Specification · Draft

Open Model Context Protocol
Version draft

Built from `7596f66` · 2026-09-23

> **WORKING DRAFT**
>
> This document changes on every merge to `main`. It is not a released version and must not be relied on for interoperability.

Contents

1 Specification
Overview
Key Details
Security and Trust & Safety
Learn More
2 Changelog
3 Deprecated Features
Deprecated
Removed
4 Architecture
Core Components
Design Principles
Capability Negotiation
5 Base Protocol
5.1 Overview
Messages
Statelessness
Auth
Schema
JSON Schema Usage
General fields
5.2 Versioning and Compatibility
Terminology
Protocol Version Negotiation
Extension Negotiation
Backward Compatibility with Initialization-Based Versions
5.3 Message Patterns
5.3.1 Overview
Request and Response
Multi Round-Trip Requests
Subscribe and Notify
Adding Patterns
5.3.2 Multi Round-Trip Requests
Multi Round-Trip Requests
5.3.3 Subscriptions
Opening a Stream
Acknowledgment
Receiving Notifications
Multiple Concurrent Subscriptions
Cancellation
5.3.4 Cancellation
Cancellation Flow
Transport-Specific Cancellation
Timeouts
Behavior Requirements
Timing Considerations
Implementation Notes
Error Handling
5.3.5 Progress

Progress Flow
Behavior Requirements
Implementation Notes
5.4 Transports
5.4.1 Overview
Messages
Request Metadata
Cancellation
Custom Transports
Backward Compatibility
5.4.2 stdio
Sending Messages
Receiving Messages
Request Metadata
Cancellation
Shutdown
Unexpected Termination
Backward Compatibility
5.4.3 Streamable HTTP
Security & Endpoint
Sending Messages
Receiving Messages
Message Flow
Cancellation
Request Metadata
Backward Compatibility
5.5 Authorization
5.5.1 Authorization
Introduction
Roles
Overview
Authorization Server Discovery
Client Registration
Scope Selection Strategy
Authorization Flow Steps
Resource Parameter Implementation
Access Token Usage
Refresh Tokens
Error Handling
Security Considerations
MCP Authorization Extensions
5.5.2 Authorization Server Discovery
Authorization Server Location
Protected Resource Metadata Discovery Requirements
Authorization Server Metadata Discovery
Sequence Diagram
5.5.3 Client Registration
Client ID Metadata Documents
Pre-registration
Dynamic Client Registration
Authorization Server Binding

5.5.4 Authorization Security Considerations
Token Audience Binding and Validation
Token Theft
Communication Security
Authorization Code Protection
Mix-Up Attacks
Open Redirection
Client ID Metadata Document Security
Confused Deputy Problem
Access Token Privilege Restriction
6 Client Features
6.1 Roots
User Interaction Model
Capabilities
Protocol Messages
Message Flow
Data Types
Error Handling
Security Considerations
Implementation Guidelines
6.2 Sampling
User Interaction Model
Tools in Sampling
Capabilities
Protocol Messages
Message Content Constraints
Cross-API Compatibility
Message Flow
Data Types
Error Handling
Security Considerations
6.3 Elicitation
User Interaction Model
Capabilities
Protocol Messages
Message Flow
Response Actions
Implementation Considerations
Error Handling
Security Considerations
7 Server Features
7.1 Overview
7.2 Discovery
Request
Response
When to Call
Data Types
7.3 Prompts
User Interaction Model
Capabilities
Protocol Messages

- Message Flow
- Data Types
- Error Handling
- Implementation Considerations
- Security

7.4 Resources

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

7.5 Tools

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

7.6 Utilities

7.6.1 Caching

- Cacheable Results
- Cache Key
- Cacheable Model
- Interaction with Notifications
- Interaction with Pagination
- Security Considerations

7.6.2 Completion

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

7.6.3 Logging

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

7.6.4 Pagination

- Pagination Model

Response Format
Request Format
Pagination Flow
Operations Supporting Pagination
Implementation Guidelines
Error Handling
8 Schema Reference
JSON-RPC
Common Types
Errors
Content
completion/complete
elicitation/create
notifications/cancelled
notifications/message
notifications/progress
notifications/prompts/list_changed
notifications/resources/list_changed
notifications/resources/updated
notifications/subscriptions/acknowledged
notifications/tools/list_changed
Multi Round-Trip
prompts/get
prompts/list
resources/list
resources/read
resources/templates/list
roots/list
sampling/createMessage
server/discover
subscriptions/listen
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
- Stateless, self-contained requests
- Per-request 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:

- **Elicitation**: Server-initiated requests for additional information from users

Additional Utilities

- Configuration
- Progress tracking
- Cancellation
- Error reporting

Extensions

Beyond the core protocol, MCP defines optional extensions that add modular, specialized, or experimental functionality. Extensions are always opt-in and require explicit support from both client and server, negotiated during initialization. Notable extensions include:

- **Tasks**: Asynchronous execution of long-running operations, with polling, mid-flight input, and durable handles
- **Skills over MCP**: Rich, structured instructions for agent workflows, discovered and consumed through MCP
- **MCP Apps**: Interactive UI elements (charts, forms, video players) rendered inline within conversations

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

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 Changelog

Changes since the most recent release will accumulate here.

3 Deprecated Features

This page is the registry of specification features that are currently in the **Deprecated** state under the feature lifecycle and deprecation policy (SEP-2596).

A Deprecated feature remains part of the specification but is scheduled for removal: new implementations **SHOULD NOT** adopt it, and existing implementations **SHOULD** migrate before the feature's earliest removal. The earliest removal marks when a feature becomes *eligible* for removal; the actual removal is a Core Maintainer decision taken during release preparation and may happen later.

This registry is a derived view kept consistent with the per-feature deprecation notices and changelog entries, which are the normative records.

Deprecated

Feature	Deprecation SEP	Deprecated in	Migration path	Earliest removal
Roots	SEP-2577	`2026-07-28`	Pass directories or files via tool parameters, resource URIs, or server configuration	First revision released on or after 2027-07-28
Sampling	SEP-2577	`2026-07-28`	Integrate directly with LLM provider APIs	First revision released on or after 2027-07-28
Logging	SEP-2577	`2026-07-28`	Log to `stderr` for stdio transports; use OpenTelemetry for observability	First revision released on or after 2027-07-28
Dynamic Client Registration	PR #2858	`2026-07-28`	Client ID Metadata Documents	First revision released on or after 2027-07-28
`includeContext: "thisServer"` / `"allServers"` (Sampling)	SEP-2596	`2025-11-25`	Omit the field or use `"none"`	Follows Sampling (SEP-2577)
HTTP+SSE transport	SEP-2596	`2025-03-26`	Streamable HTTP	Three months after SEP-2596 reaches Final

The HTTP+SSE transport and the `includeContext` values were already described as deprecated before the lifecycle policy existed; SEP-2596 reclassifies them as Deprecated under its transition provisions.

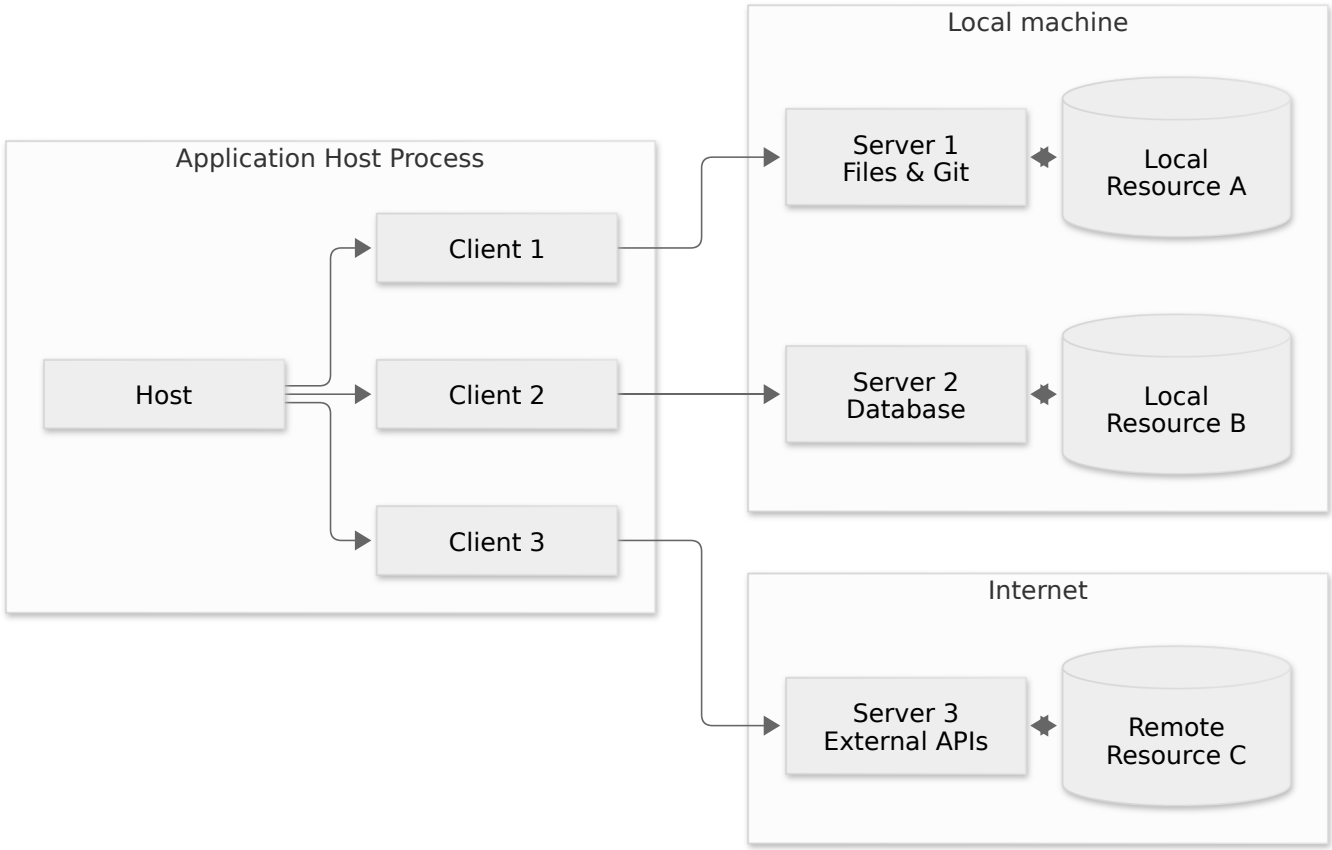
Removed

No features have been removed under this policy yet. When a Deprecated feature is removed, its row moves to this section with a link to the changelog entry recording the removal.

