with the task until the task reaches a terminal status.
    - Progress notifications for tasks **MUST** use the same `progressToken` that was provided in the initial task-augmented request
    - Progress notifications for tasks **MUST** stop after the task reaches a terminal status (`completed`, `failed`, or `cancelled`)



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

4.5.4 Tasks

> **NOTE**
>
> Tasks were introduced in version 2025-11-25 of the MCP specification and are currently considered **experimental**. The design and behavior of tasks may evolve in future protocol versions.

The Model Context Protocol (MCP) allows requestors — which can be either clients or servers, depending on the direction of communication — to augment their requests with **tasks**. Tasks are durable state machines that carry information about the underlying execution state of the request they wrap, and are intended for requestor polling and deferred result retrieval. Each task is uniquely identifiable by a receiver-generated **task ID**.

Tasks are useful for representing expensive computations and batch processing requests, and integrate seamlessly with external job APIs.

Definitions

Tasks represent parties as either "requestors" or "receivers," defined as follows:

- **Requestor:** The sender of a task-augmented request. This can be the client or the server — either can create tasks.
- **Receiver:** The receiver of a task-augmented request, and the entity executing the task. This can be the client or the server — either can receive and execute tasks.

User Interaction Model

Tasks are designed to be **requestor-driven** - requestors are responsible for augmenting requests with tasks and for polling for the results of those tasks; meanwhile, receivers tightly control which requests (if any) support task-based execution and manages the lifecycles of those tasks.

This requestor-driven approach ensures deterministic response handling and enables sophisticated patterns such as dispatching concurrent requests, which only the requestor has sufficient context to orchestrate.

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

Capabilities

Servers and clients that support task-augmented requests **MUST** declare a `tasks` capability during initialization. The `tasks` capability is structured by request category, with boolean properties indicating which specific request types support task augmentation.

Server Capabilities

Servers declare if they support tasks, and if so, which server-side requests can be augmented with tasks.

Capability	Description
`tasks.list`	Server supports the `tasks/list` operation
`tasks.cancel`	Server supports the `tasks/cancel` operation
`tasks.requests.tools.call`	Server supports task-augmented `tools/call` requests

```
{
  "capabilities": {
    "tasks": {
      "list": {},
      "cancel": {},
      "requests": {
        "tools": {
          "call": {}
        }
      }
    }
  }
}
```

Client Capabilities

Clients declare if they support tasks, and if so, which client-side requests can be augmented with tasks.

Capability	Description
`tasks.list`	Client supports the `tasks/list` operation
`tasks.cancel`	Client supports the `tasks/cancel` operation
`tasks.requests.sampling.createMessage`	Client supports task-augmented `sampling/createMessage` requests
`tasks.requests.elicitation.create`	Client supports task-augmented `elicitation/create` requests

```
{
  "capabilities": {
    "tasks": {
      "list": {},
      "cancel": {},
      "requests": {
        "sampling": {
          "createMessage": {}
        },
        "elicitation": {
          "create": {}
        }
      }
    }
  }
}
```

Capability Negotiation

During the initialization phase, both parties exchange their `tasks` capabilities to establish which operations support task-based execution. Requestors **SHOULD** only augment requests with a task if the corresponding capability has been declared by the receiver.

For example, if a server's capabilities include `tasks.requests.tools.call: {}`, then clients may augment `tools/call` requests with a task. If a client's capabilities include `tasks.requests.sampling.createMessage: {}`, then servers may augment `sampling/createMessage` requests with a task.

If `capabilities.tasks` is not defined, the peer **SHOULD NOT** attempt to create tasks during requests.

The set of capabilities in `capabilities.tasks.requests` is exhaustive. If a request type is not present, it does not support task-augmentation.

`capabilities.tasks.list` controls if the `tasks/list` operation is supported by the party.

`capabilities.tasks.cancel` controls if the `tasks/cancel` operation is supported by the party.

Tool-Level Negotiation

Tool calls are given special consideration for the purpose of task augmentation. In the result of `tools/list`, tools declare support for tasks via `execution.taskSupport`, which if present can have a value of `"required"`, `"optional"`, or `"forbidden"`.

This is to be interpreted as a fine-grained layer in addition to capabilities, following these rules:

1. If a server's capabilities do not include `tasks.requests.tools.call`, then clients **MUST NOT** attempt to use task augmentation on that server's tools, regardless of the `execution.taskSupport` value.

2. If a server's capabilities include `tasks.requests.tools.call`, then clients consider the value of `execution.taskSupport`, and handle it accordingly:
    1. If `execution.taskSupport` is not present or `"forbidden"`, clients **MUST NOT** attempt to invoke the tool as a task. Servers **SHOULD** return a `-32601` (Method not found) error if a client attempts to do so. This is the default behavior.
    2. If `execution.taskSupport` is `"optional"`, clients **MAY** invoke the tool as a task or as a normal request.
    3. If `execution.taskSupport` is `"required"`, clients **MUST** invoke the tool as a task. Servers **MUST** return a `-32601` (Method not found) error if a client does not attempt to do so.

Protocol Messages

Creating Tasks

Task-augmented requests follow a two-phase response pattern that differs from normal requests:

- **Normal requests**: The server processes the request and returns the actual operation result directly.
- **Task-augmented requests**: The server accepts the request and immediately returns a `CreateTaskResult` containing task data. The actual operation result becomes available later through `tasks/result` after the task completes.

To create a task, requestors send a request with the `task` field included in the request params. Requestors **MAY** include a `ttl` value indicating the desired task lifetime duration (in milliseconds) since its creation.

Request:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "tools/call",
  "params": {
    "name": "get_weather",
    "arguments": {
      "city": "New York"
    },
    "task": {
      "ttl": 60000
    }
  }
}
```

Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "task": {
      "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840",
      "status": "working",
      "statusMessage": "The operation is now in progress.",
      "createdAt": "2025-11-25T10:30:00Z",
      "lastUpdatedAt": "2025-11-25T10:40:00Z",
      "ttl": 60000,
      "pollInterval": 5000
    }
  }
}
```

When a receiver accepts a task-augmented request, it returns a `CreateTaskResult` containing task data. The response does not include the actual operation result. The actual result (e.g., tool result for `tools/call`) becomes available only through `tasks/result` after the task completes.

NOTE

When a task is created in response to a `tools/call` request, host applications may wish to return control to the model while the task is executing. This allows the model to continue processing other requests or perform additional work while waiting for the task to complete.

To support this pattern, servers can provide an optional `io.modelcontextprotocol/model-immediate-response` key in the `_meta` field of the `CreateTaskResult`. The value of this key should be a string intended to be passed as an immediate tool result to the model. If a server does not provide this field, the host application can fall back to its own predefined message.

This guidance is non-binding and is provisional logic intended to account for the specific use case. This behavior may be formalized or modified as part of `CreateTaskResult` in future protocol versions.

Getting Tasks

NOTE

In the Streamable HTTP (SSE) transport, clients **MAY** disconnect from an SSE stream opened by the server in response to a `tasks/get` request at any time.

While this note is not prescriptive regarding the specific usage of SSE streams, all implementations **MUST** continue to comply with the existing Streamable HTTP transport specification.

Requestors poll for task completion by sending `tasks/get` requests. Requestors **SHOULD** respect the `pollInterval` provided in responses when determining polling frequency.

Requestors **SHOULD** continue polling until the task reaches a terminal status (`completed`, `failed`, or `cancelled`), or until encountering the `input_required` status. Note that invoking `tasks/result` does not imply that the requestor needs to stop polling - requestors **SHOULD** continue polling the task status via `tasks/get` if they are not actively waiting for `tasks/result` to complete.

Request:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "tasks/get",
  "params": {
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840"
  }
}
```

Response:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "result": {
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840",
    "status": "working",
    "statusMessage": "The operation is now in progress.",
    "createdAt": "2025-11-25T10:30:00Z",
    "lastUpdatedAt": "2025-11-25T10:40:00Z",
    "ttl": 30000,
    "pollInterval": 5000
  }
}
```

Retrieving Task Results

> **NOTE**
>
> In the Streamable HTTP (SSE) transport, clients **MAY** disconnect from an SSE stream opened by the server in response to a `tasks/result` request at any time.
>
> While this note is not prescriptive regarding the specific usage of SSE streams, all implementations **MUST** continue to comply with the existing Streamable HTTP transport specification.

After a task completes the operation result is retrieved via `tasks/result`. This is distinct from the initial `CreateTaskResult` response, which contains only task data. The result structure matches the original request type (e.g., `CallToolResult` for `tools/call`).

To retrieve the result of a completed task, requestors can send a `tasks/result` request:

While `tasks/result` blocks until the task reaches a terminal status, requestors can continue polling via `tasks/get` in parallel if they are not actively blocked waiting for the result, such as if their previous `tasks/result` request failed or was cancelled. This allows requestors to monitor status changes or display progress updates while the task executes, even after invoking `tasks/result`.

Request:

```
{
  "jsonrpc": "2.0",
  "id": 4,
  "method": "tasks/result",
  "params": {
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840"
  }
}
```

Response:

```
{
  "jsonrpc": "2.0",
  "id": 4,
  "result": {
    "content": [
      {
        "type": "text",
        "text": "Current weather in New York:\nTemperature: 72°F\nConditions:
Partly cloudy"
      }
    ],
    "isError": false,
    "_meta": {
      "io.modelcontextprotocol/related-task": {
        "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840"
      }
    }
  }
}
```

Task Status Notification

When a task status changes, receivers **MAY** send a `notifications/tasks/status` notification to inform the requestor of the change. This notification includes the full task state.

Notification:

```
{
  "jsonrpc": "2.0",
  "method": "notifications/tasks/status",
  "params": {
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840",
    "status": "completed",
    "createdAt": "2025-11-25T10:30:00Z",
    "lastUpdatedAt": "2025-11-25T10:50:00Z",
    "ttl": 60000,
    "pollInterval": 5000
  }
}
```

The notification includes the full `Task` object, including the updated `status` and `statusMessage` (if present). This allows requestors to access the complete task state without making an additional `tasks/get` request.

Requestors **MUST NOT** rely on receiving this notifications, as it is optional. Receivers are not required to send status notifications and may choose to only send them for certain status transitions. Requestors **SHOULD** continue to poll via `tasks/get` to ensure they receive status updates.

Listing Tasks

To retrieve a list of tasks, requestors can send a `tasks/list` request. This operation supports pagination.

Request:

```
{
  "jsonrpc": "2.0",
  "id": 5,
  "method": "tasks/list",
  "params": {
    "cursor": "optional-cursor-value"
  }
}
```

Response:

```
{
  "jsonrpc": "2.0",
  "id": 5,
  "result": {
    "tasks": [
      {
        "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840",
        "status": "working",
        "createdAt": "2025-11-25T10:30:00Z",
        "lastUpdatedAt": "2025-11-25T10:40:00Z",
        "ttl": 30000,
        "pollInterval": 5000
      },
      {
        "taskId": "abc123-def456-ghi789",
        "status": "completed",
        "createdAt": "2025-11-25T09:15:00Z",
        "lastUpdatedAt": "2025-11-25T10:40:00Z",
        "ttl": 60000
      }
    ],
    "nextCursor": "next-page-cursor"
  }
}
```

Cancelling Tasks

To explicitly cancel a task, requestors can send a `tasks/cancel` request.

Request:

```
{
  "jsonrpc": "2.0",
  "id": 6,
  "method": "tasks/cancel",
  "params": {
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840"
  }
}
```

Response:

```
{
  "jsonrpc": "2.0",
  "id": 6,
  "result": {
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840",
    "status": "cancelled",
    "statusMessage": "The task was cancelled by request.",
    "createdAt": "2025-11-25T10:30:00Z",
    "lastUpdatedAt": "2025-11-25T10:40:00Z",
    "ttl": 30000,
    "pollInterval": 5000
  }
}
```

Behavior Requirements

These requirements apply to all parties that support receiving task-augmented requests.

Task Support and Handling

1. Receivers that do not declare the task capability for a request type **MUST** process requests of that type normally, ignoring any task-augmentation metadata if present.
2. Receivers that declare the task capability for a request type **MAY** return an error for non-task-augmented requests, requiring requestors to use task augmentation.

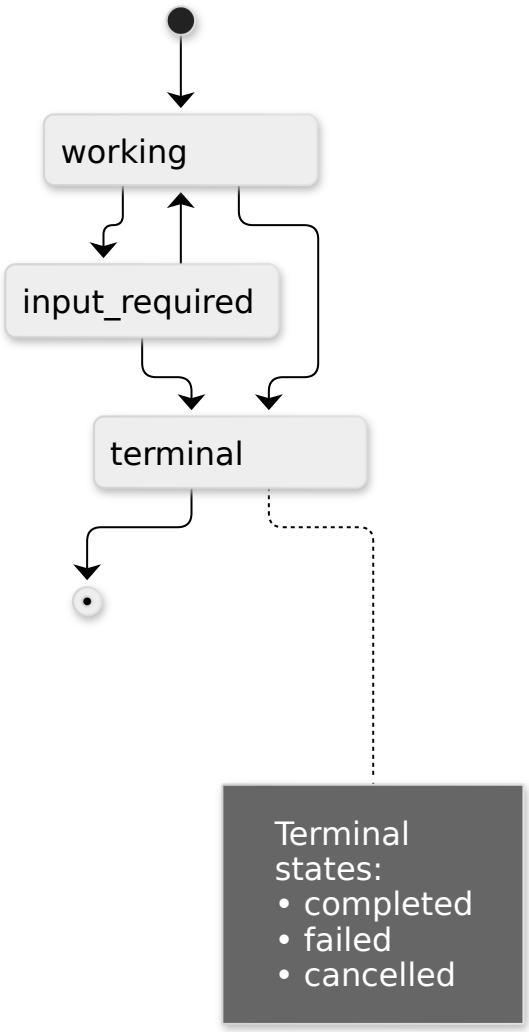
Task ID Requirements

1. Task IDs **MUST** be a string value.
2. Task IDs **MUST** be generated by the receiver when creating a task.
3. Task IDs **MUST** be unique among all tasks controlled by the receiver.

Task Status Lifecycle

1. Tasks **MUST** begin in the `working` status when created.
2. Receivers **MUST** only transition tasks through the following valid paths:
    1. From `working`: may move to `input_required`, `completed`, `failed`, or `cancelled`
    2. From `input_required`: may move to `working`, `completed`, `failed`, or `cancelled`
    3. Tasks with a `completed`, `failed`, or `cancelled` status are in a terminal state and **MUST NOT** transition to any other status

Task Status State Diagram:


working
input_required
terminal
Terminal states:
• completed
• failed
• cancelled

Input Required Status

> **NOTE**
>
> With the Streamable HTTP (SSE) transport, servers often close SSE streams after delivering a response message, which can lead to ambiguity regarding the stream used for subsequent task messages.
>
> Servers can handle this by enqueueing messages to the client to side-channel task-related messages alongside other responses.
>
> Servers have flexibility in how they manage SSE streams during task polling and result retrieval, and clients **SHOULD** expect messages to be delivered on any SSE stream, including the HTTP GET stream. One possible approach is maintaining an SSE stream on `tasks/result` (see notes on the `input_required` status). Where possible, servers **SHOULD NOT** upgrade to an SSE stream in response to a `tasks/get` request, as the client has indicated it wishes to poll for a result.
>
> While this note is not prescriptive regarding the specific usage of SSE streams, all implementations **MUST** continue to comply with the existing Streamable HTTP transport specification.

1. When the task receiver has messages for the requestor that are necessary to complete the task, the receiver **SHOULD** move the task to the `input_required` status.
2. The receiver **MUST** include the `io.modelcontextprotocol/related-task` metadata in the request to associate it with the task.
3. When the requestor encounters the `input_required` status, it **SHOULD** preemptively call `tasks/result`.
4. When the receiver receives all required input, the task **SHOULD** transition out of `input_required` status (typically back to `working`).

TTL and Resource Management

1. Receivers **MUST** include a `createdAt` ISO 8601-formatted timestamp in all task responses to indicate when the task was created.
2. Receivers **MUST** include a `lastUpdatedAt` ISO 8601-formatted timestamp in all task responses to indicate when the task was last updated.
3. Receivers **MAY** override the requested `ttl` duration.
4. Receivers **MUST** include the actual `ttl` duration (or `null` for unlimited) in `tasks/get` responses.
5. After a task's `ttl` lifetime has elapsed, receivers **MAY** delete the task and its results, regardless of the task status.
6. Receivers **MAY** include a `pollInterval` value (in milliseconds) in `tasks/get` responses to suggest polling intervals. Requestors **SHOULD** respect this value when provided.

Result Retrieval

1. Receivers that accept a task-augmented request **MUST** return a `CreateTaskResult` as the response. This result **SHOULD** be returned as soon as possible after accepting the task.

2. When a receiver receives a `tasks/result` request for a task in a terminal status (`completed`, `failed`, or `cancelled`), it **MUST** return the final result of the underlying request, whether that is a successful result or a JSON-RPC error.
3. When a receiver receives a `tasks/result` request for a task in any other non-terminal status (`working` or `input_required`), it **MUST** block the response until the task reaches a terminal status.
4. For tasks in a terminal status, receivers **MUST** return from `tasks/result` exactly what the underlying request would have returned, whether that is a successful result or a JSON-RPC error.

Associating Task-Related Messages

1. All requests, notifications, and responses related to a task **MUST** include the `io.modelcontextprotocol/related-task` key in their `_meta` field, with the value set to an object with a `taskId` matching the associated task ID.
    1. For example, an elicitation that a task-augmented tool call depends on **MUST** share the same related task ID with that tool call's task.
2. For the `tasks/get`, `tasks/result`, and `tasks/cancel` operations, the `taskId` parameter in the request **MUST** be used as the source of truth for identifying the target task. Requestors **SHOULD NOT** include `io.modelcontextprotocol/related-task` metadata in these requests, and receivers **MUST** ignore such metadata if present in favor of the RPC method parameter. Similarly, for the `tasks/get`, `tasks/list`, and `tasks/cancel` operations, receivers **SHOULD NOT** include `io.modelcontextprotocol/related-task` metadata in the result messages, as the `taskId` is already present in the response structure.

Task Notifications

1. Receivers **MAY** send `notifications/tasks/status` notifications when a task's status changes.
2. Requestors **MUST NOT** rely on receiving the `notifications/tasks/status` notification, as it is optional.
3. When sent, the `notifications/tasks/status` notification **SHOULD NOT** include the `io.modelcontextprotocol/related-task` metadata, as the task ID is already present in the notification parameters.

Task Progress Notifications

Task-augmented requests support progress notifications as defined in the progress specification. The `progressToken` provided in the initial request remains valid throughout the task lifetime.

Task Listing

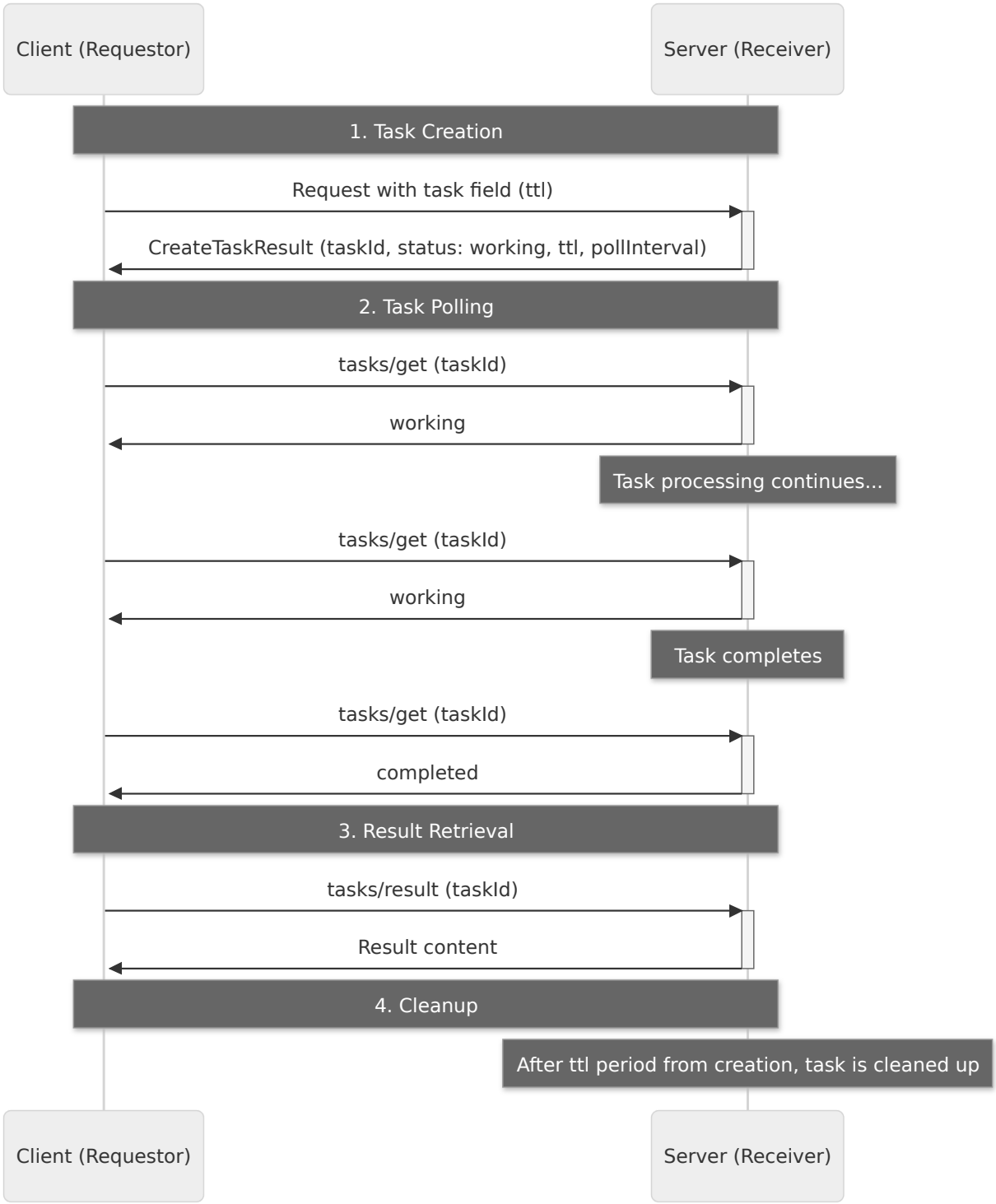
1. Receivers **SHOULD** use cursor-based pagination to limit the number of tasks returned in a single response.
2. Receivers **MUST** include a `nextCursor` in the response if more tasks are available.
3. Requestors **MUST** treat cursors as opaque tokens and not attempt to parse or modify them.
4. If a task is retrievable via `tasks/get` for a requestor, it **MUST** be retrievable via `tasks/list` for that requestor.