4 Architecture

The Model Context Protocol (MCP) follows a client-host-server architecture where each host can run multiple client instances. MCP is a stateless protocol: every request is self-contained and carries its own protocol version and capabilities. 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 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 communicates with exactly one server:

- Communicates with exactly one server
- Attaches protocol version and capabilities to every request
- 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 client input (sampling, elicitation, roots) via `InputRequiredResult` within a reply
- Must respect security constraints
- Can be local processes or remote services

Design Principles

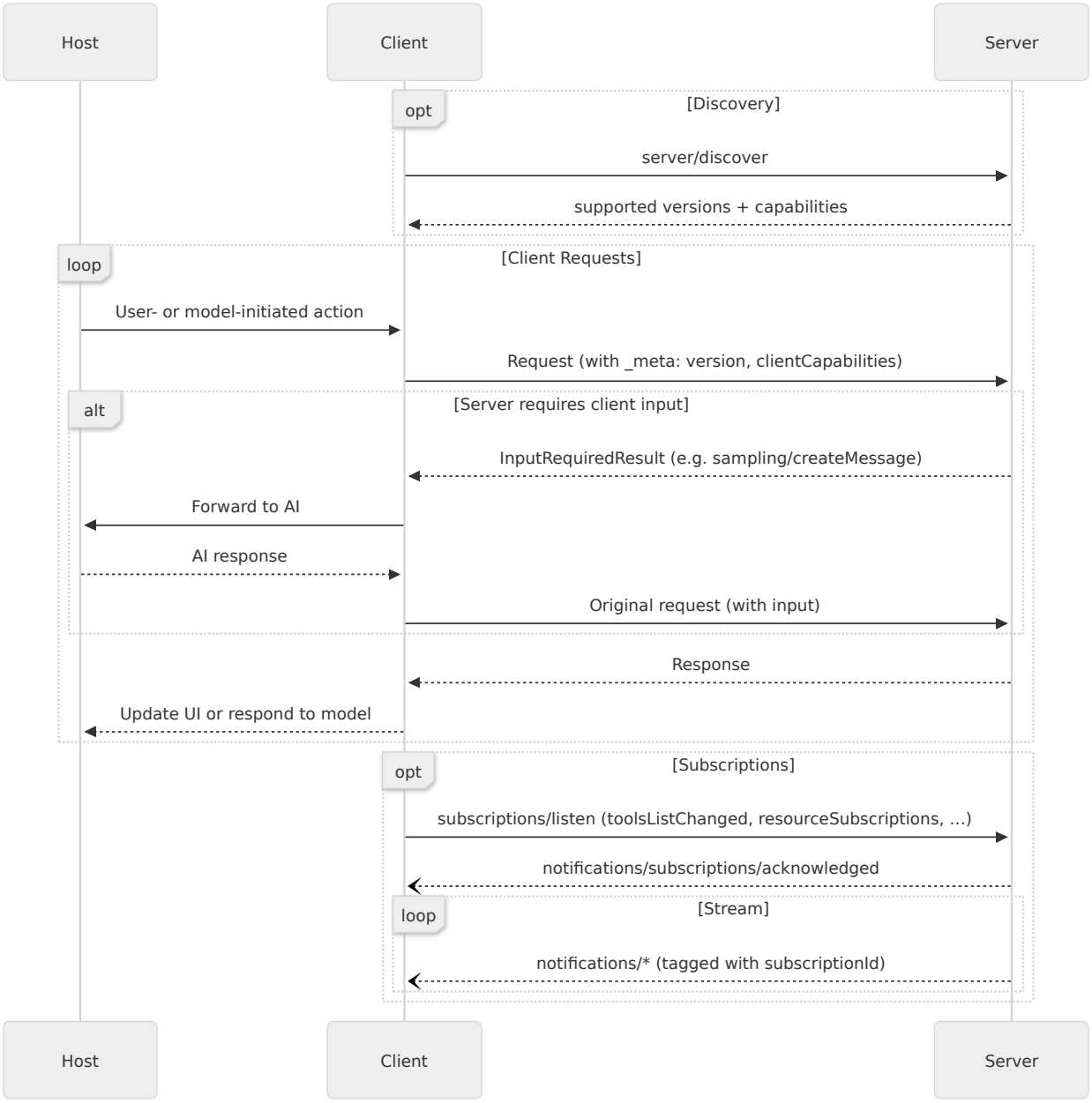
MCP is built on several key design principles that inform its architecture and implementation:

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 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 declare their supported features on each request. Clients include their capabilities in `_meta.io.modelcontextprotocol/clientCapabilities` on every request. Servers advertise their capabilities in response to `server/discover`, which clients may call before any other request for up-front capability discovery.

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



Each capability unlocks specific protocol features on a per-request basis. For example:

- Implemented server features must be advertised in the server's capabilities
- Receiving resource update notifications requires opening a `subscriptions/listen` stream with the desired resource URIs

- Tool invocation requires the server to declare tool capabilities

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

5 Base Protocol

5.1 Overview

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

- **Base Protocol**: Core JSON-RPC message types
- **Versioning and Compatibility**: Protocol version negotiation, extension negotiation, and interoperability with earlier protocol revisions
- **Message Patterns**: Messaging patterns supported by the core protocol including request and response, multi round-trip requests (MRTR), and subscribe and notify
- **Authorization**: Authentication and authorization framework for HTTP-based transports
- **Server Features**: Resources, prompts, and tools exposed by servers
- **Client Features**: Elicitation, sampling and root directory lists provided by clients
- **Utilities**: Cross-cutting concerns like logging and argument completion

All implementations **MUST** support the base protocol, versioning, and the message patterns. 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, 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** match the ID of any other request the sender has issued and not yet received a response for.

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: {
    resultType: string;
    [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.
- The `result` **MUST** include a `resultType` field to indicate the type of the result.

ResultType

The `resultType` field in a result indicates the type of the result being returned. MCP supports polymorphic result types, allowing servers to return different structures based on the outcome of the request. The `resultType` field is a string that clients can use to determine how to parse and handle the `result` object.

- A `resultType` of `"complete"` indicates the request completed successfully and the result contains the final content.
- A `resultType` of `"input_required"` indicates the request is incomplete and more information is needed to process the request. The result contains an `InputRequiredResult` object with additional information needed.
- Extensions **MAY** add additional `ResultType` values. The set of supported `ResultType` values **MUST** be created from the set defined in the core protocol and include any additional values of supported extensions that are advertised via capabilities.
- A `resultType` of any value unrecognized by the client **MUST** be considered invalid.
- For backward compatibility with servers implementing earlier protocol versions, which do not include `resultType`, clients **MUST** treat an absent `resultType` as `"complete"`.

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.
- Error responses **MAY** include a `data` member with additional information of any type, such as nested errors.

Error Codes

MCP uses the standard JSON-RPC 2.0 error codes (`-32700` , `-32600` to `-32603`) for general protocol failures.

JSON-RPC 2.0 reserves the range `-32000` to `-32099` for implementation-defined server errors. MCP partitions this range as follows:

- **`-32000` to `-32019` — legacy.** Codes in this sub-range were allocated by implementations before this policy was introduced. New codes **MUST NOT** be allocated in this sub-range, and new implementations **SHOULD NOT** use codes from this sub-range at all. Apart from `-32002` (see below), receivers **MUST NOT** assume any specific meaning for these codes.
- **`-32020` to `-32099` — reserved for the MCP specification.** Error codes in this sub-range are defined exclusively by the MCP specification and recorded in the schema. Implementations **MUST NOT** emit any code from this sub-range that is not defined by this specification and **MUST** use defined codes only with their specified meanings.

MCP defines the following error codes:

Code	Name
`-32020`	`HeaderMismatch`
`-32021`	`MissingRequiredClientCapability`
`-32022`	`UnsupportedProtocolVersion`

Codes defined by earlier protocol versions remain reserved and will not be reused. Implementations of this protocol version **MUST NOT** emit these codes:

- `-32002` — resource not found (2025-11-25 and earlier; replaced by `-32602`). Clients **SHOULD** still accept `-32002` from servers implementing earlier versions.
- `-32042` — URL elicitation required (2025-11-25 only).

Errors that are purely local to an implementation (for example, a request timeout raised inside an SDK) are not currently assigned codes by this specification. Implementations surfacing local errors in JSON-RPC-shaped structures should ensure they cannot be mistaken for errors received from the peer. Future versions of the specification may define standard codes for common local error conditions in the reserved sub-range.

New error codes for purposes not defined by this specification **SHOULD** be allocated outside the JSON-RPC reserved range (`-32768` to `-32000`); the remainder of the integer space is available for application-defined errors.

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.

Message Patterns

The Model Context Protocol (MCP) supports several Message Patterns that define how clients and servers interact:

1. **Request and Response**: A client sends a request to the server, and the server responds with a result or error.
2. **Multi Round-Trip Requests (MRTR)**: A server requires additional client input (sampling, elicitation, or roots) to complete a request.
3. **Subscribe and Notify**: A client subscribes to a stream of notifications from the server, which are sent as they occur.

Statelessness

The Model Context Protocol (MCP) is a **stateless protocol**: all the information needed to process a request is contained in the request itself. A server processes each request independently; no state should be inferred from previous requests, even those on the same connection or stream.

Specifically:

- Servers **MUST NOT** rely on prior requests over the same connection to establish context (e.g., capabilities, protocol version, client identity). Every request supplies this metadata in its `_meta` field.
- Servers **SHOULD** be prepared to handle requests associated with multiple tasks, threads, or conversations.
- Servers **SHOULD NOT** require that a client reuse the same connection or process to perform related operations.
- Clients **SHOULD NOT** use an individual task, thread, or conversation as the lifetime boundary for the stdio process.
- State that needs to span multiple requests (e.g., long-running tasks, application-level handles) **MUST** be referenced by an explicit identifier the client passes on each request.

> **NOTE**
>
> This implies that an open connection, such as a STDIO process, is not a conversation or session: clients may interleave unrelated requests on the same transport, and a server must not treat connection or process identity as a proxy for conversation or session continuity.

Long-lived requests like `subscriptions/listen` remain request/response; the response is just an open stream of notifications. Their state is scoped to the request itself, not to the connection underneath.

> **INFO**
>
> For a walkthrough of how the per-request model maps to SDK code, see the Architecture guide.

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

`$ref` Resolution

JSON Schema 2020-12 permits `$ref` to point at an absolute URI. Implementations **MUST NOT** automatically dereference `$ref` values that resolve to a network URI.

Implementations **MAY** offer an opt-in mode that fetches non-local `$ref`s but it **MUST** be disabled by default and **SHOULD** enforce an allowlist of hosts or at minimum reject loopback, link-local, and private network addresses, apply timeouts and size limits, and log dereferenced URIs.

Schemas that fail to validate due to an unresolved external `$ref` **SHOULD** be rejected rather than silently treated as permissive.

Composition-Keyword Resource Use

Composition keywords (`anyOf`, `oneOf`, `allOf`, `if`/`then`/`else`) and `$defs` enable expressive schemas but can be expensive to validate. Implementations **SHOULD** apply reasonable bounds, such as a maximum schema depth, a cap on the total number of subschemas, or a per-validation time budget, to prevent a malicious schema from acting as a Denial-of-Service vector against the validator.

General fields

`_meta`

The `_meta` property/parameter is used 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.

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.

Reserved keys:

The following `_meta` keys are reserved by this specification:

Key	Description	Defined in
`progressToken`	Opts the request into progress notifications	Progress
`io.modelcontextprotocol/protocolVersion`	Protocol version for a request	Per-request protocol fields (below)
`io.modelcontextprotocol/clientInfo`	Client name and version	Per-request protocol fields (below)
`io.modelcontextprotocol/clientCapabilities`	Client capabilities relevant to a request	Per-request protocol fields (below)
`io.modelcontextprotocol/logLevel`	Minimum log level the server should emit for a request	Logging
`io.modelcontextprotocol/subscriptionId`	Correlates a notification with its originating subscription	Subscriptions
`traceparent`, `tracestate`, `baggage`	OpenTelemetry trace context propagation	OpenTelemetry trace context (below)

Official extensions define additional `_meta` keys under the `io.modelcontextprotocol/` prefix, and third-party extensions use their own vendor prefix. In both cases the keys are specified in the extension's documentation.

Per-request protocol fields:

Client requests carry the following `io.modelcontextprotocol/*` fields in `_meta`; fields marked as required **MUST** be included on every request. Servers use these to identify the protocol version and capabilities in use without relying on any prior connection state. See Versioning and Compatibility for version negotiation rules.

Key	Type	Required	Description
`io.modelcontextprotocol/protocolVersion`	`string`	Yes	Protocol version for this request (e.g., `"2026-07-28"`)
`io.modelcontextprotocol/clientInfo`	`Implementation`	No	Client name and version
`io.modelcontextprotocol/clientCapabilities`	`ClientCapabilities`	Yes	Client capabilities relevant to this request
`io.modelcontextprotocol/logLevel`	`LoggingLevel`	No	Minimum log level the server should emit for this request

A request missing any required field is malformed; the server **MUST** reject it with JSON-RPC error code `-32602` (Invalid params). On HTTP, the response status **MUST** be `400 Bad Request`.

Clients **SHOULD** include `io.modelcontextprotocol/clientInfo` on every request unless specifically configured not to do so.

A server **MUST NOT** rely on capabilities the client has not declared. If processing a request requires a capability the client did not include in `io.modelcontextprotocol/clientCapabilities`, the server **MUST** return a `MissingRequiredClientCapabilityError` (`-32021`) whose `data.requiredCapabilities` lists the missing capabilities. On HTTP, the response status **MUST** be `400 Bad Request`.

Per-response protocol fields:

Servers **SHOULD** include the following `io.modelcontextprotocol/*` field in every result's `_meta`, unless specifically configured not to do so, to identify themselves without relying on any prior connection state:

Key	Type	Required	Description
`io.modelcontextprotocol/serverInfo`	`Implementation`	No	Server name and version

> **NOTE**
>
> `io.modelcontextprotocol/clientInfo` and `io.modelcontextprotocol/serverInfo` are self-reported by the sender and are not verified by the protocol. They are intended for display, logging, and debugging. Implementations **SHOULD NOT** use them to change the behavior of the client or server, and **SHOULD NOT** rely on them for security decisions.