Task Cancellation

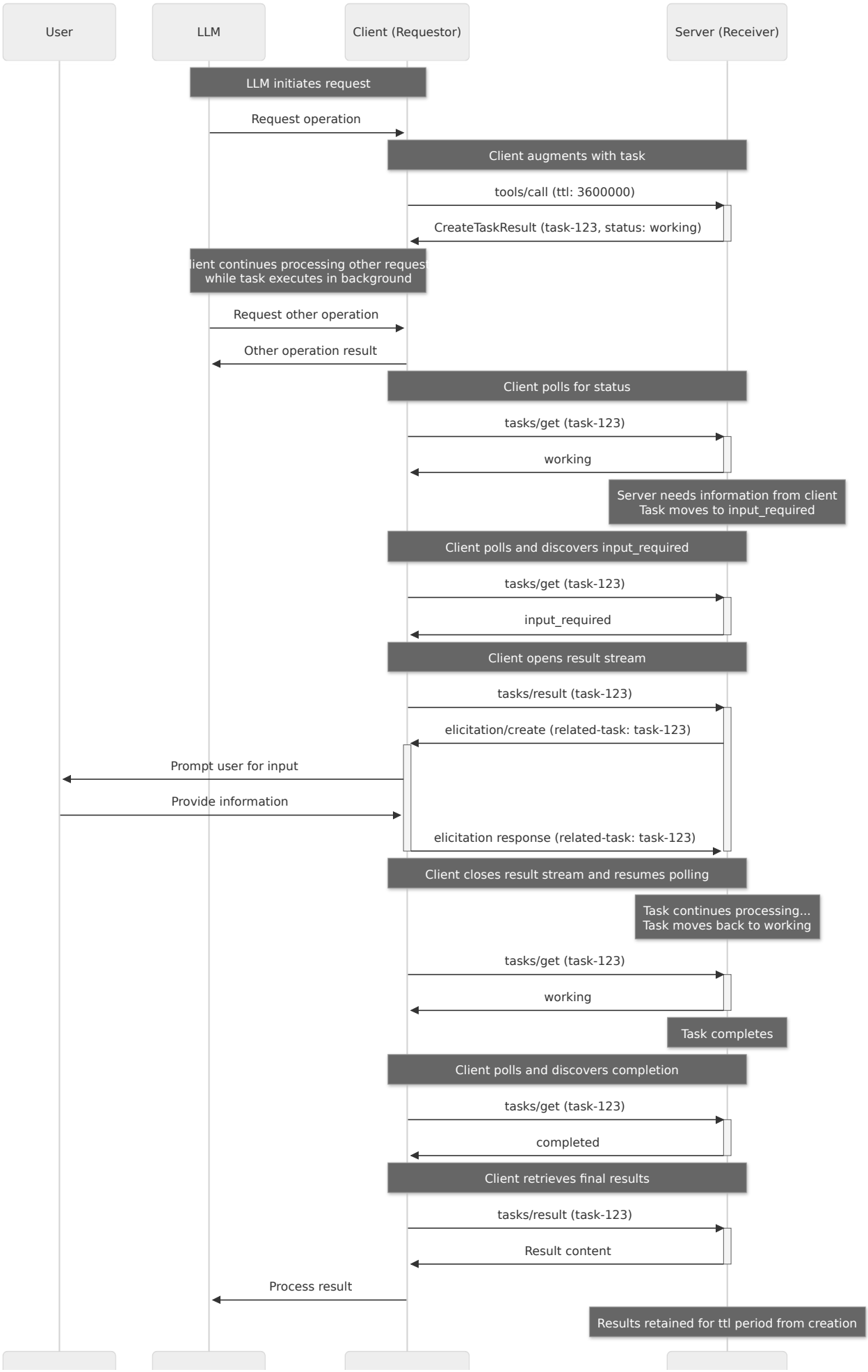
1. Receivers **MUST** reject cancellation requests for tasks already in a terminal status (`completed`, `failed`, or `cancelled`) with error code `-32602` (Invalid params).
2. Upon receiving a valid cancellation request, receivers **SHOULD** attempt to stop the task execution and **MUST** transition the task to `cancelled` status before sending the response.
3. Once a task is cancelled, it **MUST** remain in `cancelled` status even if execution continues to completion or fails.
4. The `tasks/cancel` operation does not define deletion behavior. However, receivers **MAY** delete cancelled tasks at their discretion at any time, including immediately after cancellation or after the task `ttl` expires.
5. Requestors **SHOULD NOT** rely on cancelled tasks being retained for any specific duration and should retrieve any needed information before cancelling.

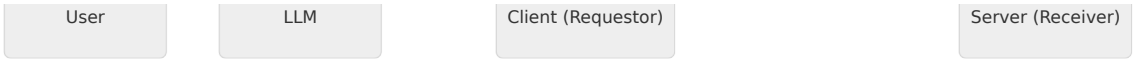
Message Flow

Basic Task Lifecycle

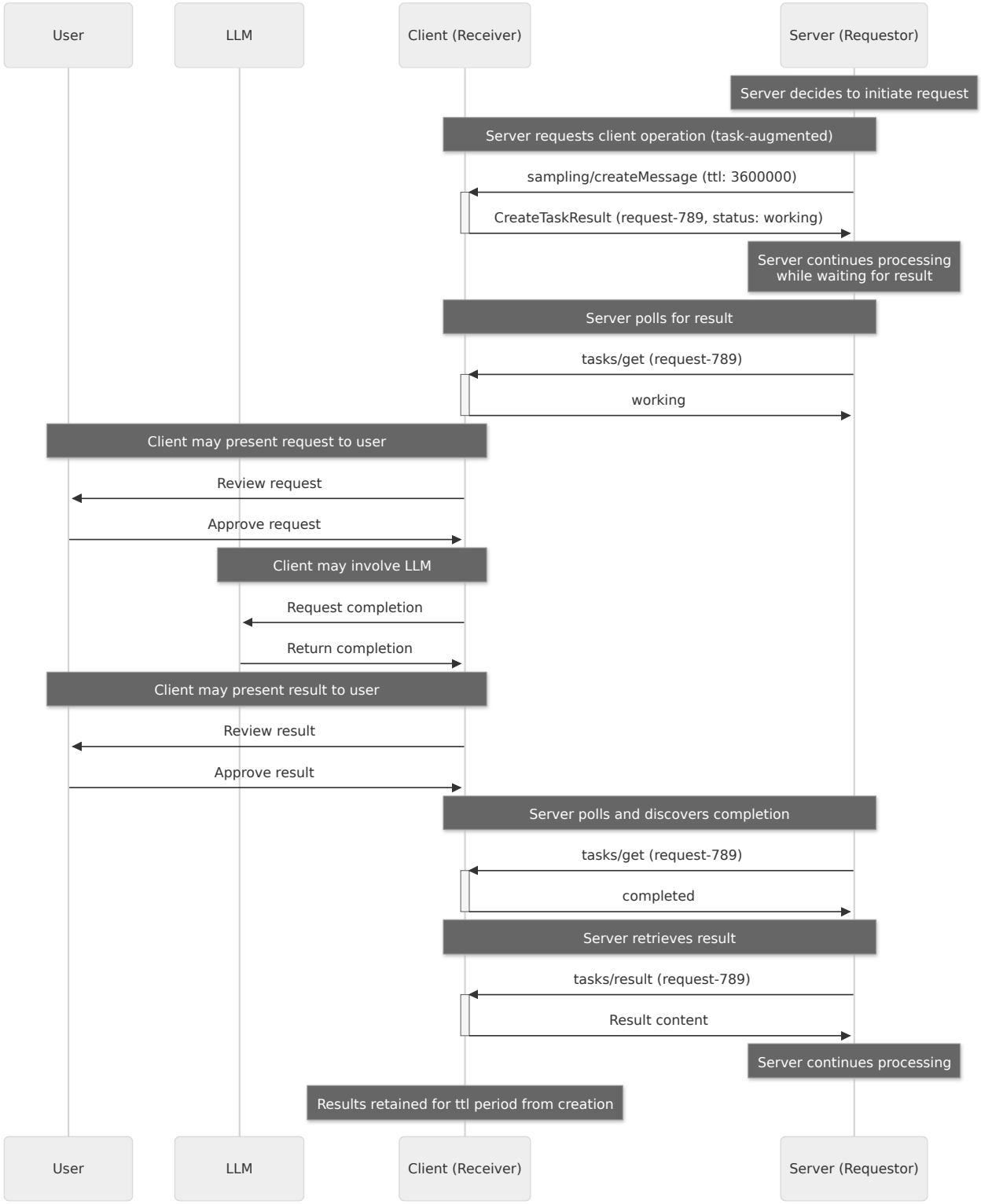


Task-Augmented Tool Call With Elicitation

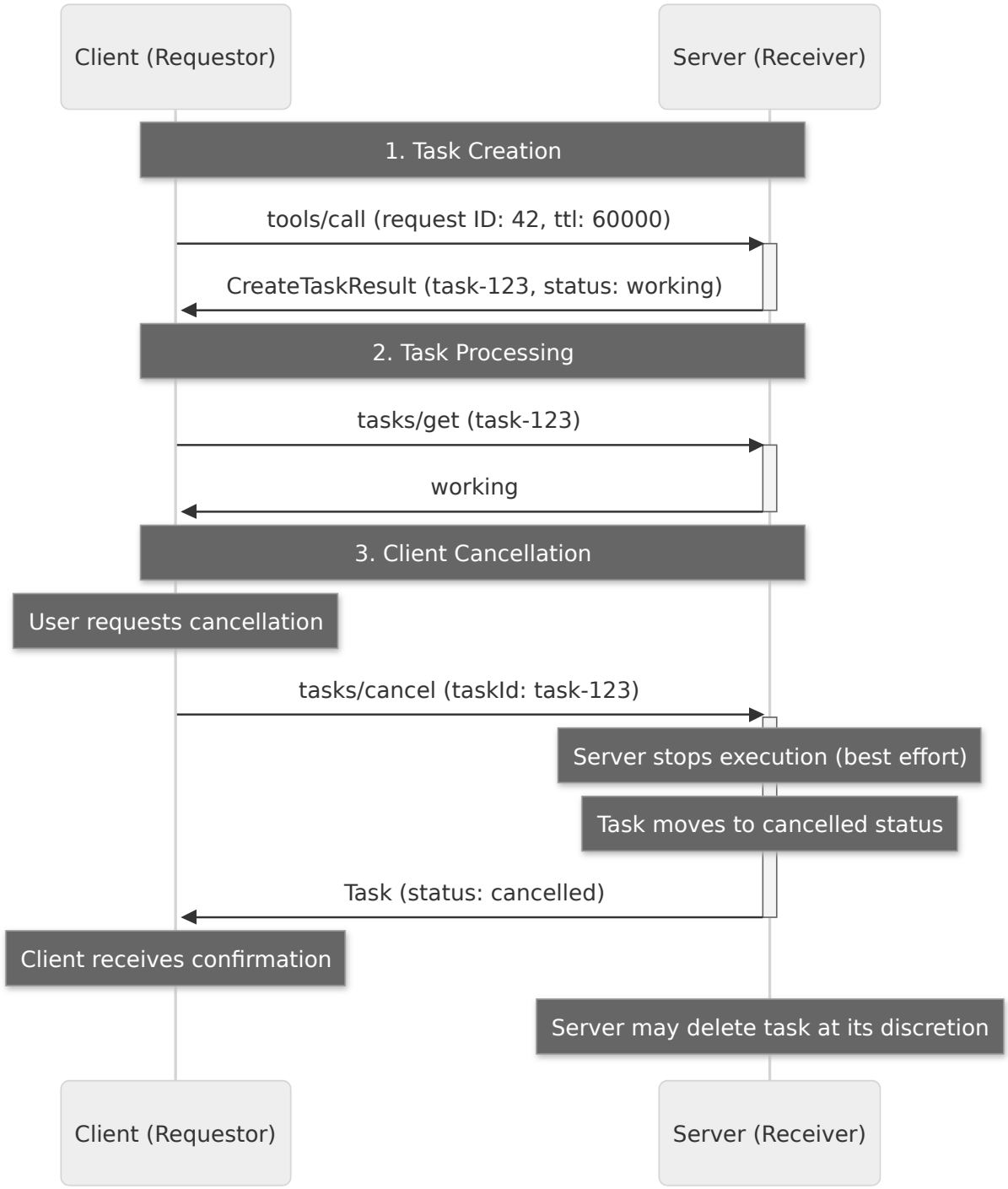

User
LLM
Client (Requestor)
Server (Receiver)
LLM initiates request
Request operation
Client augments with task
tools/call (ttl: 3600000)
CreateTaskResult (task-123, status: working)
ient continues processing other reques
while task executes in background
Request other operation
Other operation result
Client polls for status
tasks/get (task-123)
working
Server needs information from client
Task moves to input_required
Client polls and discovers input_required
tasks/get (task-123)
input_required
Client opens result stream
tasks/result (task-123)
elicitation/create (related-task: task-123)
Prompt user for input
Provide information
elicitation response (related-task: task-123)
Client closes result stream and resumes polling
Task continues processing...
Task moves back to working
tasks/get (task-123)
working
Task completes
Client polls and discovers completion
tasks/get (task-123)
completed
Client retrieves final results
tasks/result (task-123)
Result content
Process result
Results retained for ttl period from creation



Task-Augmented Sampling Request



Task Cancellation Flow



Data Types

Task

A task represents the execution state of a request. The task state includes:

- `taskId`: Unique identifier for the task
- `status`: Current state of the task execution

- `statusMessage`: Optional human-readable message describing the current state (can be present for any status, including error details for failed tasks)
- `createdAt`: ISO 8601 timestamp when the task was created
- `ttl`: Time in milliseconds from creation before task may be deleted
- `pollInterval`: Suggested time in milliseconds between status checks
- `lastUpdatedAt`: ISO 8601 timestamp when the task status was last updated

Task Status

Tasks can be in one of the following states:

- `working`: The request is currently being processed.
- `input_required`: The receiver needs input from the requestor. The requestor should call `tasks/result` to receive input requests, even though the task has not reached a terminal state.
- `completed`: The request completed successfully and results are available.
- `failed`: The associated request did not complete successfully. For tool calls specifically, this includes cases where the tool call result has `isError` set to true.
- `cancelled`: The request was cancelled before completion.

Task Parameters

When augmenting a request with task execution, the `task` field is included in the request parameters:

```
{
  "task": {
    "ttl": 60000
  }
}
```

Fields:

- `ttl` (number, optional): Requested duration in milliseconds to retain task from creation

Related Task Metadata

All requests, responses, and notifications associated with a task **MUST** include the `io.modelcontextprotocol/related-task` key in `_meta`:

```
{
  "io.modelcontextprotocol/related-task": {
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840"
  }
}
```

This associates messages with their originating task across the entire request lifecycle.

For the `tasks/get`, `tasks/list`, and `tasks/cancel` operations, requestors and receivers **SHOULD NOT** include this metadata in their messages, as the `taskId` is already present in the message structure. The `tasks/result` operation **MUST** include this metadata in its response, as the result structure itself does not contain the task ID.

Error Handling

Tasks use two error reporting mechanisms:

1. **Protocol Errors**: Standard JSON-RPC errors for protocol-level issues
2. **Task Execution Errors**: Errors in the underlying request execution, reported through task status

Protocol Errors

Receivers **MUST** return standard JSON-RPC errors for the following protocol error cases:

- Invalid or nonexistent `taskId` in `tasks/get`, `tasks/result`, or `tasks/cancel`: `-32602` (Invalid params)
- Invalid or nonexistent cursor in `tasks/list`: `-32602` (Invalid params)
- Attempt to cancel a task already in a terminal status: `-32602` (Invalid params)
- Internal errors: `-32603` (Internal error)

Additionally, receivers **MAY** return the following errors:

- Non-task-augmented request when receiver requires task augmentation for that request type: `-32600` (Invalid request)

Receivers **SHOULD** provide informative error messages to describe the cause of errors.

Example: Task augmentation required

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "error": {
    "code": -32600,
    "message": "Task augmentation required for tools/call requests"
  }
}
```

Example: Task not found

```
{
  "jsonrpc": "2.0",
  "id": 70,
  "error": {
    "code": -32602,
    "message": "Failed to retrieve task: Task not found"
  }
}
```

Example: Task expired

```
{
  "jsonrpc": "2.0",
  "id": 71,
  "error": {
    "code": -32602,
    "message": "Failed to retrieve task: Task has expired"
  }
}
```

> **NOTE**
>
> Receivers are not required to retain tasks indefinitely. It is compliant behavior for a receiver to return an error stating the task cannot be found if it has purged an expired task.

Example: Task cancellation rejected (already terminal)

```
{
  "jsonrpc": "2.0",
  "id": 74,
  "error": {
    "code": -32602,
    "message": "Cannot cancel task: already in terminal status 'completed'"
  }
}
```

Task Execution Errors

When the underlying request does not complete successfully, the task moves to the `failed` status. This includes JSON-RPC protocol errors during request execution, or for tool calls specifically, when the tool result has `isError` set to true. The `tasks/get` response **SHOULD** include a `statusMessage` field with diagnostic information about the failure.

Example: Task with execution error

```
{
  "jsonrpc": "2.0",
  "id": 4,
  "result": {
    "taskId": "786512e2-9e0d-44bd-8f29-789f820fe840",
    "status": "failed",
    "createdAt": "2025-11-25T10:30:00Z",
    "lastUpdatedAt": "2025-11-25T10:40:00Z",
    "ttl": 30000,
    "statusMessage": "Tool execution failed: API rate limit exceeded"
  }
}
```

For tasks that wrap tool call requests, when the tool result has `isError` set to `true`, the task should reach `failed` status.

The `tasks/result` endpoint returns exactly what the underlying request would have returned:

- If the underlying request resulted in a JSON-RPC error, `tasks/result` **MUST** return that same JSON-RPC error.
- If the request completed with a JSON-RPC response, `tasks/result` **MUST** return a successful JSON-RPC response containing that result.

Security Considerations

Task Isolation and Access Control

Task IDs are the primary mechanism for accessing task state and results. Without proper access controls, any party that can guess or obtain a task ID could potentially access sensitive information or manipulate tasks they did not create.

When an authorization context is provided, receivers **MUST** bind tasks to said context.

Context-binding is not practical for all applications. Some MCP servers operate in environments without authorization, such as single-user tools, or use transports that don't support authorization. In these scenarios, receivers **SHOULD** document this limitation clearly, as task results may be accessible to any requestor that can guess the task ID. If context-binding is unavailable, receivers **MUST** generate cryptographically secure task IDs with enough entropy to prevent guessing and should consider using shorter TTL durations to reduce the exposure window. Furthermore, receivers that cannot identify requestors **SHOULD NOT** declare the `tasks.list` capability, as listing tasks would expose task metadata to any requestor regardless of task ID entropy.

If context-binding is available, receivers **MUST** reject `tasks/get`, `tasks/result`, and `tasks/cancel` requests for tasks that do not belong to the same authorization context as the requestor. For `tasks/list` requests, receivers **MUST** ensure the returned task list includes only tasks associated with the requestor's authorization context.

Additionally, receivers **SHOULD** implement rate limiting on task operations to prevent denial-of-service and enumeration attacks.

Resource Management

1. Receivers **SHOULD**:
    1. Enforce limits on concurrent tasks per requestor
    2. Enforce maximum `ttl` durations to prevent indefinite resource retention
    3. Clean up expired tasks promptly to free resources
    4. Document maximum supported `ttl` duration
    5. Document maximum concurrent tasks per requestor
    6. Implement monitoring and alerting for resource usage

Audit and Logging

1. Receivers **SHOULD**:
    1. Log task creation, completion, and retrieval events for audit purposes
    2. Include auth context in logs when available
    3. Monitor for suspicious patterns (e.g., many failed task lookups, excessive polling)
2. Requestors **SHOULD**:
    1. Log task lifecycle events for debugging and audit purposes
    2. Track task IDs and their associated operations

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:

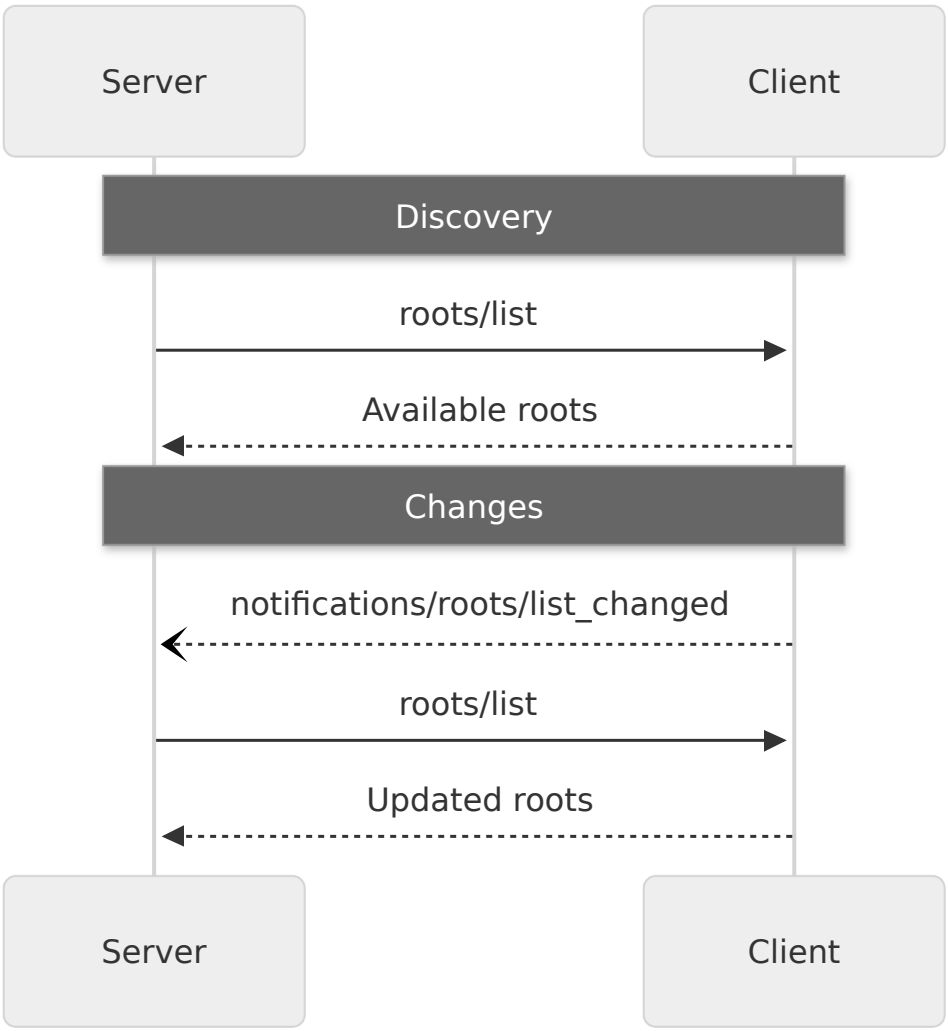
```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "roots": [
      {
        "uri": "file:///home/user/projects/myproject",
        "name": "My Project"
      }
    ]
  }
}
```

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

Tools in Sampling

Servers can request that the client's LLM use tools during sampling by providing a `tools` array and optional `toolChoice` configuration in their sampling requests. This enables servers to implement agentic behaviors where the LLM can call tools, receive results, and continue the conversation - all within a single sampling request flow.

Clients **MUST** declare support for tool use via the `sampling.tools` capability to receive tool-enabled sampling requests. Servers **MUST NOT** send tool-enabled sampling requests to Clients that have not declared support for tool use via the `sampling.tools` capability.

Capabilities

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

Basic sampling:

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

With tool use support:

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

With context inclusion support (soft-deprecated):

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

> **NOTE**
>
> The `includeContext` parameter values `"thisServer"` and `"allServers"` are soft-deprecated. Servers **SHOULD** avoid using these values (e.g. can just omit `includeContext` since it defaults to `"none"`), and **SHOULD NOT** use them unless the client declares `sampling.context` capability. These values may be removed in future spec releases.

Protocol Messages

Creating Messages

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

Request:

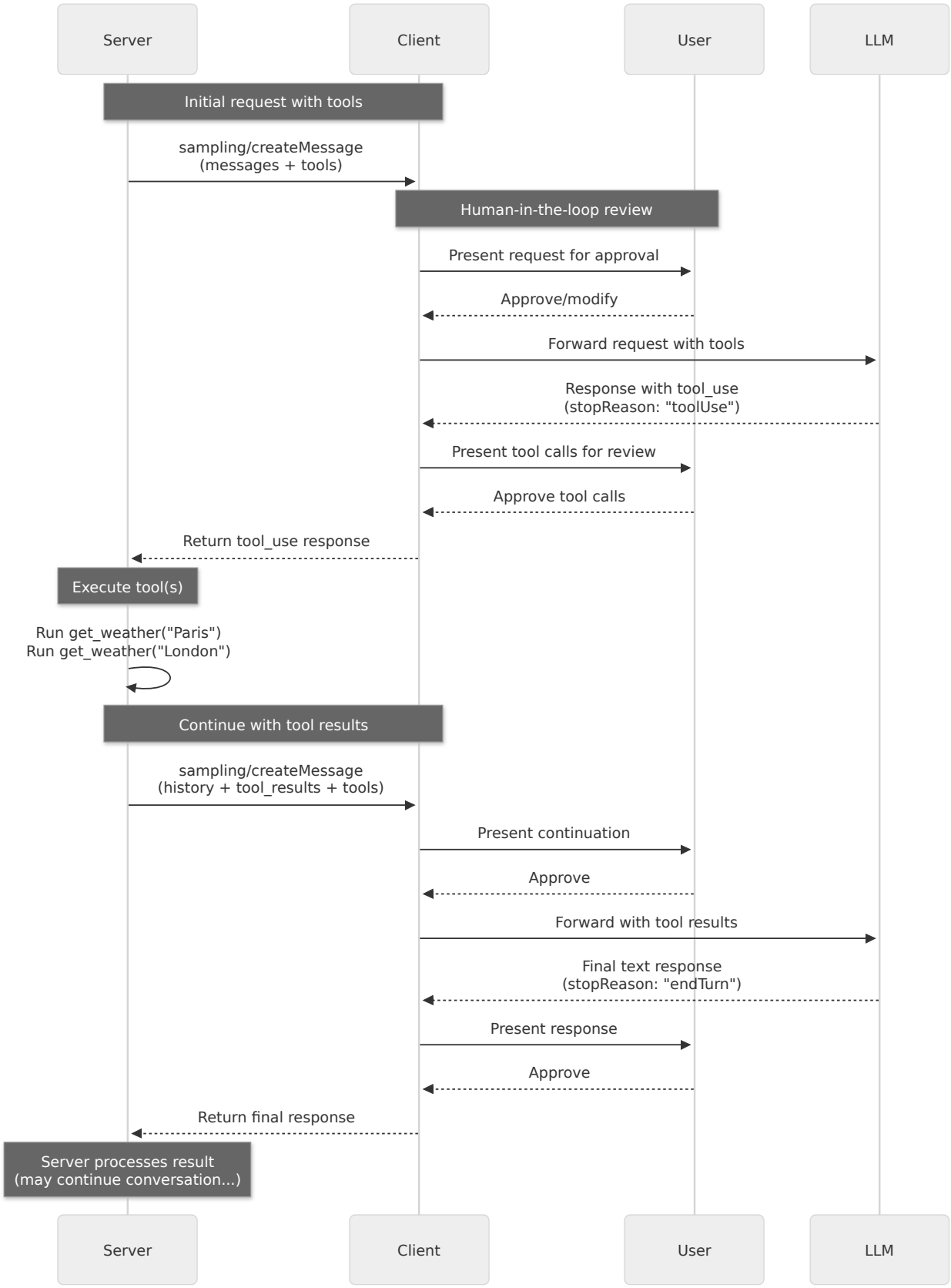
```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "sampling/createMessage",
  "params": {
    "messages": [
      {
        "role": "user",
        "content": {
          "type": "text",
          "text": "What is the capital of France?"
        }
      }
    ],
    "modelPreferences": {
      "hints": [
        {
          "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"
  }
}
```

Sampling with Tools

The following diagram illustrates the complete flow of sampling with tools, including the multi-turn tool loop:



To request LLM generation with tool use capabilities, servers include `tools` and optionally `toolChoice` in the request:

Request (Server -> Client):

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "sampling/createMessage",
  "params": {
    "messages": [
      {
        "role": "user",
        "content": {
          "type": "text",
          "text": "What's the weather like in Paris and London?"
        }
      }
    ],
    "tools": [
      {
        "name": "get_weather",
        "description": "Get current weather for a city",
        "inputSchema": {
          "type": "object",
          "properties": {
            "city": {
              "type": "string",
              "description": "City name"
            }
          },
          "required": ["city"]
        }
      }
    ],
    "toolChoice": {
      "mode": "auto"
    },
    "maxTokens": 1000
  }
}
```

Response (Client -> Server):

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "role": "assistant",
    "content": [
      {
        "type": "tool_use",
        "id": "call_abc123",
        "name": "get_weather",
        "input": {
          "city": "Paris"
        }
      },
      {
        "type": "tool_use",
        "id": "call_def456",
        "name": "get_weather",
        "input": {
          "city": "London"
        }
      }
    ],
    "model": "claude-3-sonnet-20240307",
    "stopReason": "toolUse"
  }
}
```

Multi-turn Tool Loop

After receiving tool use requests from the LLM, the server typically:

1. Executes the requested tool uses.
2. Sends a new sampling request with the tool results appended
3. Receives the LLM's response (which might contain new tool uses)
4. Repeats as many times as needed (server might cap the maximum number of iterations, and e.g. pass `toolChoice: {mode: "none"}` on the last iteration to force a final result)

Follow-up request (Server -> Client) with tool results:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "sampling/createMessage",
  "params": {
    "messages": [
      {
        "role": "user",
        "content": {
          "type": "text",
          "text": "What's the weather like in Paris and London?"
        }
      },
      {
        "role": "assistant",
        "content": [
          {
            "type": "tool_use",
            "id": "call_abc123",
            "name": "get_weather",
            "input": { "city": "Paris" }
          },
          {
            "type": "tool_use",
            "id": "call_def456",
            "name": "get_weather",
            "input": { "city": "London" }
          }
        ]
      },
      {
        "role": "user",
        "content": [
          {
            "type": "tool_result",
            "toolUseId": "call_abc123",
            "content": [
              {
                "type": "text",
                "text": "Weather in Paris: 18°C, partly cloudy"
              }
            ]
          },
          {
            "type": "tool_result",
            "toolUseId": "call_def456",
            "content": [
```

```
              {
                "type": "text",
                "text": "Weather in London: 15°C, rainy"
              }
            ]
          }
        ]
      }
    ],
    "tools": [
      {
        "name": "get_weather",
        "description": "Get current weather for a city",
        "inputSchema": {
          "type": "object",
          "properties": {
            "city": { "type": "string" }
          },
          "required": ["city"]
        }
      }
    ],
    "maxTokens": 1000
  }
}
```

Final response (Client -> Server):

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "role": "assistant",
    "content": {
      "type": "text",
      "text": "Based on the current weather data:\n\n- **Paris**: 18°C and
partly cloudy - quite pleasant!\n- **London**: 15°C and rainy - you'll want an
umbrella.\n\nParis has slightly warmer and drier conditions today."
    },
    "model": "claude-3-sonnet-20240307",
    "stopReason": "endTurn"
  }
}
```

Message Content Constraints

Tool Result Messages

When a user message contains tool results (type: "tool_result"), it **MUST** contain ONLY tool results. Mixing tool results with other content types (text, image, audio) in the same message is not allowed.

This constraint ensures compatibility with provider APIs that use dedicated roles for tool results (e.g., OpenAI's "tool" role, Gemini's "function" role).

Valid - single tool result:

```
{
  "role": "user",
  "content": {
    "type": "tool_result",
    "toolUseId": "call_123",
    "content": [{ "type": "text", "text": "Result data" }]
  }
}
```

Valid - multiple tool results:

```
{
  "role": "user",
  "content": [
    {
      "type": "tool_result",
      "toolUseId": "call_123",
      "content": [{ "type": "text", "text": "Result 1" }]
    },
    {
      "type": "tool_result",
      "toolUseId": "call_456",
      "content": [{ "type": "text", "text": "Result 2" }]
    }
  ]
}
```

Invalid - mixed content:

```
{
  "role": "user",
  "content": [
    {
      "type": "text",
      "text": "Here are the results:"
    },
    {
      "type": "tool_result",
      "toolUseId": "call_123",
      "content": [{ "type": "text", "text": "Result data" }]
    }
  ]
}
```

Tool Use and Result Balance

When using tool use in sampling, every assistant message containing `ToolUseContent` blocks **MUST** be followed by a user message that consists entirely of `ToolResultContent` blocks, with each tool use (e.g. with `id: $id`) matched by a corresponding tool result (with `toolUseId: $id`), before any other message.

This requirement ensures:

- Tool uses are always resolved before the conversation continues
- Provider APIs can concurrently process multiple tool uses and fetch their results in parallel
- The conversation maintains a consistent request-response pattern

Example valid sequence:

1. User message: "What's the weather like in Paris and London?"
2. Assistant message: `ToolUseContent` (`id: "call_abc123", name: "get_weather", input: {city: "Paris"}`) + `ToolUseContent` (`id: "call_def456", name: "get_weather", input: {city: "London"}`)
3. User message: `ToolResultContent` (`toolUseId: "call_abc123", content: "18°C, partly cloudy"`) + `ToolResultContent` (`toolUseId: "call_def456", content: "15°C, rainy"`)
4. Assistant message: Text response comparing the weather in both cities

Invalid sequence - missing tool result:

1. User message: "What's the weather like in Paris and London?"
2. Assistant message: `ToolUseContent` (`id: "call_abc123", name: "get_weather", input: {city: "Paris"}`) + `ToolUseContent` (`id: "call_def456", name: "get_weather", input: {city: "London"}`)
3. User message: `ToolResultContent` (`toolUseId: "call_abc123", content: "18°C, partly cloudy"`) ← Missing result for call_def456
4. Assistant message: Text response (invalid - not all tool uses were resolved)

Cross-API Compatibility

The sampling specification is designed to work across multiple LLM provider APIs (Claude, OpenAI, Gemini, etc.). Key design decisions for compatibility:

Message Roles

MCP uses two roles: "user" and "assistant".

Tool use requests are sent in CreateMessageResult with the "assistant" role. Tool results are sent back in messages with the "user" role. Messages with tool results cannot contain other kinds of content.

Tool Choice Modes

`CreateMessageRequest.params.toolChoice` controls the tool use ability of the model:

- `{mode: "auto"}`: Model decides whether to use tools (default)
- `{mode: "required"}`: Model MUST use at least one tool before completing
- `{mode: "none"}`: Model MUST NOT use any tools

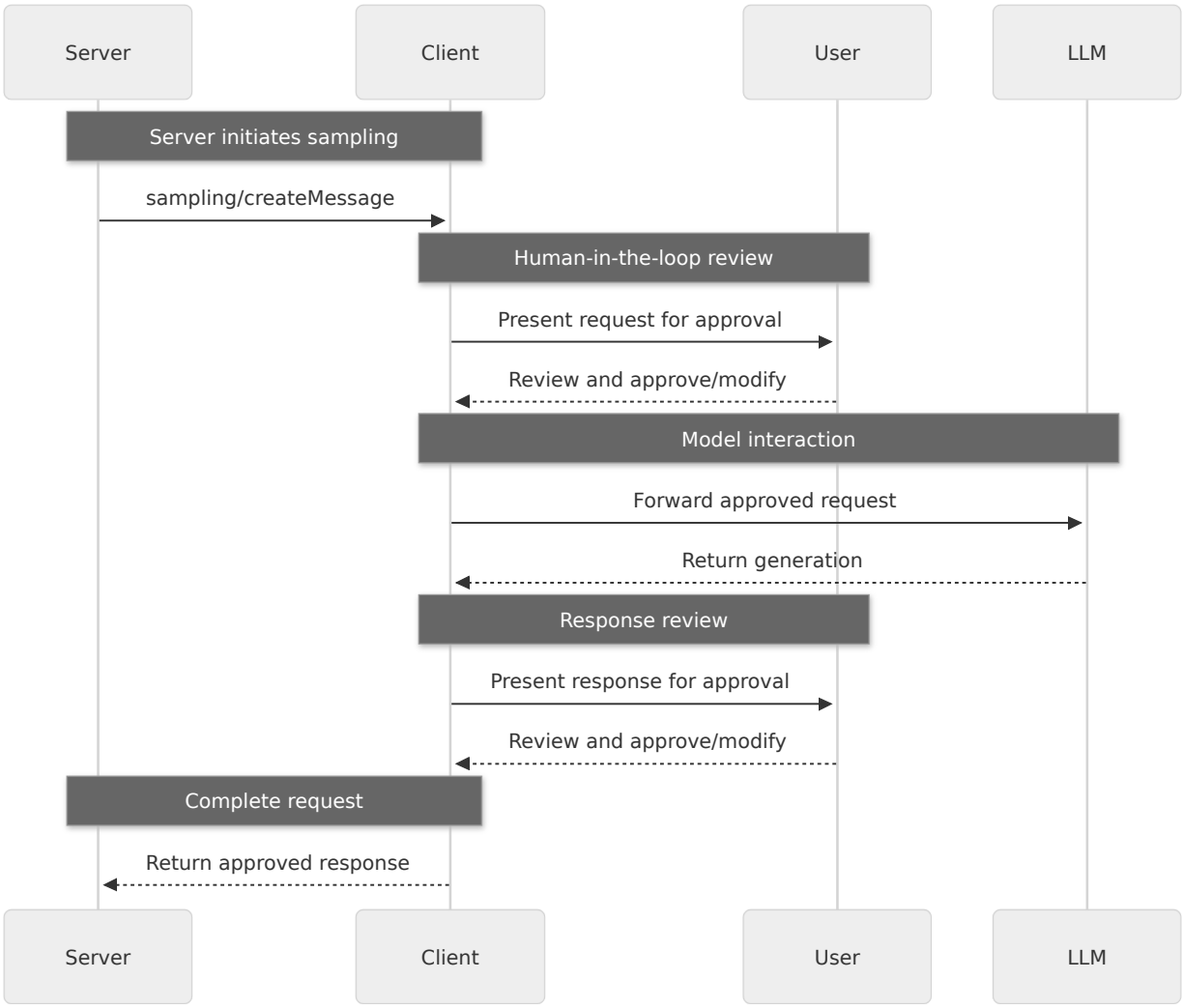
Parallel Tool Use

MCP allows models to make multiple tool use requests in parallel (returning an array of `ToolUseContent`). All major provider APIs support this:

- **Claude**: Supports parallel tool use natively
- **OpenAI**: Supports parallel tool calls (can be disabled with `parallel_tool_calls: false`)
- **Gemini**: Supports parallel function calls natively

Implementations wrapping providers that support disabling parallel tool use MAY expose this as an extension, but it is not part of the core MCP specification.

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:

- User rejected sampling request: `-1`
- Tool result missing in request: `-32602` (Invalid params)
- Tool results mixed with other content: `-32602` (Invalid params)

Example errors:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "error": {
    "code": -1,
    "message": "User rejected sampling request"
  }
}
```

```
{
  "jsonrpc": "2.0",
  "id": 4,
  "error": {
    "code": -32602,
    "message": "Tool result missing in 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

When tools are used in sampling, additional security considerations apply:

6. Servers **MUST** ensure that when replying to a `stopReason: "toolUse"`, each `ToolUseContent` item is responded to with a `ToolResultContent` item with a matching `toolUseId`, and that the user message contains only tool results (no other content types)
7. Both parties **SHOULD** implement iteration limits for tool loops

5.3 Elicitation

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.

Elicitation supports two modes:

- **Form mode**: Servers can request structured data from users with optional JSON schemas to validate responses
- **URL mode**: Servers can direct users to external URLs for sensitive interactions that must *not* pass through the MCP client

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 form mode elicitation to request sensitive information such as passwords, API keys, access tokens, or payment credentials
- Servers **MUST** use URL mode for interactions involving such sensitive information

"Sensitive information" in this context refers to secrets and credentials that grant access or authorize transactions. General contact or profile information (such as a name, email address, or username) is not categorically prohibited; whether to request such data via form mode is at the discretion of the server and subject to the user's ability to review and decline.

MCP clients **MUST**:

- Provide UI that makes it clear which server is requesting information
- Respect user privacy and provide clear decline and cancel options
- For form mode, allow users to review and modify their responses before sending
- For URL mode, clearly display the target domain/host and gather user consent before navigation to the target URL

Capabilities

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