On notifications delivered via a `subscriptions/listen` stream, the server **MUST** include `io.modelcontextprotocol/subscriptionId` in `_meta` so the client can correlate the notification with the

originating subscription request.

OpenTelemetry trace context:

As an exception to the prefix requirement above, the keys `traceparent`, `tracestate`, and `baggage` are reserved for OpenTelemetry trace context propagation. When present, their values MUST follow W3C Trace Context and W3C Baggage formats respectively.

This exception exists to maintain compatibility with existing implementations and OpenTelemetry semantic conventions for MCP.

Non-normative example of trace context in `_meta`:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "tools/call",
  "params": {
    "name": "get_weather",
    "arguments": {
      "location": "New York"
    },
    "_meta": {
      "traceparent": "00-0af7651916cd43dd8448eb211c80319c-00f067aa0ba902b7-01"
    }
  }
}
```

`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 override if the source MIME 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:

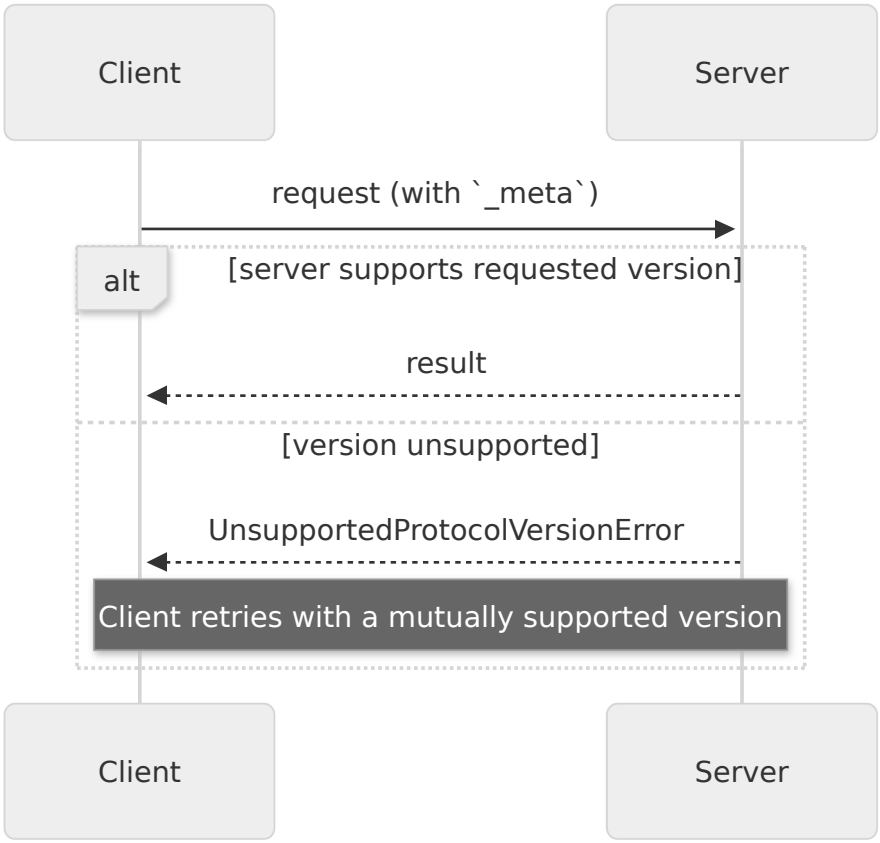
- `Implementation`: Visual identifier for the MCP server/client implementation
- `Tool`: Visual representation of the tool's functionality
- `Prompt`: Icon to display alongside prompt templates
- `Resource`: Visual indicator for different resource types

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

5.2 Versioning and Compatibility

This page defines how a client and server agree on what they are speaking: the protocol version, declared on every request; optional extensions, negotiated through capabilities; and interoperability with earlier, handshake-based protocol revisions.

There is no negotiation handshake. Every request carries its protocol version, and the server accepts or rejects each request independently:



Terminology

This page uses the following terms for interoperability across protocol revisions:

- **Modern**: protocol versions that convey version, identity, and capabilities as per-request metadata (revision `2026-07-28` and later).
- **Legacy**: protocol versions that establish a session with an `initialize` handshake (`2025-11-25` and earlier).
- **Dual-era**: an implementation that supports both modern and legacy versions.

Protocol Version Negotiation

Every request declares the protocol version it is using in its `_meta` field. On HTTP, this is also carried in the `MCP-Protocol-Version` header.

If the server does not implement the requested version (whether the version is unknown to the server, or is a known version the server has chosen not to support), it **MUST** respond with an `UnsupportedProtocolVersionError` listing the versions it does support:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "error": {
    "code": -32022,
    "message": "Unsupported protocol version",
    "data": {
      "supported": ["2026-07-28", "2025-11-25"],
      "requested": "1900-01-01"
    }
  }
}
```

The client **SHOULD** select a mutually supported version from the `supported` list and retry the request, or surface an error to the user if no compatible version exists.

Servers **MUST** implement `server/discover`. Clients **MAY** call it before sending any other requests to learn the server's supported versions up front, but are not required to: a client is free to invoke any RPC inline and handle `UnsupportedProtocolVersionError` if its preferred version is not supported.

Extension Negotiation

Clients and servers can negotiate support for optional extensions beyond the core protocol. Extensions are advertised in the `extensions` field of capabilities, which is a map of extension identifiers to per-extension settings objects. Extension identifiers **MUST** follow the `_meta` key naming rules, with a mandatory prefix.

The following is an example of a client that advertises the MCP Apps extension identified as `io.modelcontextprotocol/ui`:

```
{
  "capabilities": {
    "roots": {},
    "extensions": {
      "io.modelcontextprotocol/ui": {
        "mimeTypes": ["text/html;profile=mcp-app"]
      }
    }
  }
}
```

An example of Tasks extension identified as `io.modelcontextprotocol/tasks`:

```
{
  "capabilities": {
    "tools": {},
    "extensions": {
      "io.modelcontextprotocol/tasks": {}
    }
  }
}
```

Each extension specifies the schema of its settings object; an empty object indicates support with no additional settings.

If one party supports an extension but the other does not, the supporting party **MUST** either revert to core protocol behavior or reject the request with an appropriate error. Extensions **SHOULD** document their expected fallback behavior.

Backward Compatibility with Initialization-Based Versions

A server that wishes to support both legacy clients (which expect an `initialize` handshake) and modern clients (which use per-request metadata) **MAY** implement both behaviors.

A client that needs to interoperate with both kinds of servers detects the server's era with transport-specific mechanics, specified in the binding pages:

- stdio: probe with `server/discover` and fall back on any error that is not a recognized modern error.
- Streamable HTTP: attempt a modern request and inspect the body of a `400 Bad Request` before falling back.

In both cases, a recognized modern JSON-RPC error (such as `UnsupportedProtocolVersionError`) identifies a modern server: the client retries with a supported version rather than falling back. Anything else identifies a legacy server.

The era determination is a property of the server, not of an individual request. Clients **SHOULD** cache the result for the lifetime of the server process (stdio) or origin (HTTP), and **MAY** persist it across restarts of the same server configuration, re-probing if the cached assumption later fails.

A server that supports only modern versions **SHOULD** name the protocol versions it supports in any error it returns to an `initialize` request, on any transport: legacy clients have no fall-forward mechanism, and this message may be the only diagnostic they can surface to users.

Compatibility Matrix

The following matrix summarizes the expected outcome of every combination of client and server era:

Client	Server	Outcome
Modern	Modern	Works. `server/discover` is optional; version mismatches surface as `UnsupportedProtocolVersionError` and the client retries with a mutually supported version.
Modern	Legacy	Fails. The server may reject the request with an implementation-defined error, stay silent, or even process an era-ambiguous method under legacy semantics. On stdio, clients **SHOULD** send `server/discover` first to fail deterministically; the client then surfaces an actionable error to the user.
Dual-era	Modern	Works. The stdio probe returns a `DiscoverResult` (or `UnsupportedProtocolVersionError`); on HTTP, the first modern request succeeds or returns a modern error. The client stays modern.
Dual-era	Legacy	Works. stdio: the probe returns a non-modern error or times out, and the client falls back to `initialize`. HTTP: the modern request returns a `4xx` without a recognized modern error body, and the client falls back to `initialize` (and possibly further to the deprecated HTTP+SSE transport).
Legacy	Modern	Fails. stdio: the server rejects `initialize` with a JSON-RPC error; the exact code is implementation-defined (`initialize` is an unknown method and the request also lacks the required `_meta` fields). HTTP: the request is missing the required headers and is rejected per server validation with `400 Bad Request` (a client on the deprecated HTTP+SSE transport fails at its opening `GET` instead). Legacy clients have no fall-forward mechanism.
Legacy	Dual-era	Works. The server answers `initialize` and serves the client according to the negotiated legacy revision.
Legacy	Legacy	Works according to the legacy revision; out of scope for this document.

A dual-era **server** selects its behavior from how the client opens:

- A request carrying modern per-request `_meta` is served statelessly according to this revision.
- An `initialize` request selects legacy semantics, scoped to the stdio process (stdio) or the session (HTTP), as specified by the negotiated legacy protocol version.

A dual-era server **MAY** serve both eras concurrently on the same endpoint or process.

5.3 Message Patterns

5.3.1 Overview

This page defines the message patterns of the core protocol: the ways a client and server compose JSON-RPC requests, responses, and notifications into interactions. Every transport carries all of these patterns; transports differ only in how messages are framed and delivered.

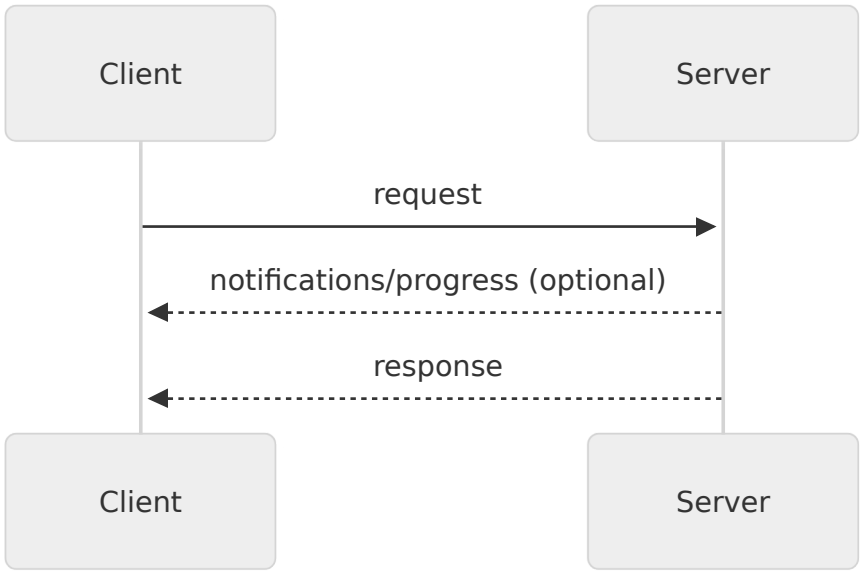
Every interaction begins with the client:

- The **client** sends JSON-RPC *requests* and *notifications*.
- The **server** answers each request with a JSON-RPC *response* (a result or error), optionally preceded by *notifications* scoped to that request.

Servers **MUST NOT** initiate JSON-RPC requests, and clients do not send JSON-RPC responses.