```
{
  "capabilities": {
    "elicitation": {
      "form": {},
      "url": {}
    }
  }
}
```

For backwards compatibility, an empty capabilities object is equivalent to declaring support for `form` mode only:

```
{
  "capabilities": {
    "elicitation": {}, // Equivalent to { "form": {} }
  },
}
```

Clients declaring the `elicitation` capability **MUST** support at least one mode (`form` or `url`).

Servers **MUST NOT** send elicitation requests with modes that are not supported by the client.

Protocol Messages

Elicitation Requests

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

All elicitation requests **MUST** include the following parameters:

Name	Type	Options	Description
`mode`	string	`form` , `url`	The mode of the elicitation. Optional for form mode (defaults to `"form"` if omitted).
`message`	string		A human-readable message explaining why the interaction is needed.

The `mode` parameter specifies the type of elicitation:

- `"form"` : In-band structured data collection with optional schema validation. Data is exposed to the client.

- `"url"`: Out-of-band interaction via URL navigation. Data (other than the URL itself) is **not** exposed to the client.

For backwards compatibility, servers **MAY** omit the `mode` field for form mode elicitation requests. Clients **MUST** treat requests without a `mode` field as form mode.

Form Mode Elicitation Requests

Form mode elicitation allows servers to collect structured data directly through the MCP client.

Form mode elicitation requests **MUST** either specify `mode: "form"` or omit the `mode` field, and include these additional parameters:

Name	Type	Description
`requestedSchema`	object	A JSON Schema defining the structure of the expected response.

Requested Schema

The `requestedSchema` parameter allows servers to define the structure of the expected response using a restricted subset of JSON Schema.

To simplify client user experience, form mode elicitation schemas are limited to flat objects with primitive properties only.

The schema is restricted to these primitive types:

1. **String Schema**

```
{
  "type": "string",
  "title": "Display Name",
  "description": "Description text",
  "minLength": 3,
  "maxLength": 50,
  "pattern": "^[A-Za-z]+$",
  "format": "email",
  "default": "user@example.com"
}
```

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

2. **Number Schema**

```
{
  "type": "number", // or "integer"
  "title": "Display Name",
  "description": "Description text",
  "minimum": 0,
  "maximum": 100,
  "default": 50
}
```

3. **Boolean Schema**

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

4. **Enum Schema**

   Single-select enum (without titles):

```
{
  "type": "string",
  "title": "Color Selection",
  "description": "Choose your favorite color",
  "enum": ["Red", "Green", "Blue"],
  "default": "Red"
}
```

   Single-select enum (with titles):

```
{
  "type": "string",
  "title": "Color Selection",
  "description": "Choose your favorite color",
  "oneOf": [
    { "const": "#FF0000", "title": "Red" },
    { "const": "#00FF00", "title": "Green" },
    { "const": "#0000FF", "title": "Blue" }
  ],
  "default": "#FF0000"
}
```

   Multi-select enum (without titles):

```
{
  "type": "array",
  "title": "Color Selection",
  "description": "Choose your favorite colors",
  "minItems": 1,
  "maxItems": 2,
  "items": {
    "type": "string",
    "enum": ["Red", "Green", "Blue"]
  },
  "default": ["Red", "Green"]
}
```

Multi-select enum (with titles):

```
{
  "type": "array",
  "title": "Color Selection",
  "description": "Choose your favorite colors",
  "minItems": 1,
  "maxItems": 2,
  "items": {
    "anyOf": [
      { "const": "#FF0000", "title": "Red" },
      { "const": "#00FF00", "title": "Green" },
      { "const": "#0000FF", "title": "Blue" }
    ]
  },
  "default": ["#FF0000", "#00FF00"]
}
```

Clients can use this schema to:

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

All primitive types support optional default values to provide sensible starting points. Clients that support defaults SHOULD pre-populate form fields with these values.

Note that complex nested structures, arrays of objects (beyond enums), and other advanced JSON Schema features are intentionally not supported to simplify client user experience.

Example: Simple Text Request

Request:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "elicitation/create",
  "params": {
    "mode": "form",
    "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"
    }
  }
}
```

Example: Structured Data Request

Request:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "elicitation/create",
  "params": {
    "mode": "form",
    "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
    }
  }
}
```

URL Mode Elicitation Requests

> **NOTE**
>
> **New feature:** URL mode elicitation is introduced in the `2025-11-25` version of the MCP specification. Its design and implementation may change in future protocol revisions.

URL mode elicitation enables servers to direct users to external URLs for out-of-band interactions that must not pass through the MCP client. This is essential for auth flows, payment processing, and other sensitive or secure operations.

URL mode elicitation requests **MUST** specify `mode: "url"`, a `message`, and include these additional parameters:

Name	Type	Description
`url`	string	The URL that the user should navigate to.
`elicitationId`	string	A unique identifier for the elicitation.

The `url` parameter **MUST** contain a valid URL.

> **NOTE**
>
> **Important**: URL mode elicitation is *not* for authorizing the MCP client's access to the MCP server (that's handled by MCP authorization). Instead, it's used when the MCP server needs to obtain sensitive information or third-party authorization on behalf of the user. The MCP client's bearer token remains unchanged. The client's only responsibility is to provide the user with context about the elicitation URL the server wants them to open.

Example: Request Sensitive Data

This example shows a URL mode elicitation request directing the user to a secure URL where they can provide sensitive information (an API key, for example). The same request could direct the user into an OAuth authorization flow, or a payment flow. The only difference is the URL and the message.

Request:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "elicitation/create",
  "params": {
    "mode": "url",
    "elicitationId": "550e8400-e29b-41d4-a716-446655440000",
    "url": "https://mcp.example.com/ui/set_api_key",
    "message": "Please provide your API key to continue."
  }
}
```

Response:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "result": {
    "action": "accept"
  }
}
```

The response with `action: "accept"` indicates that the user has consented to the interaction. It does not mean that the interaction is complete. The interaction occurs out of band and the client is not aware of the outcome until and unless the server sends a notification indicating completion.

Completion Notifications for URL Mode Elicitation

Servers **MAY** send a `notifications/elicitation/complete` notification when an out-of-band interaction started by URL mode elicitation is completed. This allows clients to react programmatically if appropriate.

Servers sending notifications:

- **MUST** only send the notification to the client that initiated the elicitation request.
- **MUST** include the `elicitationId` established in the original `elicitation/create` request.

Clients:

- **MUST** ignore notifications referencing unknown or already-completed IDs.
- **MAY** wait for this notification to automatically retry requests that received a URLElicitationRequiredError, update the user interface, or otherwise continue an interaction.
- **SHOULD** still provide manual controls that let the user retry or cancel the original request (or otherwise resume interacting with the client) if the notification never arrives.

Example

```
{
  "jsonrpc": "2.0",
  "method": "notifications/elicitation/complete",
  "params": {
    "elicitationId": "550e8400-e29b-41d4-a716-446655440000"
  }
}
```

URL Elicitation Required Error

When a request cannot be processed until an elicitation is completed, the server **MAY** return a `URLElicitationRequiredError` (code `-32042`) to indicate to the client that a URL mode elicitation is required. The server **MUST NOT** return this error except when URL mode elicitation is required.

The error **MUST** include a list of elicitations that are required to complete before the original can be retried.

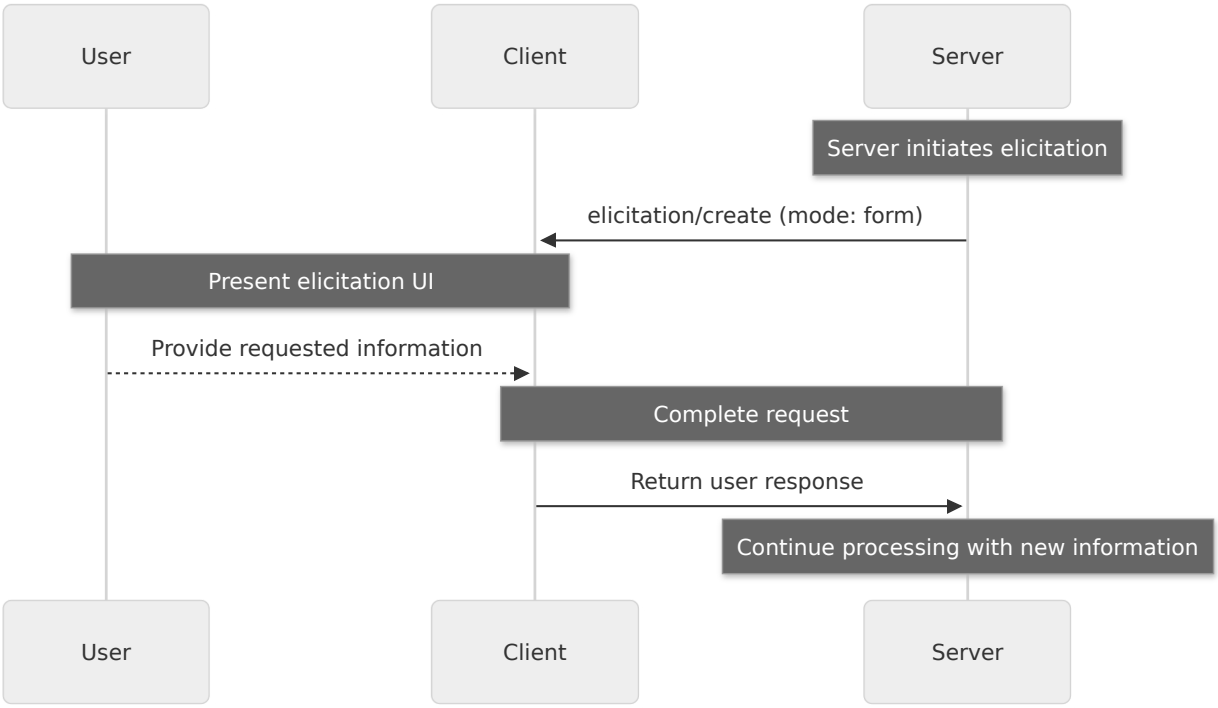
Any elicitations returned in the error **MUST** be URL mode elicitations and have an `elicitationId` property.

Error Response:

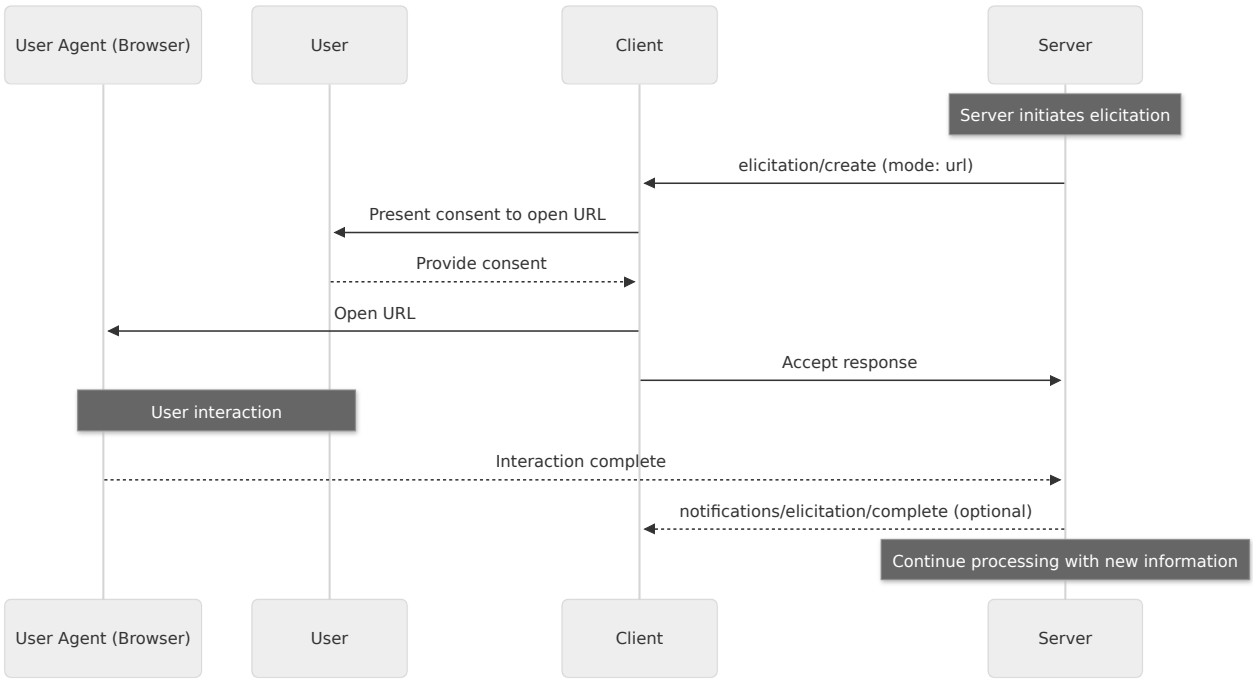
```
{
  "jsonrpc": "2.0",
  "id": 2,
  "error": {
    "code": -32042, // URL_ELICITATION_REQUIRED
    "message": "This request requires more information.",
    "data": {
      "elicitations": [
        {
          "mode": "url",
          "elicitationId": "550e8400-e29b-41d4-a716-446655440000",
          "url": "https://mcp.example.com/connect?elicitationId=550e8400-e29b-
41d4-a716-446655440000",
          "message": "Authorization is required to access your Example Co
files."
        }
      ]
    }
  }
}
```

Message Flow

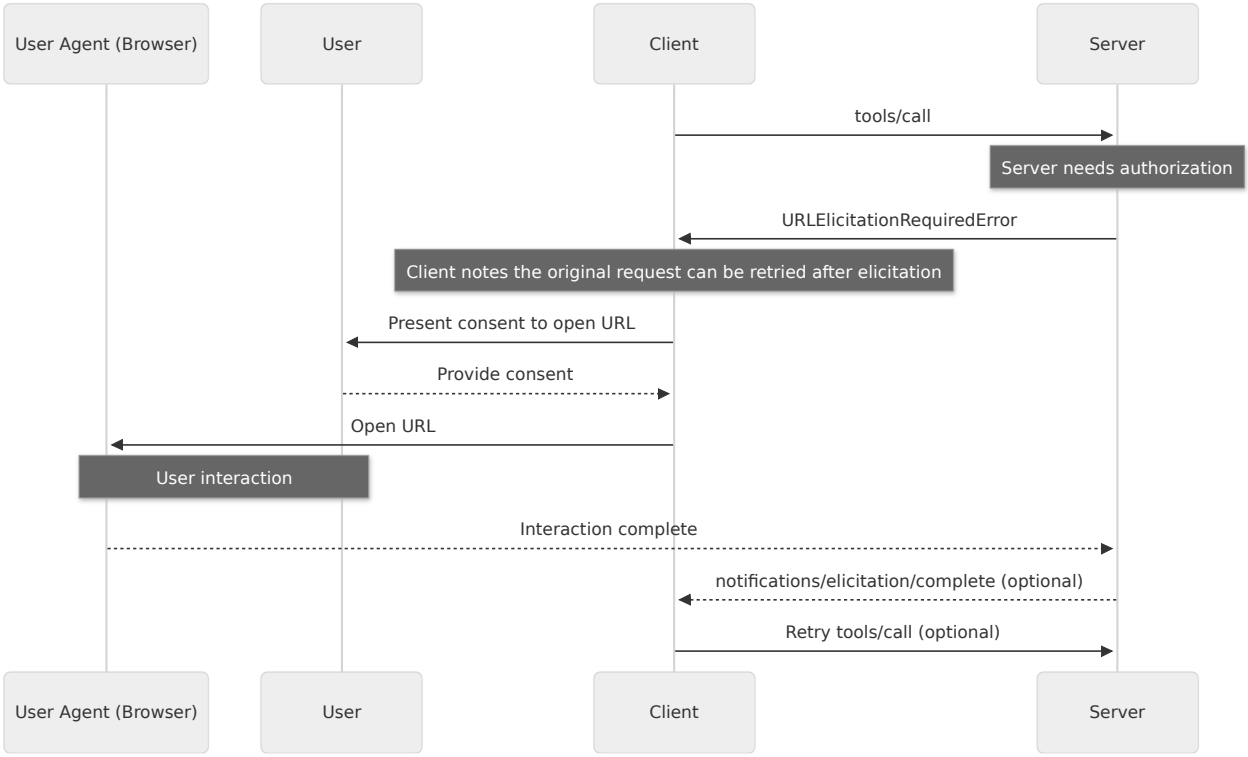
Form Mode Flow



URL Mode Flow



URL Mode With Elicitation Required Error Flow



Response Actions

Elicitation responses use a three-action model to clearly distinguish between different user actions. These actions apply to both form and URL elicitation modes.

```
{
  "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
    - For form mode: The `content` field contains the submitted data matching the requested schema
    - For URL mode: The `content` field is omitted

    - 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, browser failed to load, 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)

Implementation Considerations

Statefulness

Most practical uses of elicitation require that the server maintain state about users:

- Whether required information has been collected (e.g., the user's display name via form mode elicitation)
- Status of resource access (e.g., API keys or a payment flow via URL mode elicitation)

Servers implementing elicitation **MUST** securely associate this state with individual users following the guidelines in the security best practices document. Specifically:

- State **MUST NOT** be associated with session IDs alone
- State storage **MUST** be protected against unauthorized access
- For remote MCP servers, user identification **MUST** be derived from credentials acquired via MCP authorization when possible (e.g. `sub` claim)

> **NOTE**
>
> The examples in this section are non-normative and illustrate potential uses of elicitation. Implementers should adapt these patterns to their specific requirements while maintaining security best practices.

URL Mode Elicitation for Sensitive Data

For servers that interact with external APIs requiring sensitive information (e.g., credentials, payment information), URL mode elicitation provides a secure mechanism for users to provide this information without exposing it to the MCP client.

In this pattern:

1. The server directs users to a secure web page (served over HTTPS)
2. The page presents a branded form UI on a domain the user trusts
3. Users enter sensitive credentials directly into the secure form
4. The server stores credentials securely, bound to the user's identity
5. Subsequent MCP requests use these stored credentials for API access

This approach ensures that sensitive credentials never pass through the LLM context, MCP client or any intermediate MCP servers, reducing the risk of exposure through client-side logging or other attack vectors.

URL Mode Elicitation for OAuth Flows

URL mode elicitation enables a pattern where MCP servers act as OAuth clients to third-party resource servers. Authorization with external APIs enabled by URL mode elicitation is separate from MCP authorization. MCP servers **MUST NOT** rely on URL mode elicitation to authorize users for themselves.

Understanding the Distinction

- **MCP Authorization**: Required OAuth flow between the MCP client and MCP server (covered in the authorization specification)
- **External (third-party) Authorization**: Optional authorization between the MCP server and a third-party resource server, initiated via URL mode elicitation

In external authorization, the server acts as both:

- An OAuth resource server (to the MCP client)
- An OAuth client (to the third-party resource server)

Example scenario:

- An MCP client connects to an MCP server
- The MCP server integrates with various different third-party services
- When the MCP client calls a tool that requires access to a third-party service, the MCP server needs credentials for that service

The critical security requirements are:

1. **The third-party credentials MUST NOT transit through the MCP client**: The client must never see third-party credentials to protect the security boundary
2. **The MCP server MUST NOT use the client's credentials for the third-party service**: That would be token passthrough, which is forbidden
3. **The user MUST authorize the MCP server directly**: The interaction happens outside the MCP protocol, without involving the MCP client
4. **The MCP server is responsible for tokens**: The MCP server is responsible for storing and managing the third-party tokens obtained through the URL mode elicitation (in other words, the MCP server must be stateful).

Credentials obtained via URL mode elicitation are distinct from the MCP server credentials used by the MCP client. The MCP server **MUST NOT** transmit credentials obtained through URL mode elicitation to the MCP client.

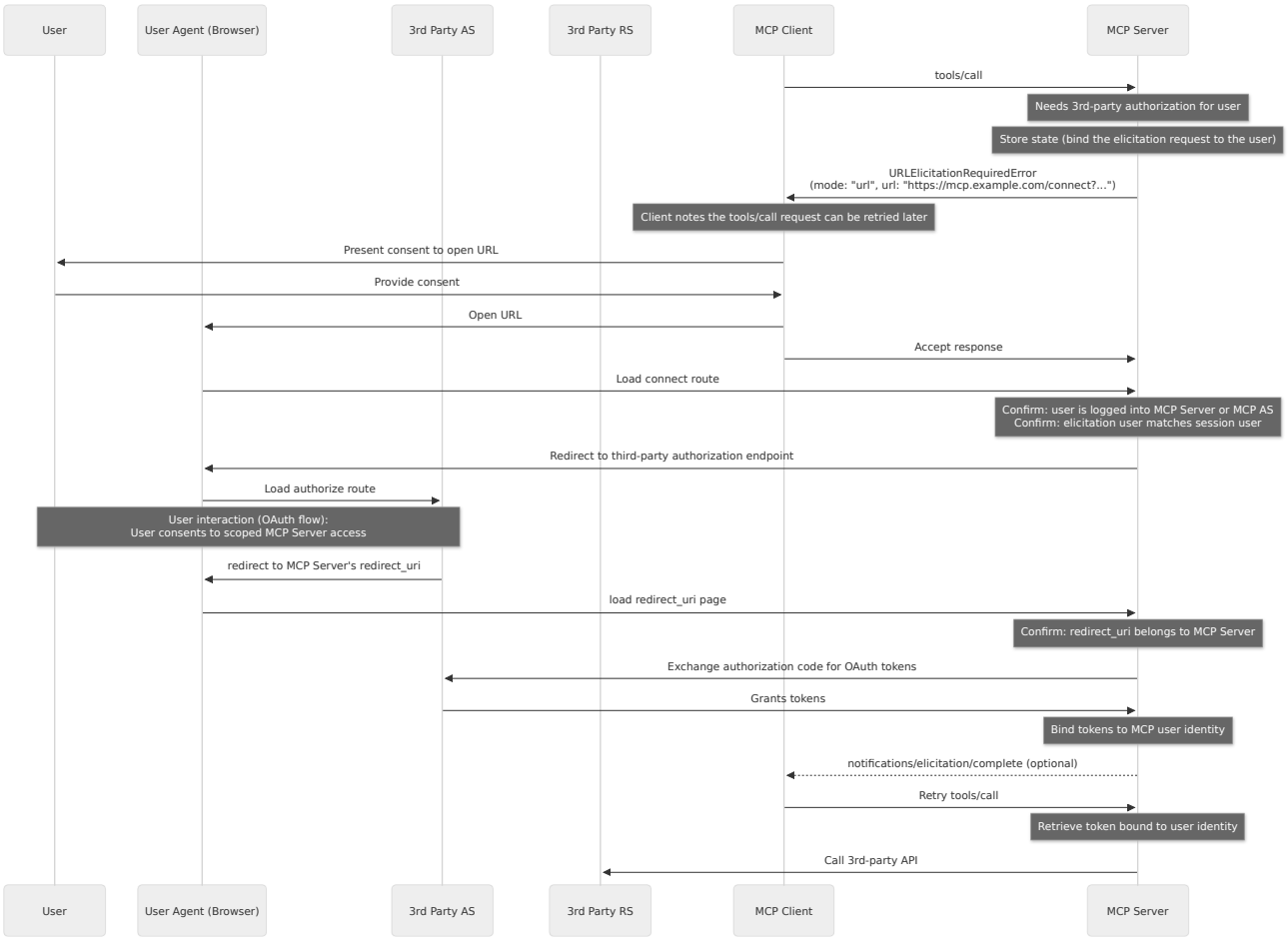
> **NOTE**
>
> For additional background, refer to the token passthrough section of the Security Best Practices document to understand why MCP servers cannot act as pass-through proxies.

Implementation Pattern

When implementing external authorization via URL mode elicitation:

1. The MCP server generates an authorization URL, acting as an OAuth client to the third-party service
2. The MCP server stores internal state that associates (binds) the elicitation request with the user's identity.
3. The MCP server sends a URL mode elicitation request to the client with a URL that can start the authorization flow.
4. The user completes the OAuth flow directly with the third-party authorization server
5. The third-party authorization server redirects back to the MCP server
6. The MCP server securely stores the third-party tokens, bound to the user's identity
7. Future MCP requests can leverage these stored tokens for API access to the third-party resource server

The following is a non-normative example of how this pattern could be implemented:



This pattern maintains clear security boundaries while enabling rich integrations with third-party services that require user authorization.

Error Handling

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

- When a request cannot be processed until an elicitation is completed: `-32042` (`URLElicitationRequiredError`)

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

- Server sends an `elicitation/create` request with a mode not declared in client capabilities: `-32602` (Invalid params)

Security Considerations

1. Servers **MUST** bind elicitation requests to the client and user identity
2. Clients **MUST** provide clear indication of which server is requesting information
3. Clients **SHOULD** implement user approval controls
4. Clients **SHOULD** allow users to decline elicitation requests at any time
5. Clients **SHOULD** implement rate limiting
6. Clients **SHOULD** present elicitation requests in a way that makes it clear what information is being requested and why

Safe URL Handling

MCP servers requesting elicitation:

1. **MUST NOT** include sensitive information about the end-user, including credentials, personal identifiable information, etc., in the URL sent to the client in a URL elicitation request.
2. **MUST NOT** provide a URL which is pre-authenticated to access a protected resource, as the URL could be used to impersonate the user by a malicious client.
3. **SHOULD NOT** include URLs intended to be clickable in any field of a form mode elicitation request.
4. **SHOULD** use HTTPS URLs for non-development environments.

These server requirements ensure that client implementations have clear rules about when to present a URL to the user, so that the client-side rules (below) can be consistently applied.

Clients implementing URL mode elicitation **MUST** handle URLs carefully to prevent users from unknowingly clicking malicious links.

When handling URL mode elicitation requests, MCP clients:

1. **MUST NOT** automatically pre-fetch the URL or any of its metadata.
2. **MUST NOT** open the URL without explicit consent from the user.
3. **MUST** show the full URL to the user for examination before consent.
4. **MUST** open the URL provided by the server in a secure manner that does not enable the client or LLM to inspect the content or user inputs. For example, on iOS,

SFSafariViewController is good, but WkWebView is not.
5. **SHOULD** highlight the domain of the URL to mitigate subdomain spoofing.
6. **SHOULD** have warnings for ambiguous/suspicious URIs (i.e., containing Punycode).
7. **SHOULD NOT** render URLs as clickable in any field of an elicitation request, except for the `url` field in a URL elicitation request (with the restrictions detailed above).

Identifying the User

Servers **MUST NOT** rely on client-provided user identification without server verification, as this can be forged. Instead, servers **SHOULD** follow security best practices.

Non-normative examples:

- Incorrect: Treat user input like "I am joe@example.com" as authoritative
- Correct: Rely on authorization to identify the user

Form Mode Security

1. Servers **MUST NOT** request sensitive information (passwords, API keys, etc.) via form mode
2. Clients **SHOULD** validate all responses against the provided schema
3. Servers **SHOULD** validate received data matches the requested schema

Phishing

URL mode elicitation returns a URL that an attacker can use to send to a victim. The MCP Server **MUST** verify the identity of the user who opens the URL before accepting information.

Typically identity verification is done by leveraging the MCP authorization server to identify the user, through a session cookie or equivalent in the browser.

For example, URL mode elicitation may be used to perform OAuth flows where the server acts as an OAuth client of another resource server. Without proper mitigation, the following phishing attack is possible:

1. A malicious user (Alice) connected to a benign server triggers an elicitation request
2. The benign server generates an authorization URL, acting as an OAuth client of a third-party authorization server
3. Alice's client displays the URL and asks for consent
4. Instead of clicking on the link, Alice tricks a victim user (Bob) of the same benign server into clicking it
5. Bob opens the link and completes the authorization, thinking they are authorizing their own connection to the benign server
6. The benign server receives a callback/redirect form the third-party authorization server, and assumes it's Alice's request
7. The tokens for the third-party server are bound to Alice's session and identity, instead of Bob's, resulting in an account takeover

To prevent this attack, the server **MUST** ensure that the user who started the elicitation request (the end-user who is accessing the server via the MCP client) is the same user who completes the authorization flow.

There are many ways to achieve this and the best way will depend on the specific implementation.

As a common, non-normative example, consider a case where the MCP server is accessible via the web and desires to perform a third-party authorization code flow. To prevent the phishing attack, the server would create a URL mode elicitation to `https://mcp.example.com/connect?elicitationId=...` rather than the third-party authorization endpoint. This "connect URL" must ensure the user who opened the page is the same user who the elicitation was generated for. It would, for example, check that the user has a valid session cookie and that the session cookie is for the same user who was using the MCP client to generate the URL mode elicitation. This could be done by comparing the authoritative subject (`sub` claim) from the MCP server's authorization server to the subject from the session cookie. Once that page ensures the same user, it can send the user to the third-party authorization server at `https://example.com/authorize?...` where a normal OAuth flow can be completed.

In other cases, the server may not be accessible via the web and may not be able to use a session cookie to identify the user. In this case, the server must use a different mechanism to identify the user who opens the elicitation URL is the same user who the elicitation was generated for.

In all implementations, the server **MUST** ensure that the mechanism to determine the user's identity is resilient to attacks where an attacker can modify the elicitation URL.

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
          }
        ],
        "icons": [
          {
            "src": "https://example.com/review-icon.svg",
            "mimeType": "image/svg+xml",
            "sizes": ["any"]
          }
        ]
      }
    ],
    "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:

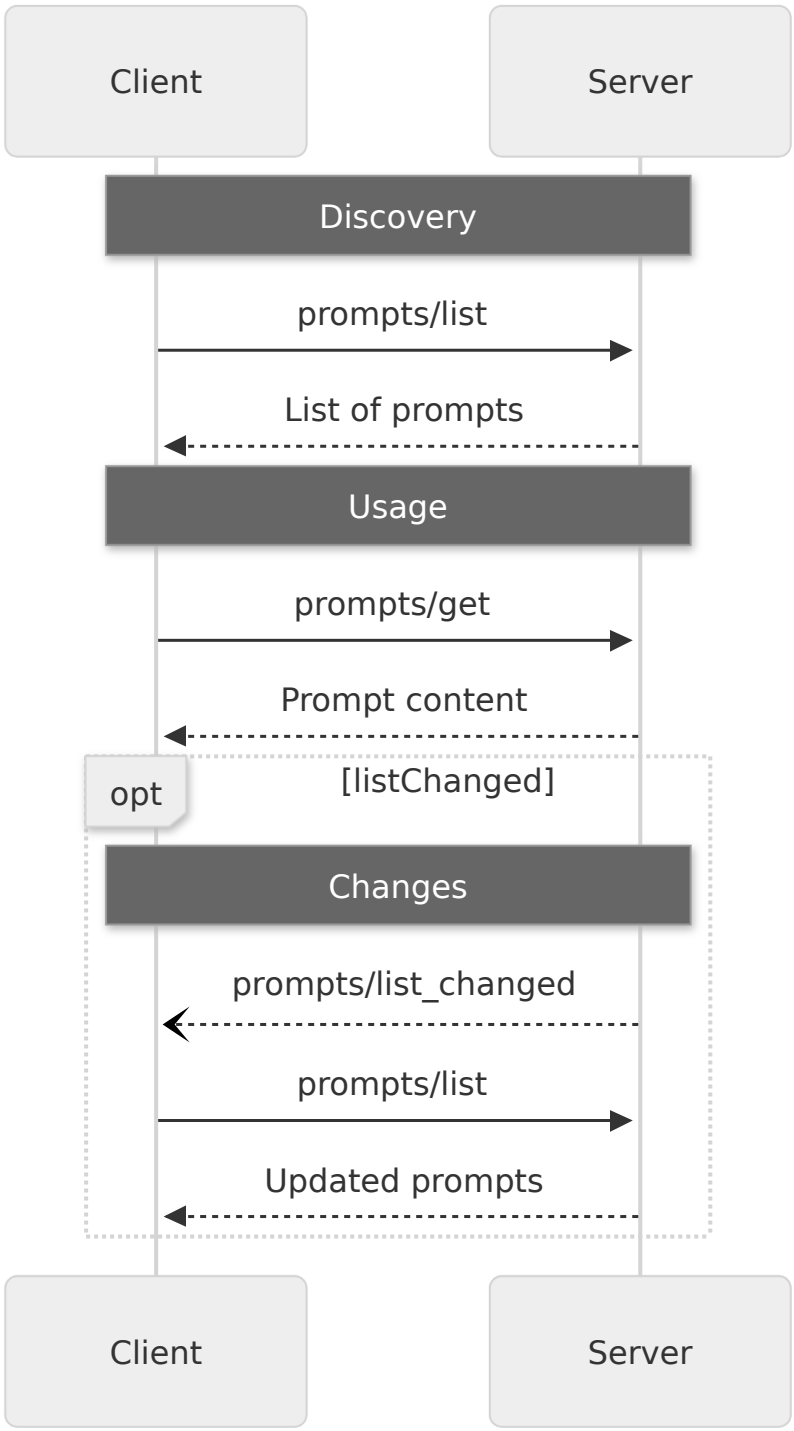
```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "description": "Code review prompt",
    "messages": [
      {
        "role": "user",
        "content": {
          "type": "text",
          "text": "Please review this Python code:\ndef hello():\n
print('world')"
        }
      }
    ]
  }
}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

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
- `icons`: Optional array of icons for display in user interfaces
- `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

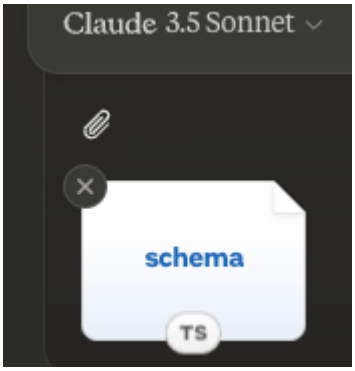
The Model Context Protocol (MCP) provides a standardized way for servers to expose resources to clients. Resources allow servers to share data that provides context to language models, such as files, database schemas, or application-specific information. Each resource is uniquely identified by a URI.

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",
        "icons": [
          {
            "src": "https://example.com/rust-file-icon.png",
            "mimeType": "image/png",
            "sizes": ["48x48"]
          }
        ]
      }
    ],
    "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.

Request:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "resources/templates/list"
}
```

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",
        "icons": [
          {
            "src": "https://example.com/folder-icon.png",
            "mimeType": "image/png",
            "sizes": ["48x48"]
          }
        ]
      }
    ]
  }
}
```

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:

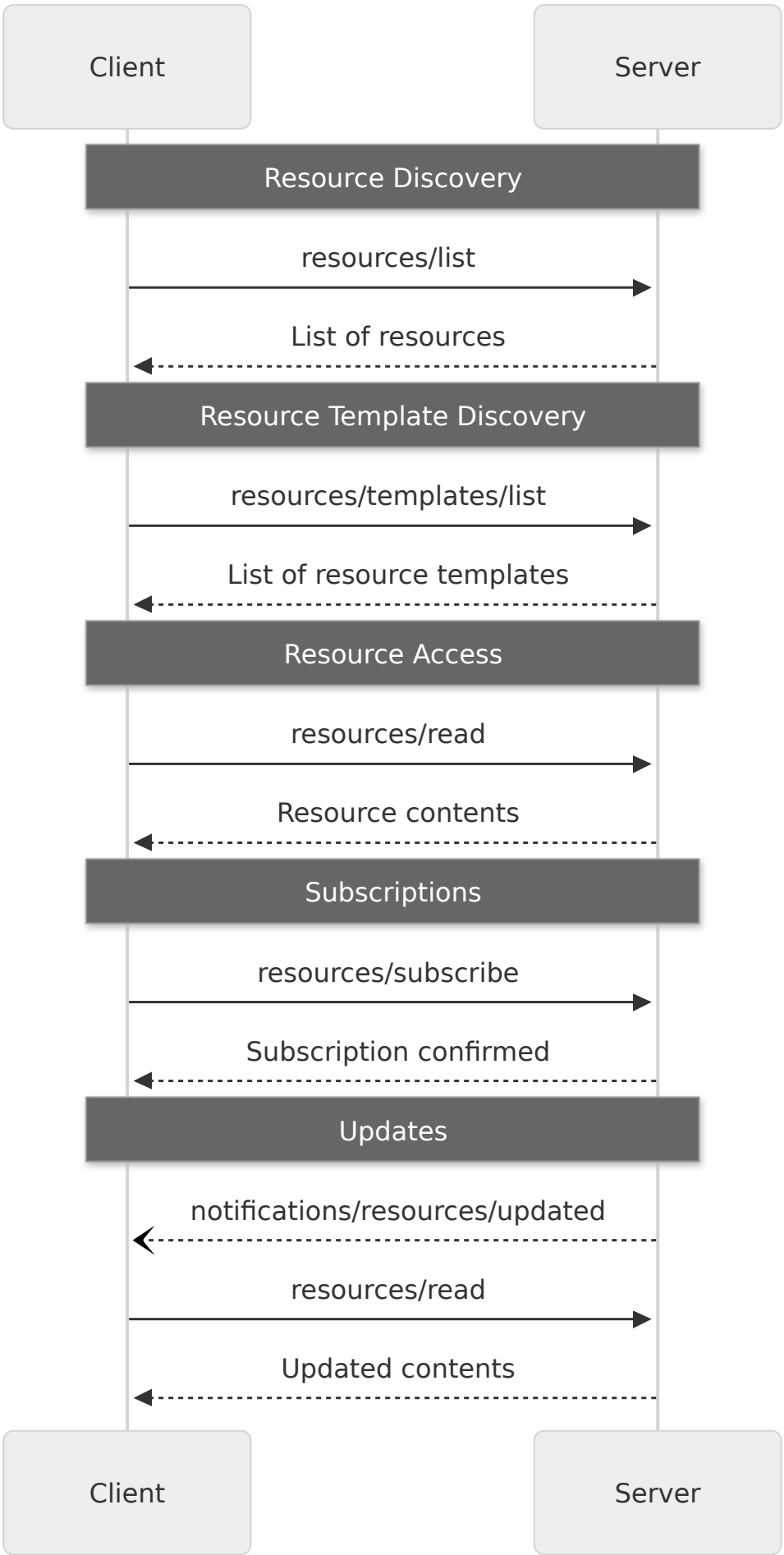
Subscribe Request:

```
{
  "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


Client
Server
Resource Discovery
resources/list
List of resources
Resource Template Discovery
resources/templates/list
List of resource templates
Resource Access
resources/read
Resource contents
Subscriptions
resources/subscribe
Subscription confirmed
Updates
notifications/resources/updated
resources/read
Updated contents
Client
Server

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
- `icons`: Optional array of icons for display in user interfaces
- `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"]
        },
        "icons": [
          {
            "src": "https://example.com/weather-icon.png",
            "mimeType": "image/png",
            "sizes": ["48x48"]
          }
        ],
        "execution": {
          "taskSupport": "optional"
        }
      }
    ],
    "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:

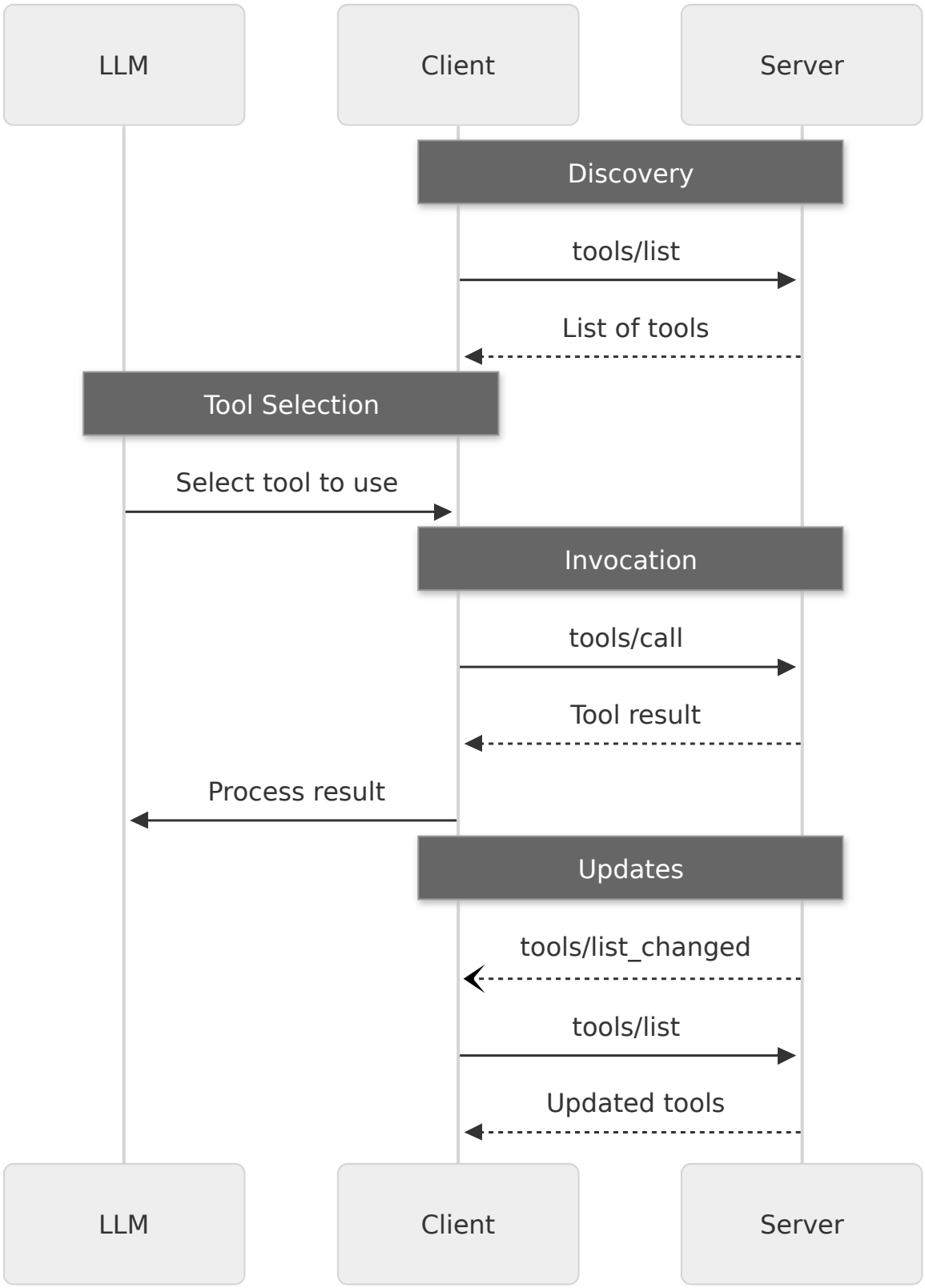
```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "content": [
      {
        "type": "text",
        "text": "Current weather in New York:\nTemperature: 72°F\nConditions:
Partly cloudy"
      }
    ],
    "isError": false
  }
}
```