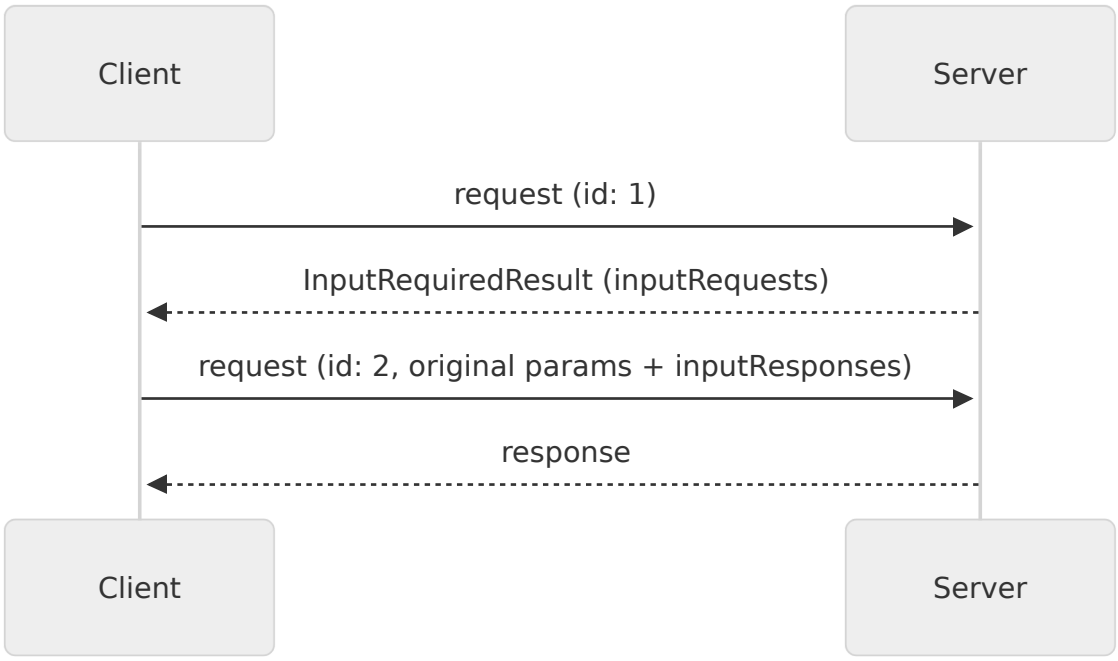
Request and Response

The client sends a request; the server answers it with a result or an error. While the request is in flight, the server **MAY** send notifications scoped to it, such as `notifications/progress` and `notifications/message`.



Multi Round-Trip Requests

When a server needs client input (sampling, elicitation, or roots) to complete a request, it answers with an `InputRequiredResult` and the client retries the request with the matching `inputResponses`. See Multi Round-Trip Requests.



Subscribe and Notify

To receive change notifications (list changes, resource updates), the client sends a `subscriptions/listen` request; the reply is a long-lived stream of the requested notification types. Stream state is scoped to the request: if the underlying channel is lost, the client re-issues the request.

Adding Patterns

All core protocol features are built from these patterns. A protocol revision that adds a pattern defines it on this page. Transports carry new patterns without changes, because patterns are expressed entirely in terms of

requests, responses, and notifications.

5.3.2 Multi Round-Trip Requests

> **NOTE**
>
> Multi Round-Trip Requests (MRTR) was introduced in this version of the MCP specification. This replaces the previous approach of sending server-initiated requests. Servers **MUST** send server-to-client requests (such as `roots/list`, `sampling/createMessage`, or `elicitation/create`) using the MRTR pattern. The previous pattern of server-initiated requests is no longer supported. This is a breaking change.

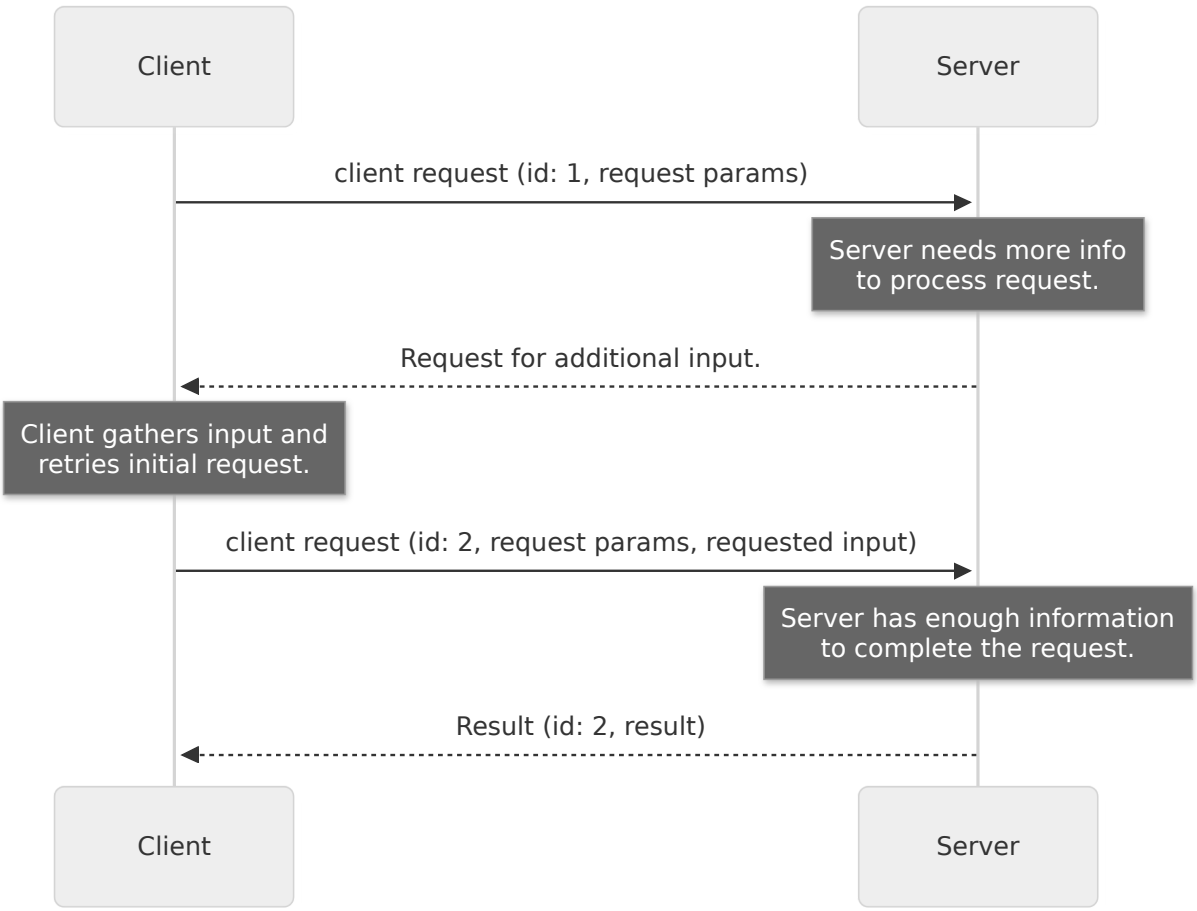
> **NOTE**
>
> For brevity, the request examples on this page omit the `_meta` request metadata (`io.modelcontextprotocol/protocolVersion`, `io.modelcontextprotocol/clientInfo`, and `io.modelcontextprotocol/clientCapabilities`). Every request **MUST** include the required `_meta` fields; see `_meta`.

Multi Round-Trip Requests

The Model Context Protocol (MCP) defines several ways for servers to request additional information from users during the processing of client requests (such as `roots/list`, `sampling/createMessage`, or `elicitation/create`). The **multi round-trip requests** pattern provides a standardized way to handle these server-requests without requiring a shared storage layer across server instances or requiring stateful load balancing.

The high level flow functions as follows:

1. Client sends an initial request to the server with the parameters needed to perform the operation.
2. Server determines that additional information is required to fulfill the request and responds requesting more information.
3. Client gathers the requested information from the user or other sources, then retries the original request including the additional requested information.
4. Server determines it has sufficient information to complete the operation, and responds with the final result.



Core Types

This flow is implemented in MCP using the following Types.

InputRequests

An `InputRequests` object is a map of server-client requests. Keys are server-assigned string identifiers; values are request objects (e.g., `ElicitRequest`, `CreateMessageRequest`, or `ListRootsRequest`).

```
{
  "github_login": {
    "method": "elicitation/create",
    "params": {
      "mode": "form",
      "message": "Please provide your GitHub username",
      "requestedSchema": {
        "type": "object",
        "properties": {
          "name": { "type": "string" }
        },
        "required": ["name"]
      }
    }
  },
  "capital_of_france": {
    "method": "sampling/createMessage",
    "params": {
      "messages": [
        {
          "role": "user",
          "content": {
            "type": "text",
            "text": "What is the capital of France?"
          }
        }
      ],
      "systemPrompt": "You are a helpful assistant.",
      "maxTokens": 100
    }
  }
}
```

InputResponses

An `InputResponses` object is a map of client responses to the server requests. Keys correspond to the keys in the `InputRequests` map; values are the client's result for each request (e.g., `ElicitResult`, `CreateMessageResult`, or `ListRootsResult`).

```
{
  "github_login": {
    "action": "accept",
    "content": {
      "name": "octocat"
    }
  },
  "capital_of_france": {
    "role": "assistant",
    "content": {
      "type": "text",
      "text": "The capital of France is Paris."
    },
    "model": "claude-3-sonnet-20240307",
    "stopReason": "endTurn"
  }
}
```

InputRequiredResult

An `InputRequiredResult` is a type of `Result`, indicating that additional input is needed before the request can be completed.

- `inputRequests` *(optional)*: An `InputRequests` map of server-initiated requests that the client must fulfill.
- `requestState` *(optional)*: An opaque string meaningful only to the server. Clients **MUST NOT** inspect, parse, modify, or make any assumptions about its contents.