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
- `icons`: Optional array of icons for display in user interfaces
- `inputSchema`: JSON Schema defining expected parameters
  - Follows the JSON Schema usage guidelines
  - Defaults to 2020-12 if no `$schema` field is present
  - **MUST** be a valid JSON Schema object (not `null`)
  - For tools with no parameters, use one of these valid approaches:
    - `{ "type": "object", "additionalProperties": false }` - **Recommended**: explicitly accepts only empty objects
    - `{ "type": "object" }` - accepts any object (including with properties)
- `outputSchema`: Optional JSON Schema defining expected output structure
  - Follows the JSON Schema usage guidelines
  - Defaults to 2020-12 if no `$schema` field is present
- `annotations`: Optional properties describing tool behavior
- `execution`: Optional object describing execution-related properties
  - `taskSupport`: Indicates whether this tool supports task-augmented execution. Values: `"forbidden"` (default), `"optional"`, or `"required"`

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

Tool Names

- Tool names **SHOULD** be between 1 and 128 characters in length (inclusive).
- Tool names **SHOULD** be considered case-sensitive.
- The following **SHOULD** be the only allowed characters: uppercase and lowercase ASCII letters (A-Z, a-z), digits (0-9), underscore (_), hyphen (-), and dot (.)
- Tool names **SHOULD NOT** contain spaces, commas, or other special characters.
- Tool names **SHOULD** be unique within a server.
- Example valid tool names:
  - getUser
  - DATA_EXPORT_v2
  - admin.tools.list

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
  }
}
```

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"
}
```

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

Schema Examples

Tool with default 2020-12 schema:

```
{
  "name": "calculate_sum",
  "description": "Add two numbers",
  "inputSchema": {
    "type": "object",
    "properties": {
      "a": { "type": "number" },
      "b": { "type": "number" }
    },
    "required": ["a", "b"]
  }
}
```

Tool with explicit draft-07 schema:

```
{
  "name": "calculate_sum",
  "description": "Add two numbers",
  "inputSchema": {
    "$schema": "http://json-schema.org/draft-07/schema#",
    "type": "object",
    "properties": {
      "a": { "type": "number" },
      "b": { "type": "number" }
    },
    "required": ["a", "b"]
  }
}
```

Tool with no parameters:

```
{
  "name": "get_current_time",
  "description": "Returns the current server time",
  "inputSchema": {
    "type": "object",
    "additionalProperties": false
  }
}
```

Error Handling

Tools use two error reporting mechanisms:

1. **Protocol Errors**: Standard JSON-RPC errors for issues like:
    - Unknown tools
    - Malformed requests (requests that fail to satisfy CallToolRequest schema)
    - Server errors
2. **Tool Execution Errors**: Reported in tool results with `isError: true`:
    - API failures
    - Input validation errors (e.g., date in wrong format, value out of range)
    - Business logic errors

Tool Execution Errors contain actionable feedback that language models can use to self-correct and retry with adjusted parameters. **Protocol Errors** indicate issues with the request structure itself that models are less likely to be able to fix. Clients **SHOULD** provide tool execution errors to

language models to enable self-correction. Clients **MAY** provide protocol errors to language models, though these are less likely to result in successful recovery.

Example protocol error:

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

Example tool execution error (input validation):

```
{
  "jsonrpc": "2.0",
  "id": 4,
  "result": {
    "content": [
      {
        "type": "text",
        "text": "Invalid departure date: must be in the future. Current date
is 08/08/2025."
      }
    ],
    "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 autocompletion suggestions for the arguments of prompts and resource templates. When users are filling in argument values for a specific prompt (identified by name) or resource template (identified by URI), servers can provide contextual suggestions.

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:

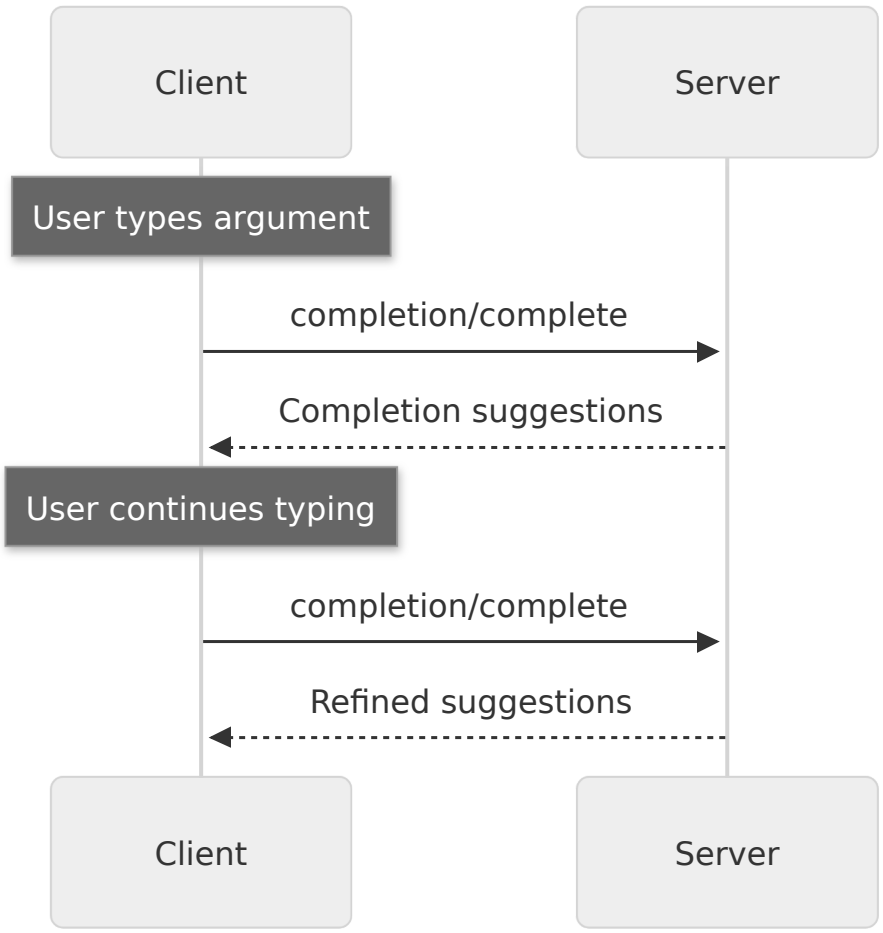
Type	Description	Example
`ref/prompt`	References a prompt by name	`{"type": "ref/prompt", "name": "code_review"}`
`ref/resource`	References a resource URI	`{"type": "ref/resource", "uri": "file:///{path}"}`

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:

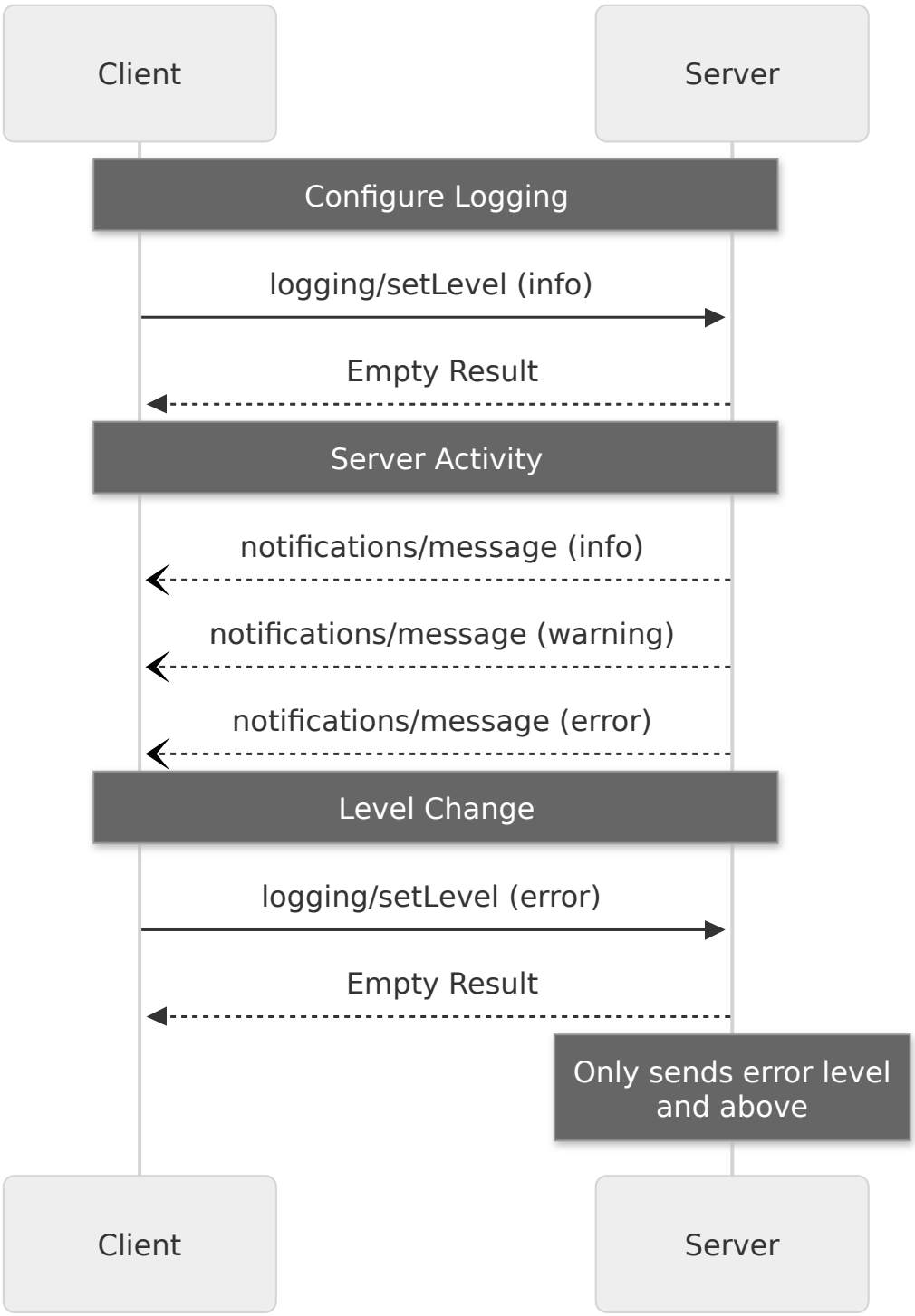
```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "logging/setLevel",
  "params": {
    "level": "info"
  }
}
```

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

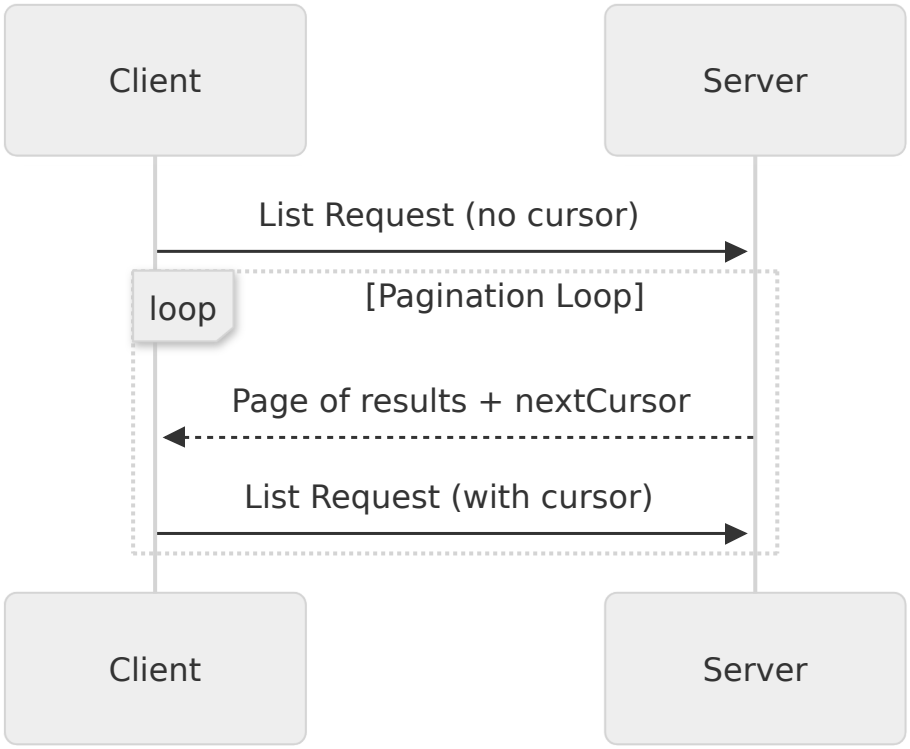
```
{
  "jsonrpc": "2.0",
  "id": "123",
  "result": {
    "resources": [...],
    "nextCursor": "eyJwYWdlIjogM30="
  }
}
```

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

`JSONRPCErrorResponse`

```
interface JSONRPCErrorResponse {
  jsonrpc: "2.0";
  id?: RequestId;
  error: Error;
}
```

A response to a request that indicates an error occurred.

> **`jsonrpc: "2.0"`**

> **`id?: RequestId`**

> **`error: Error`**

`JSONRPCMessage`

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

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?: { [key: string]: any };
  jsonrpc: "2.0";
}
```

A notification which does not expect a response.

> **`method: string`**
> Inherited from Notification.method

> **`params?: { [key: string]: any }`**
> Inherited from Notification.params

> **jsonrpc: "2.0"**

JSONRPCRequest

```
interface JSONRPCRequest {
  method: string;
  params?: { [key: string]: any };
  jsonrpc: "2.0";
  id: RequestId;
}
```

A request that expects a response.

> **method: string**
> Inherited from Request.method

> **params?: { [key: string]: any }**
> Inherited from Request.params

> **jsonrpc: "2.0"**

> **id: RequestId**

JSONRPCResponse

```
JSONRPCResponse: JSONRPCResultResponse | JSONRPCErrorResponse
```

A response to a request, containing either the result or error.

JSONRPCResultResponse

```
interface JSONRPCResultResponse {
  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 audience 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.

Error

```ts
interface Error {
  code: number;
  message: string;
  data?: unknown;
}
```

> **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.

> **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.).

Icon

```ts
interface Icon {
  src: string;
  mimeType?: string;
  sizes?: string[];
  theme?: "light" | "dark";
}
```

An optionally-sized icon that can be displayed in a user interface.

> **src: string**
>
> A standard URI pointing to an icon resource. May be an HTTP/HTTPS URL or a `data:` URI with Base64-encoded image data.
>
> Consumers SHOULD takes steps to ensure URLs serving icons are from the same domain as the client/server or a trusted domain.
>
> Consumers SHOULD take appropriate precautions when consuming SVGs as they can contain executable JavaScript.

> **mimeType?: string**
>
> Optional MIME type override if the source MIME type is missing or generic. For example: `"image/png"`, `"image/jpeg"`, or `"image/svg+xml"`.

> **sizes?: string[]**

> Optional array of strings that specify sizes at which the icon can be used. Each string should be in WxH format (e.g., `"48x48"`, `"96x96"`) or `"any"` for scalable formats like SVG.
>
> If not provided, the client should assume that the icon can be used at any size.

> **`theme?: "light" | "dark"`**
>
> Optional specifier for the theme this icon is designed for. `light` indicates the icon is designed to be used with a light background, and `dark` indicates the icon is designed to be used with a dark background.
>
> If not provided, the client should assume the icon can be used with any theme.

`LoggingLevel`

```
LoggingLevel:
  | "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.

> **mimeType: 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`

```ts
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`

```ts
ContentBlock:
  | TextContent
  | ImageContent
  | AudioContent
  | ResourceLink
  | EmbeddedResource
```

`EmbeddedResource`

```ts
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`

```ts
interface ResourceLink {
  icons?: Icon[];
  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.

> **`icons?: Icon[]`**
>
> Optional set of sized icons that the client can display in a user interface.
>
> Clients that support rendering icons MUST support at least the following MIME types:
>
> - `image/png` - PNG images (safe, universal compatibility)
> - `image/jpeg` (and `image/jpg`) - JPEG images (safe, universal compatibility)
>
> Clients that support rendering icons SHOULD also support:
>
> - `image/svg+xml` - SVG images (scalable but requires security precautions)
> - `image/webp` - WebP images (modern, efficient format)
>
> Inherited from Resource.icons

> **`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 {
  jsonrpc: "2.0";
  id: RequestId;
  method: "completion/complete";
  params: CompleteRequestParams;
}
```

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

> **jsonrpc: "2.0"**
> Inherited from JSONRPCRequest.jsonrpc

> **id: RequestId**
> Inherited from JSONRPCRequest.id

> **method: "completion/complete"**
> Overrides JSONRPCRequest.method

> **params: CompleteRequestParams**
> Overrides JSONRPCRequest.params

`CompleteRequestParams`

```
interface CompleteRequestParams {
  _meta?: { progressToken?: ProgressToken; [key: string]: unknown };
  ref: PromptReference | ResourceTemplateReference;
  argument: { name: string; value: string };
  context?: { arguments?: { [key: string]: string } };
}
```

Parameters for a `completion/complete` request.

> **_meta?: { progressToken?: ProgressToken; [key: string]: unknown }**
>
> See General fields: `_meta` for notes on `_meta` usage.
>
> Type Declaration
>
> - [key: string]: unknown
> - `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 RequestParams._meta

> **ref: PromptReference | ResourceTemplateReference**

> **argument: { name: string; value: string }**

> The argument's information
>
> Type Declaration
>
> - name: string
>
> The name of the argument
>
> - value: string
>
> The value of the argument to use for completion matching.

> **context?: { arguments?: { [key: string]: string } }**
>
> Additional, optional context for completions
>
> Type Declaration
>
> - `Optional` arguments?: { [key: string]: string }
>
> Previously-resolved variables in a URI template or prompt.

`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`

```ts
interface ElicitRequest {
  jsonrpc: "2.0";
  id: RequestId;
  method: "elicitation/create";
  params: ElicitRequestParams;
}
```

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

> **`jsonrpc: "2.0"`**
> Inherited from JSONRPCRequest.jsonrpc

> **`id: RequestId`**
> Inherited from JSONRPCRequest.id

> **`method: "elicitation/create"`**
> Overrides JSONRPCRequest.method

> **`params: ElicitRequestParams`**
> Overrides JSONRPCRequest.params

`ElicitRequestParams`

```ts
ElicitRequestParams: ElicitRequestFormParams | ElicitRequestURLParams
```

The parameters for a request to elicit additional information from the user via the client.