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "resultType": "input_required",
    "inputRequests": {
      // Elicitation request.
      "github_login": {
        "method": "elicitation/create",
        "params": {
          "mode": "form",
          "message": "Please provide your GitHub username",
          "requestedSchema": {
            "type": "object",
            "properties": {
              "name": { "type": "string" }
            },
            "required": ["name"]
          }
        }
      },
      // Sampling request.
      "capital_of_france": {
        "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
        }
      }
    },
    "requestState": "AEAD-protected blob"
  }
}
```

Supported Requests

Servers **MAY** send `InputRequiredResult` responses on the following client requests:

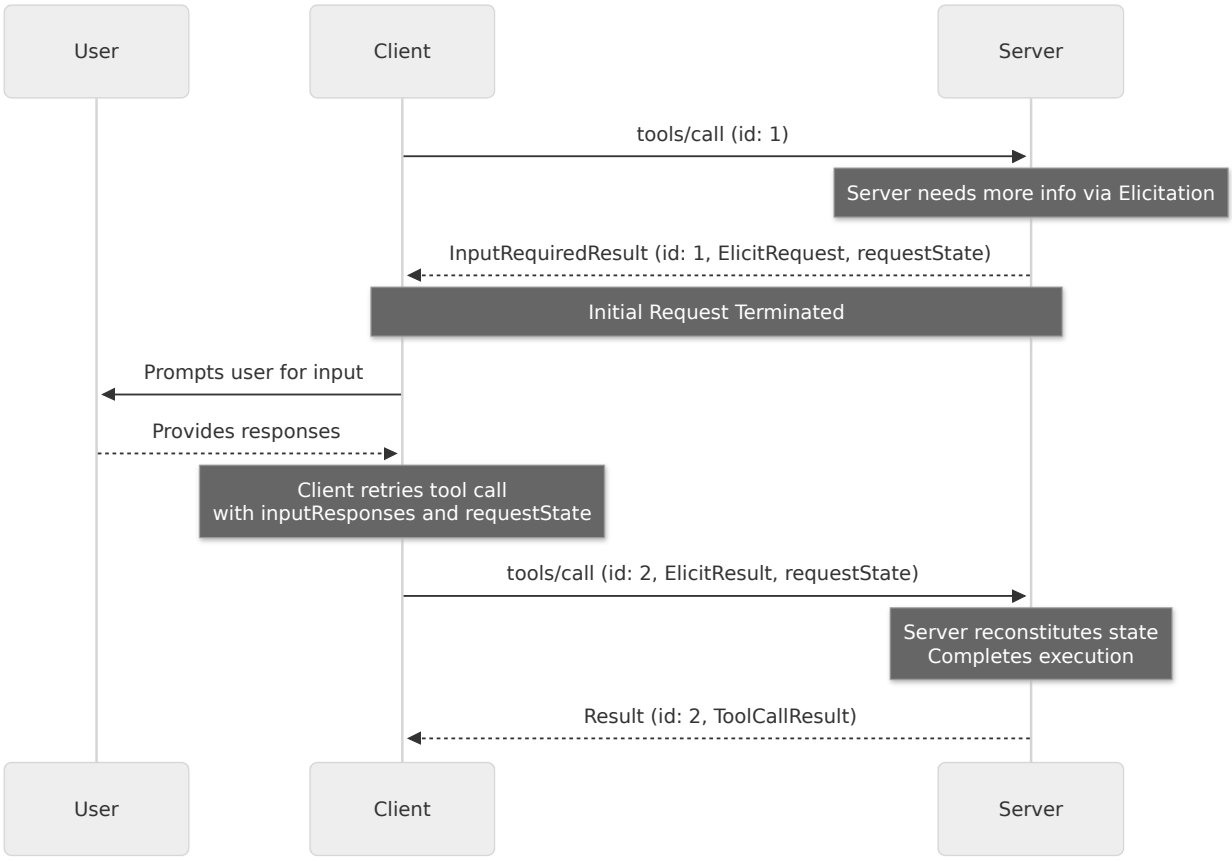
Client Request	Supports InputRequiredResult
`prompts/get`	Yes
`resources/read`	Yes
`tools/call`	Yes

Servers **MUST NOT** send `InputRequiredResult` responses on any other client requests.

Basic Workflow

The basic workflow describes how a server can request additional input from the client as part of a client-server request. In this example we use `tools/call` as the client request, but the same pattern applies to any of the supported requests listed above.

Notably, it allows servers to request additional information without maintaining any server-side state. The server encodes any needed context into the `requestState` field, which the client echoes back on retry.



Note that the requests in each step are completely independent: the server processing the retry does not need any information beyond what is directly present in the retry request.

Server Requirements (Basic Workflow)

1. Servers **MAY** respond to any supported client request with an `InputRequiredResult`.
2. The `InputRequiredResult` **MAY** include an `inputRequests` field.

- `inputRequests` keys are server assigned identifiers and **MUST** be unique within the scope of the request.
- `inputRequests` values are request objects that **MUST** be one of `ElicitRequest`, `CreateMessageRequest`, or `ListRootsRequest`

3. The `InputRequiredResult` **MAY** include a `requestState` field. If specified, this field is an opaque string meaningful only to the server. Servers are free to encode the state in any format (e.g. base64-encoded JSON, encrypted JWT, serialized binary).

4. If a client request contains a `requestState` field, servers **MUST** treat `requestState` as an attacker-controlled input. If `requestState` influences authorization, resource access, or business logic, servers **MUST** protect its integrity (e.g. HMAC or AEAD) and **MUST** reject state that fails verification. Integrity protection **MAY** be omitted only when tampering can cause nothing worse than request failure.

5. To prevent replay, servers **SHOULD** include the following inside the integrity-protected `requestState` payload and verify each on receipt:

   - the authenticated principal, rejecting state presented by a different principal.
   - a short expiry (TTL), rejecting state presented after it lapses;
   - an identifier for the originating request, e.g. the method name and a digest of its salient parameters, rejecting state presented on a request that does not match.

   > **WARNING**
   >
   > Note that these measures bound the replay window and prevent cross-user and cross-request reuse, but do not by themselves guarantee single-use. Servers for which a given `requestState` must be consumed at most once (e.g., one-time redemptions) **MUST** enforce that invariant server-side.

6. Servers **MUST** include at least one of `inputRequests` or `requestState` in every `InputRequiredResult` response.

7. Servers **MUST NOT** send an `inputRequests` that the client has not declared support for in its capabilities. For example, if a client does not declare support for `elicitation`, the server **MUST NOT** include any `elicitation/create` requests in the `inputRequests` field.

8. Servers **MUST NOT** assume that clients will fulfill the `inputRequests` or retry the original request. Servers **MAY** choose to return an `InputRequiredResult` on multiple attempts at the same request if they want to repeatedly prompt the user for information until they have what they need to complete the request.

Client Requirements (Basic Workflow)

1. If a client receives an `InputRequiredResult` that contains the `inputRequests` field, the client **MUST** construct the requested inputs before retrying the original request. If the `InputRequiredResult` does *not* contain the `inputRequests` field, the client **MAY** retry the original request immediately.
2. If an `InputRequiredResult` contains the `requestState` field, the client **MUST** echo back the exact value of that field when retrying the original request. Clients **MUST NOT** inspect, parse, modify, or make any assumptions about the `requestState` contents. If the `InputRequiredResult` does not contain a `requestState` field, the client **MUST NOT** include one in the retry.
3. The JSON-RPC `id` **MUST** be different between the initial request and the retry, as they are independent requests.
4. Both the `inputRequests` and `requestState` fields affect only the client's retry of the original request. They **MUST NOT** be used for any other request that the client may be sending in parallel.

Error Handling

Servers **SHOULD** validate that the data provided by the client is a valid `InputResponses` object and that the information inside can be correctly parsed. Protocol errors (malformed JSON, invalid schema, internal server errors) **SHOULD** return a JSON-RPC error response with an appropriate error code and message.

If additional, unexpected parameters are provided in the `InputResponses` object, the server **SHOULD** ignore any information it does not recognize or need.

If the client fails to send all the information requested in a previous `InputRequests`, and the missing information is necessary for the server to process the request, the server **SHOULD** respond with a new `InputRequiredResult` requesting the missing information again, rather than returning an error.

Security Considerations

Because `requestState` passes through the client, malicious or compromised clients could attempt to modify it to alter server behavior, bypass authorization checks, or corrupt server logic. Servers **MUST** validate request state as described in the server requirements above.

5.3.3 Subscriptions

`subscriptions/listen` opens a long-lived notification stream from the server to the client. Unlike one-off requests, the stream stays open and delivers notifications until the client cancels it. It replaces the former `resources/subscribe` RPC and the HTTP GET endpoint.

Opening a Stream

The client sends a `subscriptions/listen` request with a `notifications` filter specifying which event types it wants to receive. The server **MUST NOT** send notification types the client has not explicitly requested.

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "subscriptions/listen",
  "params": {
    "_meta": {
      "io.modelcontextprotocol/protocolVersion": "2026-07-28",
      "io.modelcontextprotocol/clientInfo": {
        "name": "ExampleClient",
        "version": "1.0.0"
      },
      "io.modelcontextprotocol/clientCapabilities": {}
    },
    "notifications": {
      "toolsListChanged": true,
      "resourceSubscriptions": ["file:///project/config.json"]
    }
  }
}
```

Notification Filter

Field	Type	Description
`toolsListChanged`	`boolean`	Receive `notifications/tools/list_changed` when tools change
`promptsListChanged`	`boolean`	Receive `notifications/prompts/list_changed` when prompts change
`resourcesListChanged`	`boolean`	Receive `notifications/resources/list_changed` when list changes
`resourceSubscriptions`	`string[]`	Receive `notifications/resources/updated` for these resource URIs

All fields are optional. Omitting a field is equivalent to not subscribing to that notification type.

Acknowledgment

The server **MUST** send `notifications/subscriptions/acknowledged` as the first message carrying the subscription's ID in `_meta` under `io.modelcontextprotocol/subscriptionId`, and **MUST NOT** send any notification on the subscription before it. On stdio, where every subscription shares one channel, this ordering is defined per subscription ID and not per channel: messages belonging to other subscriptions **MAY** be interleaved before it.

The `notifications` field in the acknowledgment reflects the subset the server agreed to honor. Notification types the server does not support are omitted.

```
{
  "jsonrpc": "2.0",
  "method": "notifications/subscriptions/acknowledged",
  "params": {
    "_meta": {
      "io.modelcontextprotocol/subscriptionId": 1
    },
    "notifications": {
      "toolsListChanged": true,
      "resourceSubscriptions": ["file:///project/config.json"]
    }
  }
}
```

The client **SHOULD** check the acknowledged filter against what it requested and handle any unsupported types gracefully.

Receiving Notifications

All notifications delivered on the stream carry `io.modelcontextprotocol/subscriptionId` in `_meta`, identifying the `subscriptions/listen` request that opened the stream. The value is the JSON-RPC ID of the `subscriptions/listen` request. In the examples above, the request used `"id": 1`, so the acknowledgment and all subsequent notifications carry the subscription ID `1`. On stdio, where all messages share a single channel, clients **MUST** use this field to correlate notifications with their originating subscription.

```
{
  "jsonrpc": "2.0",
  "method": "notifications/resources/updated",
  "params": {
    "_meta": {
      "io.modelcontextprotocol/subscriptionId": 1
    },
    "uri": "file:///project/config.json"
  }
}
```

Multiple Concurrent Subscriptions

A client **MAY** have multiple active subscriptions concurrently — for example, one listening for tools-list changes and another for resource updates. Each subscription is identified by the JSON-RPC request ID of its `subscriptions/listen` request, and every notification on the stream carries that ID in `io.modelcontextprotocol/subscriptionId` so clients can demultiplex them.

Cancellation

A subscription ends when:

- The **client** cancels it — close the SSE stream (HTTP) or send `notifications/cancelled` referencing the `subscriptions/listen` request ID (stdio).
- The **server** tears it down (e.g., during shutdown) — it **SHOULD** send a successful `subscriptions/listen` response to signal a graceful end (see Graceful Closure), then close the stream.
- The underlying transport closes (HTTP timeout, TCP disconnect, stdio process exit).

Graceful Closure

When the server ends a subscription on its own initiative (for example, during shutdown), it **SHOULD** respond to the original `subscriptions/listen` request with a completion result before closing the stream. The result carries no method-specific data beyond the standard result fields and subscription metadata. This is the JSON-RPC response to the long-lived request, correlated by its `id`, and signals that the subscription ended gracefully — as opposed to an abrupt transport drop, which carries no response.

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "resultType": "complete",
    "_meta": {
      "io.modelcontextprotocol/subscriptionId": 1
    }
  }
}
```

Like every other message on the stream, the response carries `io.modelcontextprotocol/subscriptionId` in `_meta`, identifying which subscription it closes. The value matches the JSON-RPC `id` of the originating `subscriptions/listen` request.

A client that receives this response knows the subscription closed cleanly; a transport that closes without it indicates an unexpected disconnect, which the client **MAY** treat as a trigger to reconnect.

On **stdio**, if the connection is terminated and then re-established, the client **MUST** re-send `subscriptions/listen` to re-establish its subscriptions — the server holds no subscription state across reconnections.

See Cancellation for the full rules.

5.3.4 Cancellation

The Model Context Protocol (MCP) supports optional cancellation of in-progress requests through notification messages. A client **SHOULD** send a cancellation notification to indicate that a request it previously issued should be terminated.

A server **MUST** send `notifications/cancelled` referencing a `subscriptions/listen` request ID when it tears down that subscription stream (see Subscriptions). Servers **MUST NOT** send `notifications/cancelled` for any other purpose.

Cancellation Flow

When a client 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"
  }
}
```

Transport-Specific Cancellation

How a client signals cancellation depends on the transport:

- **Streamable HTTP**: Closing the SSE response stream is the cancellation signal. The server **MUST** treat a client disconnect as cancellation of that request. No `notifications/cancelled` message is required or expected.
- **stdio**: There is no per-request stream to close. The client **MUST** send a `notifications/cancelled` notification referencing the request ID.

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** cancel the request and stop waiting for a response. As described in Transport-Specific Cancellation, this means:

- **Streamable HTTP**: closing the response stream for the request, which constitutes cancellation.
- **stdio**: sending a `notifications/cancelled` notification referencing the request ID.

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.

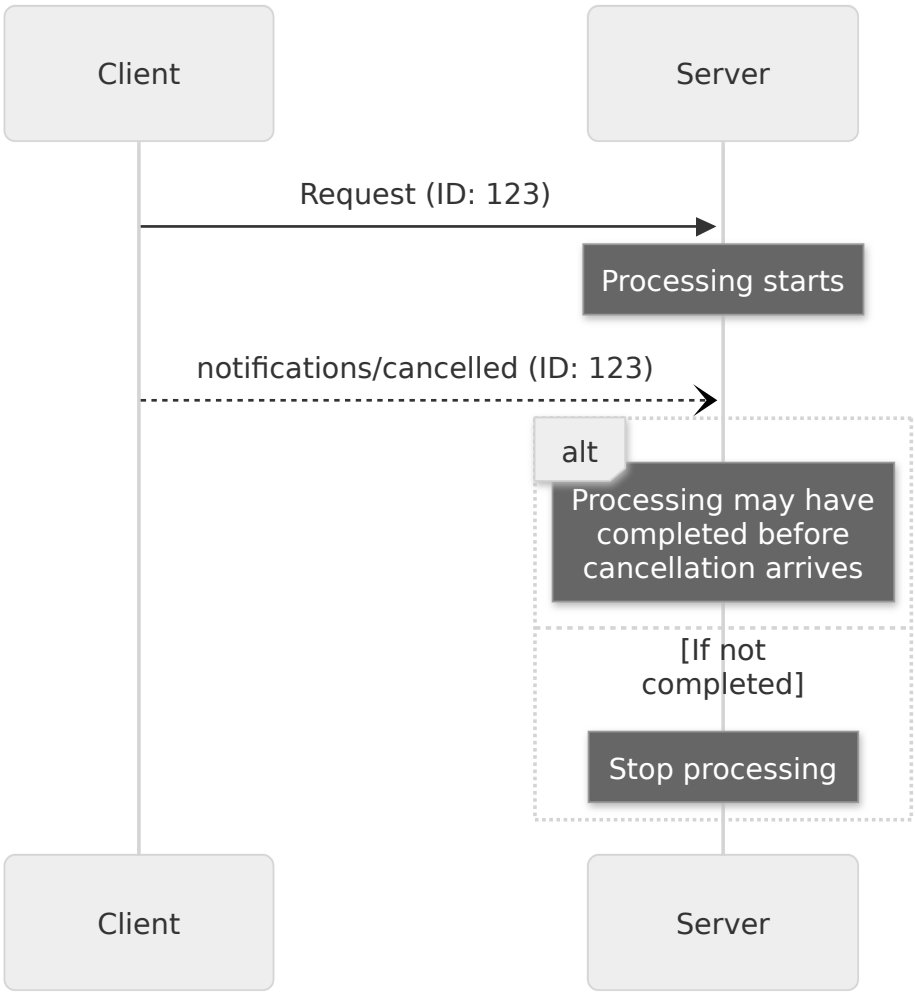
Behavior Requirements

1. Cancellation notifications **MUST** only reference requests that:
    - Were previously issued by the client
    - Are believed to still be in-progress
2. Server-sent cancellation notifications **MUST** reference a `subscriptions/listen` request, to terminate that subscription stream
3. Servers receiving cancellation notifications **SHOULD**:
    - Stop processing the cancelled request
    - Free associated resources
    - Not send a response for the cancelled request
4. Servers **MAY** ignore cancellation notifications if:
    - The referenced request is unknown
    - Processing has already completed
    - The request cannot be cancelled
5. The client **SHOULD** ignore any response to the cancelled 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.

5.3.5 Progress

The Model Context Protocol (MCP) supports optional progress tracking for long-running operations through notification messages. The server **MAY** send progress notifications to report the status of requests the client has issued.

Progress Flow

When a client 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 client using any means, but **MUST** be unique across all active requests.