`ElicitResult`

```ts
interface ElicitResult {
  _meta?: { [key: string]: unknown };
  action: "accept" | "decline" | "cancel";
  content?: { [key: string]: string | number | boolean | string[] };
  [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 decline the action
- "cancel": User dismissed without making an explicit choice

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

The submitted form data, only present when action is "accept" and mode was "form". Contains values matching the requested schema. Omitted for out-of-band mode responses.

BooleanSchema

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

type: "boolean"

title?: string

description?: string

default?: boolean

ElicitRequestFormParams

```
interface ElicitRequestFormParams {
  task?: TaskMetadata;
  _meta?: { progressToken?: ProgressToken; [key: string]: unknown };
  mode?: "form";
  message: string;
  requestedSchema: {
    $schema?: string;
    type: "object";
    properties: { [key: string]: PrimitiveSchemaDefinition };
    required?: string[];
  };
}
```

The parameters for a request to elicit non-sensitive information from the user via a form in the client.

task?: TaskMetadata

If specified, the caller is requesting task-augmented execution for this request. The request will return a CreateTaskResult immediately, and the actual result can be retrieved later via

tasks/result.

Task augmentation is subject to capability negotiation - receivers MUST declare support for task augmentation of specific request types in their capabilities.

Inherited from TaskAugmentedRequestParams.task

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

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

Type Declaration

- [key: string]: unknown
- `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 TaskAugmentedRequestParams._meta

`mode?: "form"`

The elicitation mode.

`message: string`

The message to present to the user describing what information is being requested.

`requestedSchema: { $schema?: string; type: "object"; properties: { [key: string]: PrimitiveSchemaDefinition }; required?: string[]; }`

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

`ElicitRequestURLParams`

```
interface ElicitRequestURLParams {
  task?: TaskMetadata;
  _meta?: { progressToken?: ProgressToken; [key: string]: unknown };
  mode: "url";
  message: string;
  elicitationId: string;
  url: string;
}
```

The parameters for a request to elicit information from the user via a URL in the client.

`task?: TaskMetadata`

If specified, the caller is requesting task-augmented execution for this request. The request will return a CreateTaskResult immediately, and the actual result can be retrieved later via

tasks/result.

Task augmentation is subject to capability negotiation - receivers MUST declare support for task augmentation of specific request types in their capabilities.

Inherited from TaskAugmentedRequestParams.task

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

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

Type Declaration

- [key: string]: unknown
- `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 TaskAugmentedRequestParams._meta

`mode: "url"`

The elicitation mode.

`message: string`

The message to present to the user explaining why the interaction is needed.

`elicitationId: string`

The ID of the elicitation, which must be unique within the context of the server. The client MUST treat this ID as an opaque value.

`url: string`

The URL that the user should navigate to.

`EnumSchema`

```
EnumSchema:
  | SingleSelectEnumSchema
  | MultiSelectEnumSchema
  | LegacyTitledEnumSchema
```

`LegacyTitledEnumSchema`

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

Use TitledSingleSelectEnumSchema instead. This interface will be removed in a future version.

> **type: "string"**

> **title?: string**

> **description?: string**

> **enum: string[]**

> **enumNames?: string[]**
>
> (Legacy) Display names for enum values. Non-standard according to JSON schema 2020-12.

> **default?: string**

`MultiSelectEnumSchema`

```
MultiSelectEnumSchema:
  | UntitledMultiSelectEnumSchema
  | TitledMultiSelectEnumSchema
```

`NumberSchema`

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

> **type: "number" | "integer"**

> **title?: string**

> **description?: string**

> **minimum?: number**

> **maximum?: number**

> **default?: number**

PrimitiveSchemaDefinition

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

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

SingleSelectEnumSchema

```
SingleSelectEnumSchema:
  | UntitledSingleSelectEnumSchema
  | TitledSingleSelectEnumSchema
```

StringSchema

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

> **type: "string"**

> **title?: string**

> **description?: string**

> **minLength?: number**

> **maxLength?: number**

> **format?: "uri" | "email" | "date" | "date-time"**

> **default?: string**

`TitledMultiSelectEnumSchema`

```
interface TitledMultiSelectEnumSchema {
  type: "array";
  title?: string;
  description?: string;
  minItems?: number;
  maxItems?: number;
  items: { anyOf: { const: string; title: string }[] };
  default?: string[];
}
```

Schema for multiple-selection enumeration with display titles for each option.

> **`type: "array"`**

> **`title?: string`**
>
> Optional title for the enum field.

> **`description?: string`**
>
> Optional description for the enum field.

> **`minItems?: number`**
>
> Minimum number of items to select.

> **`maxItems?: number`**
>
> Maximum number of items to select.

> **`items: { anyOf: { const: string; title: string }[] }`**
>
> Schema for array items with enum options and display labels.
>
> Type Declaration
>
> - anyOf: { const: string; title: string }[]
>
> Array of enum options with values and display labels.

> **`default?: string[]`**
>
> Optional default value.

`TitledSingleSelectEnumSchema`

```ts
interface TitledSingleSelectEnumSchema {
  type: "string";
  title?: string;
  description?: string;
  oneOf: { const: string; title: string }[];
  default?: string;
}
```

Schema for single-selection enumeration with display titles for each option.

> **type: "string"**

> **title?: string**
>
> Optional title for the enum field.

> **description?: string**
>
> Optional description for the enum field.

> **oneOf: { const: string; title: string }[]**
>
> Array of enum options with values and display labels.
>
> Type Declaration
>
> - const: string
>
>   The enum value.
>
> - title: string
>
>   Display label for this option.

> **default?: string**
>
> Optional default value.

`UntitledMultiSelectEnumSchema`

```ts
interface UntitledMultiSelectEnumSchema {
  type: "array";
  title?: string;
  description?: string;
  minItems?: number;
  maxItems?: number;
  items: { type: "string"; enum: string[] };
  default?: string[];
}
```

Schema for multiple-selection enumeration without display titles for options.

> **type: "array"**

> **title?: string**
>
> Optional title for the enum field.

> **description?: string**
>
> Optional description for the enum field.

> **minItems?: number**
>
> Minimum number of items to select.

> **maxItems?: number**
>
> Maximum number of items to select.

> **items: { type: "string"; enum: string[] }**
>
> Schema for the array items.
>
> Type Declaration
>
> - type: "string"
> - enum: string[]
>
> Array of enum values to choose from.

> **default?: string[]**
>
> Optional default value.

`UntitledSingleSelectEnumSchema`

```
interface UntitledSingleSelectEnumSchema {
  type: "string";
  title?: string;
  description?: string;
  enum: string[];
  default?: string;
}
```

Schema for single-selection enumeration without display titles for options.

> **type: "string"**

> **title?: string**
>
> Optional title for the enum field.

> **description?: string**

> Optional description for the enum field.

> **enum: string[]**
>
> Array of enum values to choose from.

> **default?: string**
>
> Optional default value.

initialize

InitializeRequest

```
interface InitializeRequest {
  jsonrpc: "2.0";
  id: RequestId;
  method: "initialize";
  params: InitializeRequestParams;
}
```

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

> **jsonrpc: "2.0"**
> Inherited from JSONRPCRequest.jsonrpc

> **id: RequestId**
> Inherited from JSONRPCRequest.id

> **method: "initialize"**
> Overrides JSONRPCRequest.method

> **params: InitializeRequestParams**
> Overrides JSONRPCRequest.params

InitializeRequestParams

```
interface InitializeRequestParams {
  _meta?: { progressToken?: ProgressToken; [key: string]: unknown };
  protocolVersion: string;
  capabilities: ClientCapabilities;
  clientInfo: Implementation;
}
```

Parameters for an `initialize` request.

> **_meta?: { progressToken?: ProgressToken; [key: string]: unknown }**
>
> See General fields: `_meta` for notes on `_meta` usage.
>
> Type Declaration
>
> - [key: string]: unknown
> - `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 RequestParams._meta

> **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**

`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?: { context?: object; tools?: object };
  elicitation?: { form?: object; url?: object };
  tasks?: {
    list?: object;
    cancel?: object;
    requests?: {
      sampling?: { createMessage?: object };
      elicitation?: { create?: 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?: { context?: object; tools?: object }`**
>
> Present if the client supports sampling from an LLM.
>
> Type Declaration
>
> - `Optional` context?: object
>
> Whether the client supports context inclusion via includeContext parameter. If not declared, servers SHOULD only use `includeContext: "none"` (or omit it).

- `Optional` tools?: object

  Whether the client supports tool use via tools and toolChoice parameters.

elicitation?: { form?: object; url?: object }

Present if the client supports elicitation from the server.

tasks?: { list?: object; cancel?: object; requests?: { sampling?: { createMessage?: object }; elicitation?: { create?: object }; }; }

Present if the client supports task-augmented requests.

Type Declaration

- `Optional` list?: object

  Whether this client supports tasks/list.

- `Optional` cancel?: object

  Whether this client supports tasks/cancel.

- `Optional` requests?: { sampling?: { createMessage?: object }; elicitation?: { create?: object } }

  Specifies which request types can be augmented with tasks.

  - `Optional` sampling?: { createMessage?: object }

    Task support for sampling-related requests.

    - `Optional` createMessage?: object

      Whether the client supports task-augmented sampling/createMessage requests.

  - `Optional` elicitation?: { create?: object }

    Task support for elicitation-related requests.

    - `Optional` create?: object

      Whether the client supports task-augmented elicitation/create requests.

Implementation

```ts
interface Implementation {
  icons?: Icon[];
  name: string;
  title?: string;
  version: string;
  description?: string;
  websiteUrl?: string;
}
```

Describes the MCP implementation.

> **icons?: Icon[]**
>
> Optional set of sized icons that the client can display in a user interface.
>
> Clients that support rendering icons MUST support at least the following MIME types:
>
> - `image/png` - PNG images (safe, universal compatibility)
> - `image/jpeg` (and `image/jpg`) - JPEG images (safe, universal compatibility)
>
> Clients that support rendering icons SHOULD also support:
>
> - `image/svg+xml` - SVG images (scalable but requires security precautions)
> - `image/webp` - WebP images (modern, efficient format)
>
> Inherited from Icons.icons

> **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**

> **description?: string**
>
> An optional human-readable description of what this implementation does.
>
> This can be used by clients or servers to provide context about their purpose and capabilities. For example, a server might describe the types of resources or tools it provides, while a client might describe its intended use case.

> **websiteUrl?: string**

> An optional URL of the website for this implementation.

`ServerCapabilities`

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

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.

tasks?: { list?: object; cancel?: object; requests?: { tools?: { call?: object } }; }

Present if the server supports task-augmented requests.

Type Declaration

- `Optional` list?: object

  Whether this server supports tasks/list.

- `Optional` cancel?: object

  Whether this server supports tasks/cancel.

- `Optional` requests?: { tools?: { call?: object } }

  Specifies which request types can be augmented with tasks.

  - `Optional` tools?: { call?: object }

    Task support for tool-related requests.

    - `Optional` call?: object

      Whether the server supports task-augmented tools/call requests.

logging/setLevel

SetLevelRequest

```
interface SetLevelRequest {
  jsonrpc: "2.0";
  id: RequestId;
  method: "logging/setLevel";
  params: SetLevelRequestParams;
}
```

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

> **jsonrpc: "2.0"**
> Inherited from JSONRPCRequest.jsonrpc

> **id: RequestId**
> Inherited from JSONRPCRequest.id

> **method: "logging/setLevel"**
> Overrides JSONRPCRequest.method

> **params: SetLevelRequestParams**
> Overrides JSONRPCRequest.params

SetLevelRequestParams

```
interface SetLevelRequestParams {
  _meta?: { progressToken?: ProgressToken; [key: string]: unknown };
  level: LoggingLevel;
}
```

Parameters for a `logging/setLevel` request.

> **_meta?: { progressToken?: ProgressToken; [key: string]: unknown }**
>
> See General fields: `_meta` for notes on `_meta` usage.
>
> Type Declaration
>
> - [key: string]: unknown
> - `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 RequestParams._meta

> **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.

notifications/cancelled

CancelledNotification

```ts
interface CancelledNotification {
  jsonrpc: "2.0";
  method: "notifications/cancelled";
  params: CancelledNotificationParams;
}
```

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.

For task cancellation, use the `tasks/cancel` request instead of this notification.

> **`jsonrpc: "2.0"`**
> Inherited from JSONRPCNotification.jsonrpc

> **`method: "notifications/cancelled"`**
> Overrides JSONRPCNotification.method

> **`params: CancelledNotificationParams`**
> Overrides JSONRPCNotification.params

`CancelledNotificationParams`

```ts
interface CancelledNotificationParams {
  _meta?: { [key: string]: unknown };
  requestId?: RequestId;
  reason?: string;
}
```

Parameters for a `notifications/cancelled` notification.

> **`_meta?: { [key: string]: unknown }`**
>
> See General fields: `_meta` for notes on `_meta` usage.
>
> Inherited from NotificationParams._meta

> **`requestId?: RequestId`**
>
> The ID of the request to cancel.
>
> This MUST correspond to the ID of a request previously issued in the same direction. This MUST be provided for cancelling non-task requests. This MUST NOT be used for cancelling tasks (use the `tasks/cancel` request instead).

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

notifications/initialized

InitializedNotification

```
interface InitializedNotification {
  jsonrpc: "2.0";
  method: "notifications/initialized";
  params?: NotificationParams;
}
```

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

> **jsonrpc: "2.0"**
> Inherited from JSONRPCNotification.jsonrpc

> **method: "notifications/initialized"**
> Overrides JSONRPCNotification.method

> **params?: NotificationParams**
> Overrides JSONRPCNotification.params

notifications/tasks/status

TaskStatusNotification

```
interface TaskStatusNotification {
  jsonrpc: "2.0";
  method: "notifications/tasks/status";
  params: TaskStatusNotificationParams;
}
```

An optional notification from the receiver to the requestor, informing them that a task's status has changed. Receivers are not required to send these notifications.

> **jsonrpc: "2.0"**
> Inherited from JSONRPCNotification.jsonrpc

> **method: "notifications/tasks/status"**
> Overrides JSONRPCNotification.method

> **params: TaskStatusNotificationParams**
> Overrides JSONRPCNotification.params

TaskStatusNotificationParams

```
TaskStatusNotificationParams: NotificationParams & Task
```

Parameters for a `notifications/tasks/status` notification.

notifications/message

LoggingMessageNotification

```
interface LoggingMessageNotification {
  jsonrpc: "2.0";
  method: "notifications/message";
  params: LoggingMessageNotificationParams;
}
```

JSONRPCNotification 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.

> **jsonrpc: "2.0"**
> Inherited from JSONRPCNotification.jsonrpc

> **method: "notifications/message"**
> Overrides JSONRPCNotification.method

> **params: LoggingMessageNotificationParams**
> Overrides JSONRPCNotification.params

LoggingMessageNotificationParams

```
interface LoggingMessageNotificationParams {
  _meta?: { [key: string]: unknown };
  level: LoggingLevel;
  logger?: string;
  data: unknown;
}
```

Parameters for a `notifications/message` notification.

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

> See General fields: `_meta` for notes on `_meta` usage.
>
> Inherited from NotificationParams._meta

> **`level: LoggingLevel`**
>
> The severity of this log message.

> **`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.

`notifications/progress`

`ProgressNotification`

```
interface ProgressNotification {
  jsonrpc: "2.0";
  method: "notifications/progress";
  params: ProgressNotificationParams;
}
```

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

> **`jsonrpc: "2.0"`**
> Inherited from JSONRPCNotification.jsonrpc

> **`method: "notifications/progress"`**
> Overrides JSONRPCNotification.method

> **`params: ProgressNotificationParams`**
> Overrides JSONRPCNotification.params

`ProgressNotificationParams`

```ts
interface ProgressNotificationParams {
  _meta?: { [key: string]: unknown };
  progressToken: ProgressToken;
  progress: number;
  total?: number;
  message?: string;
}
```

Parameters for a `notifications/progress` notification.

> **`_meta?: { [key: string]: unknown }`**
>
> See General fields: `_meta` for notes on `_meta` usage.
>
> Inherited from NotificationParams._meta

> **`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.

> **`total?: number`**
>
> Total number of items to process (or total progress required), if known.

> **`message?: string`**
>
> An optional message describing the current progress.

`notifications/prompts/list_changed`

`PromptListChangedNotification`

```ts
interface PromptListChangedNotification {
  jsonrpc: "2.0";
  method: "notifications/prompts/list_changed";
  params?: NotificationParams;
}
```

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.

> **`jsonrpc: "2.0"`**

> Inherited from JSONRPCNotification.jsonrpc

> **method: "notifications/prompts/list_changed"**
> Overrides JSONRPCNotification.method

> **params?: NotificationParams**
> Overrides JSONRPCNotification.params

notifications/resources/list_changed

ResourceListChangedNotification

```
interface ResourceListChangedNotification {
  jsonrpc: "2.0";
  method: "notifications/resources/list_changed";
  params?: NotificationParams;
}
```

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.

> **jsonrpc: "2.0"**
> Inherited from JSONRPCNotification.jsonrpc

> **method: "notifications/resources/list_changed"**
> Overrides JSONRPCNotification.method

> **params?: NotificationParams**
> Overrides JSONRPCNotification.params

notifications/resources/updated

ResourceUpdatedNotification

```
interface ResourceUpdatedNotification {
  jsonrpc: "2.0";
  method: "notifications/resources/updated";
  params: ResourceUpdatedNotificationParams;
}
```

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.

> **jsonrpc: "2.0"**
> Inherited from JSONRPCNotification.jsonrpc

> **method: "notifications/resources/updated"**
> Overrides JSONRPCNotification.method

> **params: ResourceUpdatedNotificationParams**
> Overrides JSONRPCNotification.params

ResourceUpdatedNotificationParams

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

Parameters for a `notifications/resources/updated` notification.

> **_meta?: { [key: string]: unknown }**
>
> See General fields: `_meta` for notes on `_meta` usage.
>
> Inherited from NotificationParams._meta

> **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.

notifications/roots/list_changed

RootsListChangedNotification

```
interface RootsListChangedNotification {
  jsonrpc: "2.0";
  method: "notifications/roots/list_changed";
  params?: NotificationParams;
}
```

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.

> **jsonrpc: "2.0"**
> Inherited from JSONRPCNotification.jsonrpc

> **method: "notifications/roots/list_changed"**
> Overrides JSONRPCNotification.method

> **params?: NotificationParams**
> Overrides JSONRPCNotification.params

notifications/tools/list_changed

ToolListChangedNotification

```
interface ToolListChangedNotification {
  jsonrpc: "2.0";
  method: "notifications/tools/list_changed";
  params?: NotificationParams;
}
```

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.

> **jsonrpc: "2.0"**
> Inherited from JSONRPCNotification.jsonrpc

> **method: "notifications/tools/list_changed"**
> Overrides JSONRPCNotification.method

> **params?: NotificationParams**
> Overrides JSONRPCNotification.params

notifications/elicitation/complete

ElicitationCompleteNotification

```
interface ElicitationCompleteNotification {
  jsonrpc: "2.0";
  method: "notifications/elicitation/complete";
  params: { elicitationId: string };
}
```

An optional notification from the server to the client, informing it of a completion of a out-of-band elicitation request.

> **jsonrpc: "2.0"**
> Inherited from JSONRPCNotification.jsonrpc

> **method: "notifications/elicitation/complete"**
> Overrides JSONRPCNotification.method

> **params: { elicitationId: string }**

> Type Declaration
>
> - elicitationId: string
>
>   The ID of the elicitation that completed.
>
> Overrides JSONRPCNotification.params

ping

PingRequest

```
interface PingRequest {
  jsonrpc: "2.0";
  id: RequestId;
  method: "ping";
  params?: RequestParams;
}
```

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.

> **jsonrpc: "2.0"**
> Inherited from JSONRPCRequest.jsonrpc
>
> **id: RequestId**
> Inherited from JSONRPCRequest.id
>
> **method: "ping"**
> Overrides JSONRPCRequest.method
>
> **params?: RequestParams**
> Overrides JSONRPCRequest.params

tasks

CreateTaskResult

```
interface CreateTaskResult {
  _meta?: { [key: string]: unknown };
  task: Task;
  [key: string]: unknown;
}
```

A response to a task-augmented request.

> **_meta?: { [key: string]: unknown }**
>
> See General fields: `_meta` for notes on `_meta` usage.
>
> Inherited from Result._meta

> **task: Task**

`RelatedTaskMetadata`

```ts
interface RelatedTaskMetadata {
  taskId: string;
}
```

Metadata for associating messages with a task. Include this in the `_meta` field under the key `io.modelcontextprotocol/related-task`.

> **taskId: string**
>
> The task identifier this message is associated with.

`Task`

```ts
interface Task {
  taskId: string;
  status: TaskStatus;
  statusMessage?: string;
  createdAt: string;
  lastUpdatedAt: string;
  ttl: number | null;
  pollInterval?: number;
}
```

Data associated with a task.

> **taskId: string**
>
> The task identifier.

> **status: TaskStatus**
>
> Current task state.

> **statusMessage?: string**
>
> Optional human-readable message describing the current task state. This can provide context for any status, including:
>
> - Reasons for "cancelled" status

> - Summaries for "completed" status
> - Diagnostic information for "failed" status (e.g., error details, what went wrong)

> **`createdAt: string`**
>
> ISO 8601 timestamp when the task was created.

> **`lastUpdatedAt: string`**
>
> ISO 8601 timestamp when the task was last updated.

> **`ttl: number | null`**
>
> Actual retention duration from creation in milliseconds, null for unlimited.

> **`pollInterval?: number`**
>
> Suggested polling interval in milliseconds.

`TaskMetadata`

```ts
interface TaskMetadata {
  ttl?: number;
}
```

Metadata for augmenting a request with task execution. Include this in the `task` field of the request parameters.

> **`ttl?: number`**
>
> Requested duration in milliseconds to retain task from creation.

`TaskStatus`

```ts
TaskStatus: "working" | "input_required" | "completed" | "failed" |
"cancelled"
```

The status of a task.

`tasks/get`

`GetTaskRequest`

```
interface GetTaskRequest {
  jsonrpc: "2.0";
  id: RequestId;
  method: "tasks/get";
  params: { taskId: string };
}
```

A request to retrieve the state of a task.

> **jsonrpc: "2.0"**
> Inherited from JSONRPCRequest.jsonrpc

> **id: RequestId**
> Inherited from JSONRPCRequest.id

> **method: "tasks/get"**
> Overrides JSONRPCRequest.method

> **params: { taskId: string }**
> Type Declaration
>
> - taskId: string
>
> The task identifier to query.
>
> Overrides JSONRPCRequest.params

`GetTaskResult`

```
GetTaskResult: Result & Task
```

The response to a tasks/get request.

`tasks/result`

`GetTaskPayloadRequest`

```
interface GetTaskPayloadRequest {
  jsonrpc: "2.0";
  id: RequestId;
  method: "tasks/result";
  params: { taskId: string };
}
```

A request to retrieve the result of a completed task.

> **jsonrpc: "2.0"**
> Inherited from JSONRPCRequest.jsonrpc

> **id: RequestId**
> Inherited from JSONRPCRequest.id

> **method: "tasks/result"**
> Overrides JSONRPCRequest.method

> **params: { taskId: string }**
> Type Declaration
>
> - taskId: string
>
> The task identifier to retrieve results for.
>
> Overrides JSONRPCRequest.params

GetTaskPayloadResult

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

The response to a tasks/result request. The structure matches the result type of the original request. For example, a tools/call task would return the CallToolResult structure.

> **_meta?: { [key: string]: unknown }**
>
> See General fields: `_meta` for notes on `_meta` usage.
>
> Inherited from Result._meta

tasks/list

ListTasksRequest

```
interface ListTasksRequest {
  jsonrpc: "2.0";
  id: RequestId;
  params?: PaginatedRequestParams;
  method: "tasks/list";
}
```

A request to retrieve a list of tasks.

> **jsonrpc: "2.0"**

> Inherited from PaginatedRequest.jsonrpc

> **id: RequestId**
> Inherited from PaginatedRequest.id

> **params?: PaginatedRequestParams**
> Inherited from PaginatedRequest.params

> **method: "tasks/list"**
> Overrides PaginatedRequest.method

ListTasksResult

```
interface ListTasksResult {
  _meta?: { [key: string]: unknown };
  nextCursor?: string;
  tasks: Task[];
  [key: string]: unknown;
}
```

The response to a tasks/list request.

> **_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

> **tasks: Task[]**

tasks/cancel

CancelTaskRequest

```
interface CancelTaskRequest {
  jsonrpc: "2.0";
  id: RequestId;
  method: "tasks/cancel";
  params: { taskId: string };
}
```

A request to cancel a task.

> **jsonrpc: "2.0"**
> Inherited from JSONRPCRequest.jsonrpc

> **id: RequestId**
> Inherited from JSONRPCRequest.id

> **method: "tasks/cancel"**
> Overrides JSONRPCRequest.method

> **params: { taskId: string }**
> Type Declaration
>
> - taskId: string
>
> The task identifier to cancel.
>
> Overrides JSONRPCRequest.params

CancelTaskResult

```
CancelTaskResult: Result & Task
```

The response to a tasks/cancel request.

prompts/get

GetPromptRequest

```
interface GetPromptRequest {
  jsonrpc: "2.0";
  id: RequestId;
  method: "prompts/get";
  params: GetPromptRequestParams;
}
```

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

> **jsonrpc: "2.0"**
> Inherited from JSONRPCRequest.jsonrpc

> **id: RequestId**
> Inherited from JSONRPCRequest.id

> **method: "prompts/get"**
> Overrides JSONRPCRequest.method

> **params: GetPromptRequestParams**

> Overrides JSONRPCRequest.params

`GetPromptRequestParams`

```
interface GetPromptRequestParams {
  _meta?: { progressToken?: ProgressToken; [key: string]: unknown };
  name: string;
  arguments?: { [key: string]: string };
}
```

Parameters for a `prompts/get` request.

> **`_meta?: { progressToken?: ProgressToken; [key: string]: unknown }`**
>
> See General fields: `_meta` for notes on `_meta` usage.
>
> Type Declaration
>
> - [key: string]: unknown
> - `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 RequestParams._meta

> **`name: string`**
>
> The name of the prompt or prompt template.

> **`arguments?: { [key: string]: string }`**
>
> Arguments to use for templating the prompt.

`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 {
  jsonrpc: "2.0";
  id: RequestId;
  params?: PaginatedRequestParams;
  method: "prompts/list";
}
```

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

> **jsonrpc: "2.0"**
> Inherited from PaginatedRequest.jsonrpc

> **id: RequestId**
> Inherited from PaginatedRequest.id

> **params?: PaginatedRequestParams**
> 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 {
  icons?: Icon[];
  name: string;
  title?: string;
  description?: string;
  arguments?: PromptArgument[];
  _meta?: { [key: string]: unknown };
}
```

A prompt or prompt template that the server offers.

> **icons?: Icon[]**
>
> Optional set of sized icons that the client can display in a user interface.
>
> Clients that support rendering icons MUST support at least the following MIME types:
>
> - `image/png` - PNG images (safe, universal compatibility)

- `image/jpeg` (and `image/jpg`) - JPEG images (safe, universal compatibility)

Clients that support rendering icons SHOULD also support:

- `image/svg+xml` - SVG images (scalable but requires security precautions)
- `image/webp` - WebP images (modern, efficient format)

Inherited from Icons.icons

`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`

```ts
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 {
  jsonrpc: "2.0";
  id: RequestId;
  params?: PaginatedRequestParams;
  method: "resources/list";
}
```

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

`jsonrpc: "2.0"`
Inherited from PaginatedRequest.jsonrpc

`id: RequestId`
Inherited from PaginatedRequest.id

`params?: PaginatedRequestParams`
Inherited from PaginatedRequest.params

`method: "resources/list"`
Overrides PaginatedRequest.method

`ListResourcesResult`

```ts
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`

```ts
interface Resource {
  icons?: Icon[];
  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.

> **`icons?: Icon[]`**
>
> Optional set of sized icons that the client can display in a user interface.
>
> Clients that support rendering icons MUST support at least the following MIME types:
>
> - `image/png` - PNG images (safe, universal compatibility)
> - `image/jpeg` (and `image/jpg`) - JPEG images (safe, universal compatibility)
>
> Clients that support rendering icons SHOULD also support:

- `image/svg+xml` - SVG images (scalable but requires security precautions)
- `image/webp` - WebP images (modern, efficient format)

Inherited from Icons.icons

`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 {
  jsonrpc: "2.0";
  id: RequestId;
  method: "resources/read";
  params: ReadResourceRequestParams;
}
```

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

> **jsonrpc: "2.0"**
> Inherited from JSONRPCRequest.jsonrpc

> **id: RequestId**
> Inherited from JSONRPCRequest.id

> **method: "resources/read"**
> Overrides JSONRPCRequest.method

> **params: ReadResourceRequestParams**
> Overrides JSONRPCRequest.params

ReadResourceRequestParams

```
interface ReadResourceRequestParams {
  _meta?: { progressToken?: ProgressToken; [key: string]: unknown };
  uri: string;
}
```

Parameters for a `resources/read` request.

> **_meta?: { progressToken?: ProgressToken; [key: string]: unknown }**
>
> See General fields: `_meta` for notes on `_meta` usage.
>
> Type Declaration
>
> - [key: string]: unknown
> - `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 ResourceRequestParams._meta

> **uri: string**
>
> The URI of the resource. The URI can use any protocol; it is up to the server how to interpret it.
>
> Inherited from ResourceRequestParams.uri

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 {
  jsonrpc: "2.0";
  id: RequestId;
  method: "resources/subscribe";
  params: SubscribeRequestParams;
}
```

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

> **jsonrpc: "2.0"**
>
> Inherited from JSONRPCRequest.jsonrpc

> **id: RequestId**
>
> Inherited from JSONRPCRequest.id

> **method: "resources/subscribe"**
>
> Overrides JSONRPCRequest.method

> **params: SubscribeRequestParams**

> Overrides JSONRPCRequest.params

SubscribeRequestParams

```
interface SubscribeRequestParams {
  _meta?: { progressToken?: ProgressToken; [key: string]: unknown };
  uri: string;
}
```

Parameters for a `resources/subscribe` request.

> **_meta?: { progressToken?: ProgressToken; [key: string]: unknown }**
>
> See General fields: `_meta` for notes on `_meta` usage.
>
> Type Declaration
>
> - [key: string]: unknown
> - `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 ResourceRequestParams._meta

> **uri: string**
>
> The URI of the resource. The URI can use any protocol; it is up to the server how to interpret it.
>
> Inherited from ResourceRequestParams.uri

resources/templates/list

ListResourceTemplatesRequest

```
interface ListResourceTemplatesRequest {
  jsonrpc: "2.0";
  id: RequestId;
  params?: PaginatedRequestParams;
  method: "resources/templates/list";
}
```

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

> **jsonrpc: "2.0"**
>
> Inherited from PaginatedRequest.jsonrpc

> **id: RequestId**
> Inherited from PaginatedRequest.id

> **params?: PaginatedRequestParams**
> 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 {
  icons?: Icon[];
  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.

icons?: Icon[]

Optional set of sized icons that the client can display in a user interface.

Clients that support rendering icons MUST support at least the following MIME types:

- `image/png` - PNG images (safe, universal compatibility)
- `image/jpeg` (and `image/jpg`) - JPEG images (safe, universal compatibility)

Clients that support rendering icons SHOULD also support:

- `image/svg+xml` - SVG images (scalable but requires security precautions)
- `image/webp` - WebP images (modern, efficient format)

Inherited from Icons.icons

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 {
  jsonrpc: "2.0";
  id: RequestId;
  method: "resources/unsubscribe";
  params: UnsubscribeRequestParams;
}
```

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

jsonrpc: "2.0"
Inherited from JSONRPCRequest.jsonrpc

id: RequestId
Inherited from JSONRPCRequest.id

method: "resources/unsubscribe"
Overrides JSONRPCRequest.method

params: UnsubscribeRequestParams
Overrides JSONRPCRequest.params

UnsubscribeRequestParams

```
interface UnsubscribeRequestParams {
  _meta?: { progressToken?: ProgressToken; [key: string]: unknown };
  uri: string;
}
```

Parameters for a `resources/unsubscribe` request.

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

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

Type Declaration

- [key: string]: unknown
- `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 ResourceRequestParams._meta

> **uri: string**
>
> The URI of the resource. The URI can use any protocol; it is up to the server how to interpret it.
>
> Inherited from ResourceRequestParams.uri

roots/list

ListRootsRequest

```
interface ListRootsRequest {
  jsonrpc: "2.0";
  id: RequestId;
  method: "roots/list";
  params?: RequestParams;
}
```

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.

> **jsonrpc: "2.0"**
> Inherited from JSONRPCRequest.jsonrpc

> **id: RequestId**
> Inherited from JSONRPCRequest.id

> **method: "roots/list"**
> Overrides JSONRPCRequest.method

> **params?: RequestParams**
> Overrides JSONRPCRequest.params

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 {
  jsonrpc: "2.0";
  id: RequestId;
  method: "sampling/createMessage";
  params: CreateMessageRequestParams;
}
```

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.

> **`jsonrpc: "2.0"`**
> Inherited from JSONRPCRequest.jsonrpc

> **`id: RequestId`**
> Inherited from JSONRPCRequest.id

> **`method: "sampling/createMessage"`**
> Overrides JSONRPCRequest.method

> **`params: CreateMessageRequestParams`**
> Overrides JSONRPCRequest.params

`CreateMessageRequestParams`

```
interface CreateMessageRequestParams {
  task?: TaskMetadata;
  _meta?: { progressToken?: ProgressToken; [key: string]: unknown };
  messages: SamplingMessage[];
  modelPreferences?: ModelPreferences;
  systemPrompt?: string;
  includeContext?: "none" | "thisServer" | "allServers";
  temperature?: number;
  maxTokens: number;
  stopSequences?: string[];
  metadata?: object;
  tools?: Tool[];
  toolChoice?: ToolChoice;
}
```

Parameters for a `sampling/createMessage` request.

> **`task?: TaskMetadata`**
>
> If specified, the caller is requesting task-augmented execution for this request. The request will return a CreateTaskResult immediately, and the actual result can be retrieved later via tasks/result.
>
> Task augmentation is subject to capability negotiation - receivers MUST declare support for task augmentation of specific request types in their capabilities.
>
> Inherited from TaskAugmentedRequestParams.task

> **`_meta?: { progressToken?: ProgressToken; [key: string]: unknown }`**
>
> See General fields: `_meta` for notes on `_meta` usage.
>
> Type Declaration

- [key: string]: unknown
- `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 TaskAugmentedRequestParams._meta

messages: SamplingMessage[]

modelPreferences?: ModelPreferences

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

systemPrompt?: string

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

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.

Default is "none". Values "thisServer" and "allServers" are soft-deprecated. Servers SHOULD only use these values if the client declares ClientCapabilities.sampling.context. These values may be removed in future spec releases.

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.

stopSequences?: string[]

metadata?: object

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

tools?: Tool[]

Tools that the model may use during generation. The client MUST return an error if this field is provided but ClientCapabilities.sampling.tools is not declared.

toolChoice?: ToolChoice

Controls how the model uses tools. The client MUST return an error if this field is provided but ClientCapabilities.sampling.tools is not declared. Default is `{ mode: "auto" }`.

`CreateMessageResult`

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

The client's response to a sampling/createMessage 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.
>
> Standard values:
>
> - "endTurn": Natural end of the assistant's turn
> - "stopSequence": A stop sequence was encountered
> - "maxTokens": Maximum token limit was reached
> - "toolUse": The model wants to use one or more tools
>
> This field is an open string to allow for provider-specific stop reasons.

> **`role: Role`**
>
> Inherited from SamplingMessage.role

> **`content: SamplingMessageContentBlock | SamplingMessageContentBlock[]`**
>
> Inherited from SamplingMessage.content

`ModelHint`

```ts
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`

```ts
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: SamplingMessageContentBlock | SamplingMessageContentBlock[];
  _meta?: { [key: string]: unknown };
}
```

Describes a message issued to or received from an LLM API.

> **`role: Role`**

> **`content: SamplingMessageContentBlock | SamplingMessageContentBlock[]`**

> **`_meta?: { [key: string]: unknown }`**
>
> See General fields: `_meta` for notes on `_meta` usage.

`SamplingMessageContentBlock`

```
SamplingMessageContentBlock:
  | TextContent
  | ImageContent
  | AudioContent
  | ToolUseContent
  | ToolResultContent
```

`ToolChoice`

```
interface ToolChoice {
  mode?: "none" | "required" | "auto";
}
```

Controls tool selection behavior for sampling requests.

> **`mode?: "none" | "required" | "auto"`**
>
> Controls the tool use ability of the model:
>
> - "auto": Model decides whether to use tools (default)
> - "required": Model MUST use at least one tool before completing
> - "none": Model MUST NOT use any tools

`ToolResultContent`

```ts
interface ToolResultContent {
  type: "tool_result";
  toolUseId: string;
  content: ContentBlock[];
  structuredContent?: { [key: string]: unknown };
  isError?: boolean;
  _meta?: { [key: string]: unknown };
}
```

The result of a tool use, provided by the user back to the assistant.

> **`type: "tool_result"`**

> **`toolUseId: string`**
>
> The ID of the tool use this result corresponds to.
>
> This MUST match the ID from a previous ToolUseContent.

> **`content: ContentBlock[]`**
>
> The unstructured result content of the tool use.
>
> This has the same format as CallToolResult.content and can include text, images, audio, resource links, and embedded resources.

> **`structuredContent?: { [key: string]: unknown }`**
>
> An optional structured result object.
>
> If the tool defined an outputSchema, this SHOULD conform to that schema.

> **`isError?: boolean`**
>
> Whether the tool use resulted in an error.
>
> If true, the content typically describes the error that occurred. Default: false

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

> Optional metadata about the tool result. Clients SHOULD preserve this field when including tool results in subsequent sampling requests to enable caching optimizations.
>
> See General fields: `_meta` for notes on `_meta` usage.

`ToolUseContent`

```
interface ToolUseContent {
  type: "tool_use";
  id: string;
  name: string;
  input: { [key: string]: unknown };
  _meta?: { [key: string]: unknown };
}
```

A request from the assistant to call a tool.

> **`type: "tool_use"`**

> **`id: string`**
>
> A unique identifier for this tool use.
>
> This ID is used to match tool results to their corresponding tool uses.

> **`name: string`**
>
> The name of the tool to call.

> **`input: { [key: string]: unknown }`**
>
> The arguments to pass to the tool, conforming to the tool's input schema.

> **`_meta?: { [key: string]: unknown }`**
>
> Optional metadata about the tool use. Clients SHOULD preserve this field when including tool uses in subsequent sampling requests to enable caching optimizations.
>
> See General fields: `_meta` for notes on `_meta` usage.

`tools/call`

`CallToolRequest`

```
interface CallToolRequest {
  jsonrpc: "2.0";
  id: RequestId;
  method: "tools/call";
  params: CallToolRequestParams;
}
```

Used by the client to invoke a tool provided by the server.

> **jsonrpc: "2.0"**
> Inherited from JSONRPCRequest.jsonrpc

> **id: RequestId**
> Inherited from JSONRPCRequest.id

> **method: "tools/call"**
> Overrides JSONRPCRequest.method

> **params: CallToolRequestParams**
> Overrides JSONRPCRequest.params

`CallToolRequestParams`

```
interface CallToolRequestParams {
  task?: TaskMetadata;
  _meta?: { progressToken?: ProgressToken; [key: string]: unknown };
  name: string;
  arguments?: { [key: string]: unknown };
}
```

Parameters for a `tools/call` request.

> **task?: TaskMetadata**
>
> If specified, the caller is requesting task-augmented execution for this request. The request will return a CreateTaskResult immediately, and the actual result can be retrieved later via tasks/result.
>
> Task augmentation is subject to capability negotiation - receivers MUST declare support for task augmentation of specific request types in their capabilities.
>
> Inherited from TaskAugmentedRequestParams.task

> **_meta?: { progressToken?: ProgressToken; [key: string]: unknown }**
>
> See General fields: `_meta` for notes on `_meta` usage.
>
> Type Declaration
>
> - [key: string]: unknown

- `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 TaskAugmentedRequestParams._meta

name: string

The name of the tool.

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

Arguments to use for the tool call.

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 {
  jsonrpc: "2.0";
  id: RequestId;
  params?: PaginatedRequestParams;
  method: "tools/list";
}
```

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

> **`jsonrpc: "2.0"`**
> Inherited from PaginatedRequest.jsonrpc

> **`id: RequestId`**
> Inherited from PaginatedRequest.id

> **`params?: PaginatedRequestParams`**
> 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 {
  icons?: Icon[];
  name: string;
  title?: string;
  description?: string;
  inputSchema: {
    $schema?: string;
    type: "object";
    properties?: { [key: string]: object };
    required?: string[];
  };
  execution?: ToolExecution;
  outputSchema?: {
    $schema?: string;
    type: "object";
    properties?: { [key: string]: object };
    required?: string[];
  };
  annotations?: ToolAnnotations;
  _meta?: { [key: string]: unknown };
}
```

Definition for a tool the client can call.

> **icons?: Icon[]**
>
> Optional set of sized icons that the client can display in a user interface.
>
> Clients that support rendering icons MUST support at least the following MIME types:
>
> - `image/png` - PNG images (safe, universal compatibility)
> - `image/jpeg` (and `image/jpg`) - JPEG images (safe, universal compatibility)
>
> Clients that support rendering icons SHOULD also support:
>
> - `image/svg+xml` - SVG images (scalable but requires security precautions)
> - `image/webp` - WebP images (modern, efficient format)
>
> Inherited from Icons.icons

> **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: { $schema?: string; type: "object"; properties?: { [key: string]: object }; required?: string[]; }`**
>
> A JSON Schema object defining the expected parameters for the tool.

> **`execution?: ToolExecution`**
>
> Execution-related properties for this tool.

> **`outputSchema?: { $schema?: string; 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.
>
> Defaults to JSON Schema 2020-12 when no explicit $schema is provided. Currently restricted to type: "object" at the root level.

> **`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`

```ts
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 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

`ToolExecution`

```
interface ToolExecution {
  taskSupport?: "forbidden" | "optional" | "required";
}
```

Execution-related properties for a tool.

> **`taskSupport?: "forbidden" | "optional" | "required"`**
>
> Indicates whether this tool supports task-augmented execution. This allows clients to handle long-running operations through polling the task system.
>
> - "forbidden": Tool does not support task-augmented execution (default when absent)
> - "optional": Tool may support task-augmented execution
> - "required": Tool requires task-augmented execution
>
> Default: "forbidden"