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

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

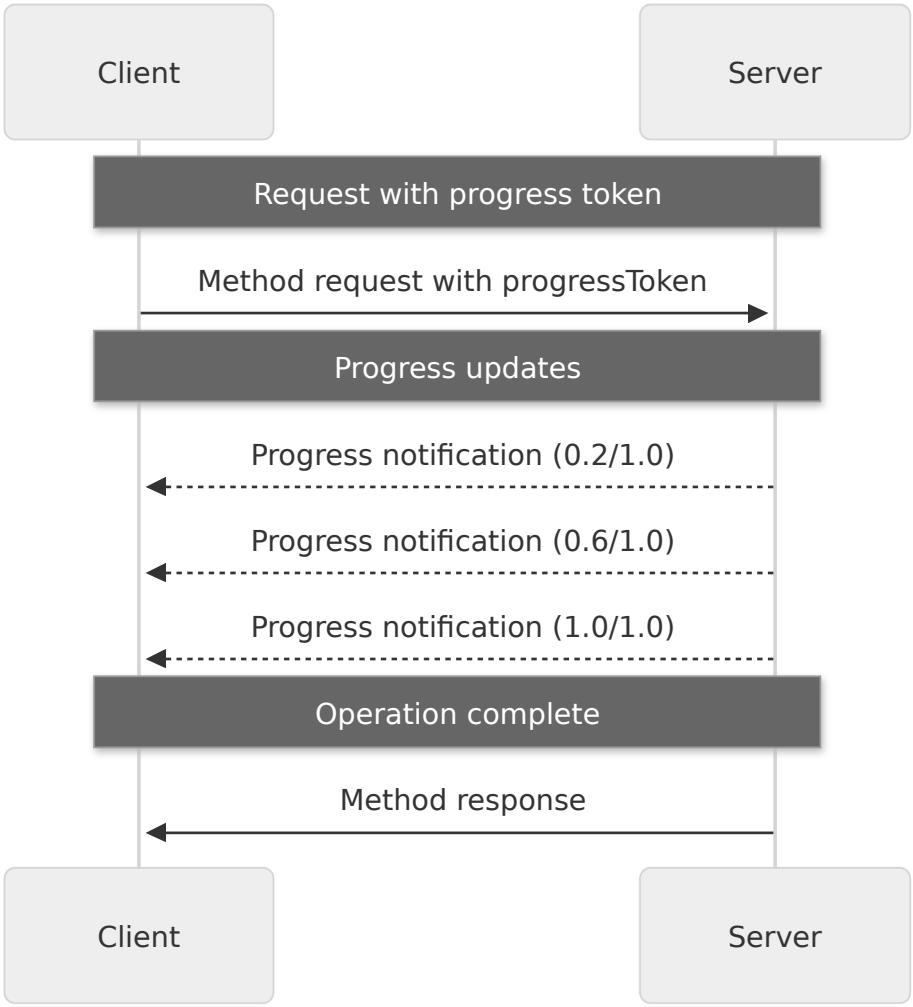
- The original progress token
- The current progress value so far
- An optional "total" value
- An optional "message" value

```
{
  "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. Servers receiving a request with a progress token **MAY**:
    - Choose not to send any progress notifications
    - Send notifications at whatever frequency they deem appropriate
    - Omit the total value if unknown



Implementation Notes

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

5.4 Transports

5.4.1 Overview

This page defines what a transport must provide to carry MCP messages, the standard transport bindings, and the requirements for defining new ones.

Protocol semantics are identical on every transport. A transport is a **binding**: it defines how messages are framed and delivered, how request metadata is carried, and how cancellation and termination are signaled. It does not define what the messages mean: the message patterns are part of the core protocol and are the same on every binding. The binding pages specify the standard transports:

1. stdio: newline-delimited messages over the standard streams of a client-launched subprocess.
2. Streamable HTTP: each message is an HTTP POST to a single MCP endpoint; replies arrive as a JSON object or a request-scoped SSE stream.

It is also possible for clients and servers to implement custom transports.

Messages

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

A binding **MUST** deliver client-sent *requests* and *notifications* to the server, and server-sent *responses* and *notifications* to the client. No other message direction exists: per the message patterns, servers do not initiate JSON-RPC requests and clients do not send JSON-RPC responses.

Request Metadata

All protocol metadata travels in the message body: every request carries its protocol version and client capabilities in `_meta.io.modelcontextprotocol/*` fields.

A binding **MAY** additionally mirror selected body fields into envelope metadata. The Streamable HTTP transport mirrors them into HTTP headers so that intermediaries can route and inspect requests without parsing the body. The body remains the source of truth; bindings that mirror metadata define how mismatches are rejected.

Cancellation

Each binding defines how a client abandons an in-flight request: on stdio the client sends a `notifications/cancelled` notification; on Streamable HTTP it closes the request's response stream. The protocol-level rules are the same everywhere; see Cancellation.

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** preserve the JSON-RPC message format, the message patterns, and the per-request metadata model. Custom transports **SHOULD** document their connection establishment, message framing, and cancellation patterns to aid interoperability.

Custom transports that run over a reliable bidirectional byte stream (e.g., Unix domain sockets or TCP) **SHOULD** reuse the stdio framing rather than defining a new one: the stdio binding is just newline-delimited JSON-RPC over a byte stream, and only its process-lifecycle rules are specific to standard streams.

Backward Compatibility

Earlier protocol revisions established a connection-scoped session with an `initialize` handshake and allowed servers to initiate JSON-RPC requests. Clients and servers that interoperate with those revisions detect the counterpart's era and fall back as described in Versioning: Backward Compatibility, which includes a compatibility matrix for implementors. Each binding page describes its transport-specific detection mechanics.

5.4.2 stdio

In the **stdio** transport, the client launches the MCP server as a subprocess. The two ends communicate over the subprocess's standard streams:

- The server reads JSON-RPC messages from `stdin` and writes JSON-RPC messages to `stdout`.
- Each message is a single JSON-RPC request, notification, or response.
- Messages are delimited by newlines, and **MUST NOT** contain embedded newlines.
- The server **MAY** write UTF-8 strings to `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.

Standard streams are the canonical channel, but nothing in this binding depends on them except the process lifecycle. The wire format (one newline-delimited JSON-RPC message per line over a reliable bidirectional byte stream) works unchanged over Unix domain sockets, TCP connections, or any similar channel. Custom transports built on such streams **SHOULD** reuse this framing and the message rules on this page; only the subprocess-specific aspects (launch, `stderr`, shutdown by closing the stream, process restart) need channel-specific equivalents.

Sending Messages

The client sends messages by writing JSON-RPC *requests* and *notifications* to the server's `stdin`, one message per line. The client **MUST NOT** write JSON-RPC *responses*.

Receiving Messages

The client reads server messages from `stdout`, one message per line. All messages share this single channel; there are no per-request streams.

The server writes three kinds of messages:

1. *Responses* to client requests, correlated by JSON-RPC `id`.
2. *Notifications* that relate to an in-flight request, such as `notifications/progress` and `notifications/message`.
3. *Notifications* delivered for an active `subscriptions/listen` request. Clients **MUST** correlate these using the `io.modelcontextprotocol/subscriptionId` field in `_meta`; see `SubscriptionsListenRequest`.

The server **MUST NOT** write JSON-RPC *requests* to `stdout`. Server-to-client interactions are carried in `InputRequiredResult` replies; see Multi Round-Trip Requests.

Request Metadata

All request metadata for the stdio transport is carried inline in the JSON-RPC message body. The protocol version, per-request capabilities, and optional client identity live in `_meta.io.modelcontextprotocol/*`; the method name and arguments live where JSON-RPC puts them. There is no header layer.

Cancellation

To cancel an in-flight request, the client **MUST** send a `notifications/cancelled` notification referencing the request's ID. Because stdio is a single shared bidirectional channel, there is no per-request stream to close. Servers **SHOULD** stop work on a cancelled request as soon as practical and **MUST NOT** send any further messages for it. See Cancellation for the full rules.

Shutdown

The client **SHOULD** initiate shutdown by:

1. Closing the input stream to the child process (the server).
2. Waiting for the server to exit.
3. If the server does not exit within a reasonable time, forcibly terminating the process using the mechanism appropriate for the operating system.

On POSIX systems, forced termination typically escalates from `SIGTERM` to `SIGKILL`. On Windows, where POSIX signals are not available, clients can use `TerminateProcess` or Job Objects.

Servers **SHOULD** exit promptly when their standard input is closed or reads return end-of-file. This is the primary graceful-shutdown signal and the only portable one, so honoring it reduces the need for forced termination.

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

Unexpected Termination

If the server process exits unexpectedly, the client **SHOULD** restart it. Because the protocol is stateless, any in-flight requests are simply lost and the client can retry them against the fresh process. Active `subscriptions/listen` streams must also be re-established after restart.

Backward Compatibility

A client that supports both modern (per-request-metadata) MCP versions and a legacy version that requires an `initialize` handshake **SHOULD** probe with `server/discover` before sending any other request, setting its preferred modern version in `_meta`. The probe has three possible outcomes:

- The server returns a `DiscoverResult`: the server is modern. Select a mutually supported version from `supportedVersions` and continue.
- The server returns a recognized modern JSON-RPC error such as `UnsupportedProtocolVersionError`: the server is modern but does not support the requested version. Use one of the versions in its advertised `supported` list. Do **not** fall back to `initialize`.
- The server returns any other error, or does not respond within a reasonable timeout: the server is legacy. Fall back to the `initialize` handshake.

The fallback **MUST NOT** be keyed to one specific error code: legacy servers respond to unknown pre-`initialize` requests with implementation-defined errors (commonly `-32601` or `-32602`) or not at all.

A client that only supports modern versions does not need to probe, but probing is still **RECOMMENDED**: some legacy servers do not validate that a request arrives after `initialize` and would process an era-ambiguous method (such as `tools/call`) under legacy semantics. Probing yields a deterministic failure instead.

See Versioning: Backward Compatibility for the era model and a compatibility matrix for implementors.

5.4.3 Streamable HTTP

> **INFO**
>
> Streamable HTTP was introduced in protocol version 2025-03-26 as a replacement for the HTTP+SSE transport from protocol version 2024-11-05.

> **INFO**
>
> Revision 2026-07-28 changed the behavior of Streamable HTTP. Clients must ensure they handle backwards compatibility correctly. Changes included:
>
> - Removal of the GET stream endpoint.
> - Removal of protocol-level sessions.
>
> See the changelog and Backward Compatibility below.

In the **Streamable HTTP** transport, the server operates as an independent process that can handle multiple client connections. At a glance:

- The server exposes a single HTTP endpoint (the **MCP endpoint**) that accepts POST.
- The client sends every JSON-RPC request or notification as its own HTTP POST.
- The server answers each request with either a single JSON object or a Server-Sent Events (SSE) stream scoped to that request, carrying request-related notifications followed by the final response.
- Server-to-client interactions (sampling, elicitation, roots) are embedded in results as input requests per Multi Round-Trip Requests (MRTR) (SEP-2322).
- Long-lived change notifications (such as list changes and resource updates) are delivered on the response stream of a `subscriptions/listen` request.

See Message Flow for sequence diagrams of these interactions.

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

Security & Endpoint

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

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.
2. The client **MUST** include an `Accept` header listing both `application/json` and `text/event-stream` as supported content types.
3. The client **MUST** include the request metadata headers on each POST request.
4. The body of the HTTP POST **MUST** be a single JSON-RPC *request* or *notification*. The client **MUST NOT** send JSON-RPC *responses*.
5. If the body is a JSON-RPC *notification*:
    - If the server accepts it, the server **MUST** return HTTP status code `202 Accepted` with no body.
    - If the server cannot accept it, 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`.
6. If the body is a JSON-RPC *request*, the server **MUST** return either `Content-Type: application/json` (a single JSON object) or `Content-Type: text/event-stream` (an SSE response stream). The client **MUST** support both.

> **NOTE**
>
> This revision of the core protocol defines no client-to-server *notifications* over Streamable HTTP. The only client-sent notification in the core protocol, `notifications/cancelled`, is used only on the stdio transport; on Streamable HTTP, closing the SSE response stream is itself the cancellation signal and no `notifications/cancelled` message is expected (see Cancellation). The notification rules above describe the transport mechanics for a notification POST; header requirements for notification POSTs are not defined by this revision.

Receiving Messages

When the server returns an SSE response stream (`Content-Type: text/event-stream`):

- The server **MAY** send JSON-RPC *notifications* — for example, `notifications/progress` or `notifications/message` — before the final response. These notifications **MUST** relate to the originating client request.
- The server **MUST NOT** send independent JSON-RPC *requests* on this stream. Server-to-client interactions (sampling, elicitation, list-roots) are embedded as input requests inside an `InputRequiredResult` per MRTR (SEP-2322), not delivered as separate requests on this or any other stream. This is a change from Streamable HTTP in protocol versions `2025-03-26` through `2025-11-25`, where servers could send such requests on SSE streams.
- The final JSON-RPC *response* **SHOULD** terminate the stream.

Long-lived notification streams are obtained by sending a `subscriptions/listen` request. The server's response is itself an SSE stream that stays open and delivers the change notifications the client opted in to (such as `notifications/tools/list_changed` or `notifications/resources/updated`). Request-scoped notifications like `notifications/progress` and `notifications/message` are **not** delivered on the listen stream — they flow only on the response stream of the request they relate to.

When initiating an SSE stream, servers **SHOULD** include the `X-Accel-Buffering: no` header in the HTTP response. This instructs reverse proxies (such as nginx) to disable response buffering, ensuring that SSE events are delivered to clients immediately rather than being held in a buffer. Without this header, proxies may accumulate messages before sending them to the client, introducing unwanted latency and potentially breaking the real-time nature of SSE communication.

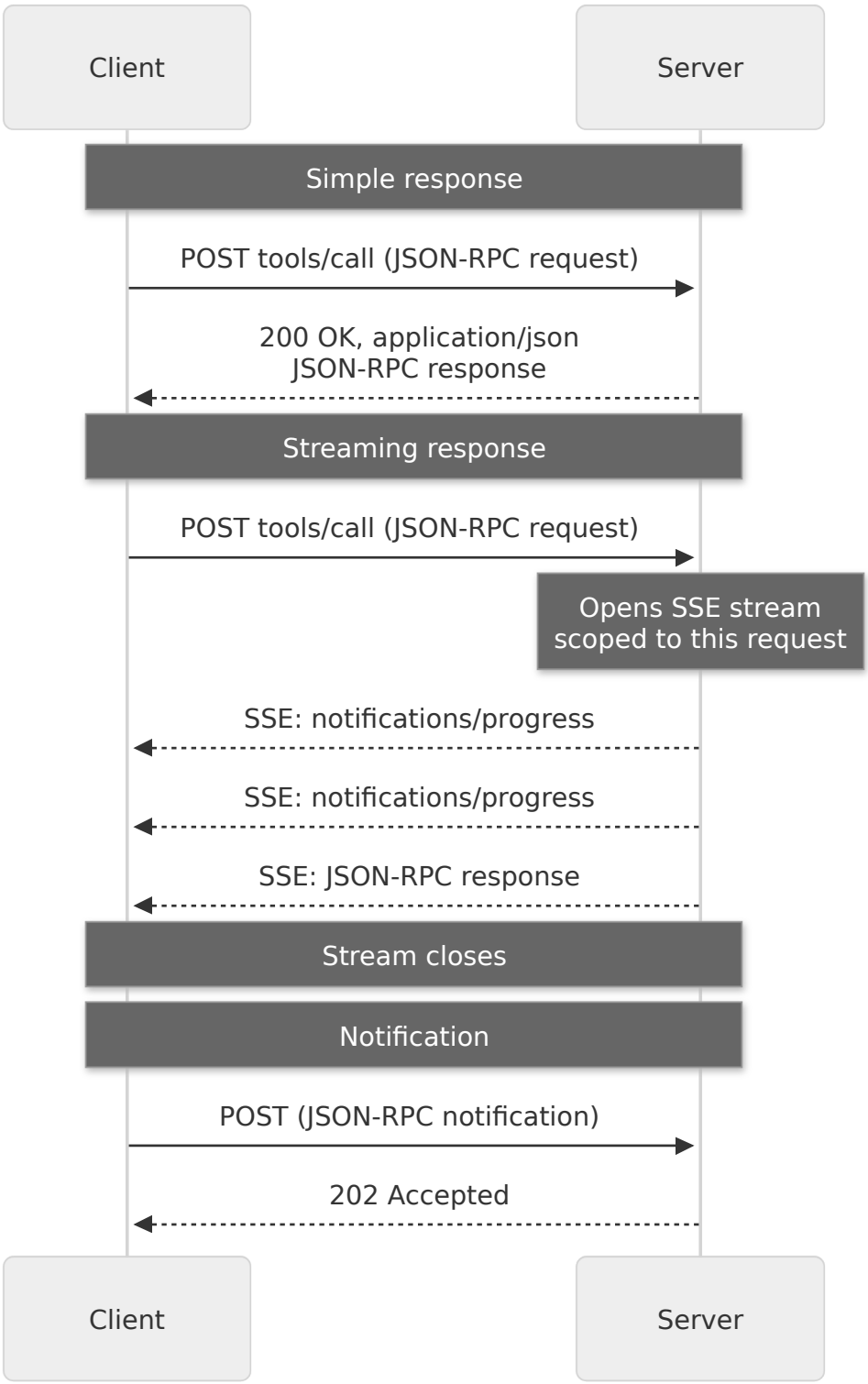
> **NOTE**
>
> For long-lived streams — in particular the `subscriptions/listen` response stream — servers are encouraged to periodically emit an SSE comment line (a line beginning with a colon, e.g. `:\r\n`) as a keep-alive. This keeps the connection from being closed by intermediaries or client idle timeouts during quiet periods when no notifications are flowing. Per the SSE specification, any line beginning with a colon is a comment that carries no event data; clients must ignore such lines and must not treat them as malformed input.

Resumable SSE streams via `Last-Event-ID` are not supported.

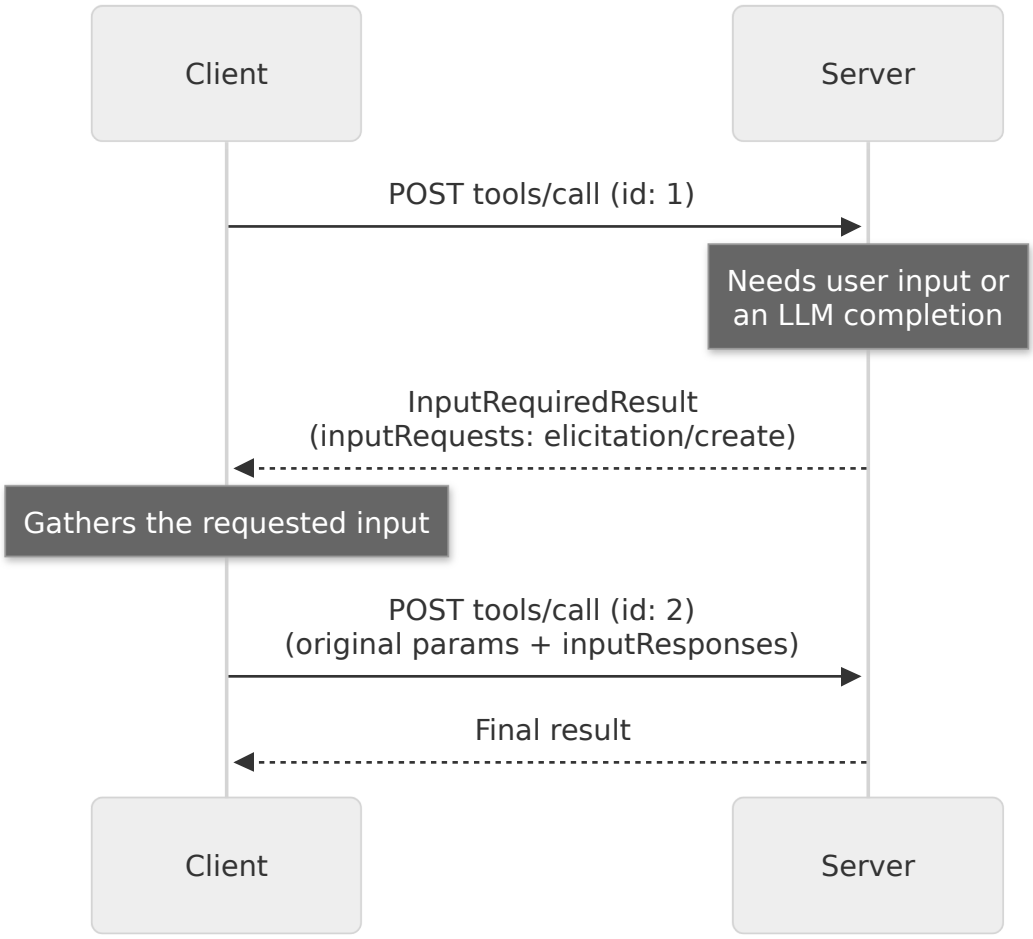
Message Flow

The following diagrams illustrate the message flows on a single MCP endpoint.

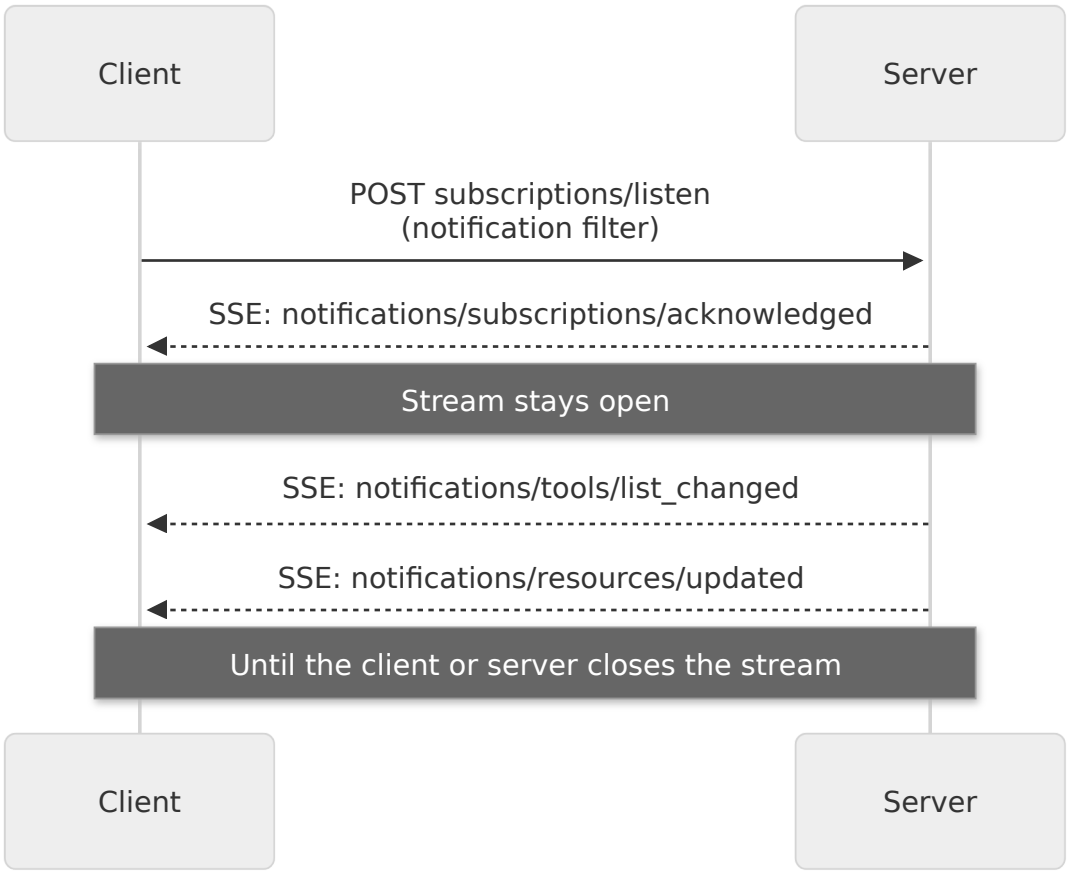
Requests and responses. Each request is its own POST; the server chooses per request whether to respond with a single JSON object or an SSE stream:



Server-to-client interactions (MRTR). When the server needs input from the client — sampling, elicitation, or roots — it does not send its own JSON-RPC request. It returns an `InputRequiredResult` containing `inputRequests`, and the client retries the original request with the matching `inputResponses` (see Multi Round-Trip Requests):



Change notifications. Clients that want server-initiated change notifications open a long-lived stream with `subscriptions/listen`; the response stream stays open and carries only the notification types the client opted in to:



Cancellation

Closing the SSE response stream **MUST** be treated by the server as cancellation of that request. Because each request has its own response stream, the transport-level disconnect is unambiguous. The server **SHOULD** stop work on the cancelled request as soon as practical and **MUST NOT** send any further messages for it. See Cancellation for the full rules.

Request Metadata

The Streamable HTTP transport mirrors selected JSON-RPC body fields into HTTP headers so that intermediaries (load balancers, gateways, observability tooling) can route and inspect requests without parsing the body.

Protocol Version Header

Every POST request to the MCP endpoint **MUST** include an `MCP-Protocol-Version` header.

For example: `MCP-Protocol-Version: 2026-07-28`

The header value **MUST** match the `io.modelcontextprotocol/protocolVersion` field carried in the request body's `_meta`. If the values do not match, the server **MUST** reject the request with `400 Bad Request` and a `HeaderMismatch` JSON-RPC error (see Server Validation).

If the server does not implement the requested protocol version (whether the version is unknown to the server, or is a known version the server has chosen not to support), it **MUST** respond with `400 Bad Request` and an `UnsupportedProtocolVersionError` listing its supported versions. See Versioning: Protocol Version Negotiation for the negotiation flow.

If the server does not implement the requested RPC method, it **MUST** respond with `404 Not Found` and a JSON-RPC error with code `-32601` (`Method not found`). The JSON-RPC error body distinguishes this case from a `404` returned by a legacy HTTP+SSE server that does not host the modern MCP endpoint (see Backward Compatibility).

A server that supports clients implementing protocol versions earlier than `2025-06-18` (which did not define the `MCP-Protocol-Version` header) **MAY** treat a request that omits the header as protocol version `2025-03-26`. A server that does not support such clients **MUST** reject a request without the header per Server Validation.

Standard Request Headers

Header Name	Source Field	Required For
`Mcp-Method`	`method`	All requests
`Mcp-Name`	`params.name` or `params.uri`	`tools/call`, `resources/read`, `prompts/get` requests

These headers are **REQUIRED** for compliance.

If the `Mcp-Name` source value cannot be safely represented as a plain ASCII header value, clients **MUST** encode it using the Base64 sentinel format described in Value Encoding.

`tools/call` request:

```
POST /mcp HTTP/1.1
Content-Type: application/json
MCP-Protocol-Version: 2026-07-28
Mcp-Method: tools/call
Mcp-Name: get_weather

{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "tools/call",
  "params": {
    "name": "get_weather",
    "arguments": {
      "location": "Seattle, WA"
    },
    "_meta": {
      "io.modelcontextprotocol/protocolVersion": "2026-07-28",
      "io.modelcontextprotocol/clientInfo": {
        "name": "ExampleClient",
        "version": "1.0.0"
      },
      "io.modelcontextprotocol/clientCapabilities": {}
    }
  }
}
```

`resources/read` request:

```
POST /mcp HTTP/1.1
Content-Type: application/json
MCP-Protocol-Version: 2026-07-28
Mcp-Method: resources/read
Mcp-Name: file:///projects/myapp/config.json

{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "resources/read",
  "params": {
    "uri": "file:///projects/myapp/config.json",
    "_meta": {
      "io.modelcontextprotocol/protocolVersion": "2026-07-28",
      "io.modelcontextprotocol/clientInfo": {
        "name": "ExampleClient",
        "version": "1.0.0"
      },
      "io.modelcontextprotocol/clientCapabilities": {}
    }
  }
}
```

Custom Headers from Tool Parameters

MCP servers **MAY** designate specific tool parameters to be mirrored into HTTP headers using an `x-mcp-header` extension property in the parameter's schema within the tool's `inputSchema`. See Tool Definitions for details on how to annotate tool parameters.

While the use of `x-mcp-header` is optional for servers, clients **MUST** support this feature. When a server's tool definition includes `x-mcp-header` annotations, conforming clients **MUST** mirror the designated parameter values into HTTP headers.

Schema Extension

The `x-mcp-header` property specifies the name portion used to construct the header name `Mcp-Param-{name}`.

Constraints on `x-mcp-header` values:

- **MUST NOT** be empty
- **MUST** match HTTP field-name token syntax (`1*tchar` , RFC 9110 Section 5.1)
- **MUST NOT** contain control characters, including carriage return (CR, `\r`) or line feed (LF, `\n`)
- **MUST** be case-insensitively unique among all `x-mcp-header` values in the `inputSchema`
- **MUST** only be applied to parameters with primitive types (integer, string, boolean). Parameters with type `number` are not permitted. Integer values **MUST** be within the safe range for integers represented using IEEE754 double-precision floating point numbers ($-2^{53}+1$ to $2^{53}-1$)
- **MUST** only be applied to properties that are *statically reachable* from the schema root: reachable via a chain consisting solely of `properties` keys. The chain **MUST NOT** pass through `items` (or any other array keyword), composition keywords (`oneOf` , `anyOf` , `allOf` , `not`), conditional keywords (`if` / `then` / `else`), or `$ref` . Nested object properties are permitted as long as every step in the chain is a `properties` key. An `x-mcp-header` annotation anywhere else makes the annotation — and thus the tool definition — invalid.

Header extraction is defined as reading the instance value at the exact property path of the annotated property (the chain of `properties` keys leading to it). If no value is present at that path in the call arguments, the header is omitted.

Clients using the Streamable HTTP transport **MUST** reject tool definitions where any `x-mcp-header` value violates these constraints. Rejection means the client **MUST** exclude the invalid tool from the result of `tools/list`. Clients **SHOULD** log a warning when rejecting a tool definition, including the tool name and the reason for rejection. This ensures that a single malformed tool definition does not prevent other valid tools from being used. Clients using other transports (e.g., stdio) **MAY** ignore `x-mcp-header` annotations entirely.

Example tool definition:

```
{
  "name": "execute_sql",
  "description": "Execute SQL on Google Cloud Spanner",
  "inputSchema": {
    "type": "object",
    "properties": {
      "region": {
        "type": "string",
        "description": "The region to execute the query in",
        "x-mcp-header": "Region"
      },
      "query": {
        "type": "string",
        "description": "The SQL query to execute"
      }
    },
    "required": ["region", "query"]
  }
}
```

Resulting HTTP request:

```
POST /mcp HTTP/1.1
Content-Type: application/json
MCP-Protocol-Version: 2026-07-28
Mcp-Method: tools/call
Mcp-Name: execute_sql
Mcp-Param-Region: us-west1

{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "tools/call",
  "params": {
    "_meta": {
      "io.modelcontextprotocol/protocolVersion": "2026-07-28",
      "io.modelcontextprotocol/clientInfo": {
        "name": "ExampleClient",
        "version": "1.0.0"
      },
      "io.modelcontextprotocol/clientCapabilities": {}
    },
    "name": "execute_sql",
    "arguments": {
      "region": "us-west1",
      "query": "SELECT * FROM users"
    }
  }
}
```

Value Encoding

Clients **MUST** encode parameter values before including them in HTTP headers to ensure safe transmission and prevent injection attacks.

Type conversion: Convert the parameter value to its string representation:

- `string`: Use the value as-is
- `integer`: Convert to decimal string representation (e.g., `42`, `-7`)
- `boolean`: Convert to lowercase `"true"` or `"false"`

Per RFC 9110, HTTP header field values must consist of visible ASCII characters (0x21-0x7E), space (0x20), and horizontal tab (0x09). When a value cannot be safely represented as a plain ASCII header value (e.g., it contains non-ASCII characters, control characters, or has leading/trailing whitespace), clients **MUST** use Base64 encoding of the UTF-8 representation with the following format:

```
Mcp-Param-{Name}: =?base64?{Base64EncodedValue}?=
```

The same encoding rule applies to the `Mcp-Name` header value. Tool and prompt names are only **SHOULD**-constrained to header-safe characters, so a name (or resource URI) outside the safe set is carried as:

```
Mcp-Name: =?base64?{Base64EncodedValue}?=
```

The prefix `=?base64?` and suffix `?=` indicate that the value is Base64-encoded. These markers are case-sensitive and **MUST** appear exactly as shown (lowercase). Servers and intermediaries that need to inspect these values **MUST** decode them accordingly. In particular, servers **MUST** decode an encoded `Mcp-Name` or `Mcp-Param-{Name}` value before comparing it to the corresponding request body value during Server Validation.

To avoid ambiguity, clients **MUST** also Base64-encode any plain-ASCII value that matches the sentinel pattern (i.e., starts with `=?base64?` and ends with `?=`).

Encoding examples:

Original Value	Reason	Encoded Header Value
`"us-west1"`	Plain ASCII	`Mcp-Param-Region: us-west1`
`"Hello, 世界"`	Contains non-ASCII	`Mcp-Param-Greeting: =?base64?SGVsbG8sIOS4lueVjA==?=`
`" padded "`	Leading/trailing spaces	`Mcp-Param-Text: =?base64?IHBhZGRlZCA=?=`
`"line1\nline2"`	Contains newline	`Mcp-Param-Text: =?base64?bGluZTEKbGluZTI=?=`
`"=?base64?literal?="`	Matches sentinel pattern	`Mcp-Param-Val: =?base64?PT9iYXNlNjQ/bGl0ZXJhbD89?=`

Client Behavior

When constructing a `tools/call` request via HTTP transport, the client **MUST**:

1. Extract the values for any standard headers from the request body (e.g., `method`, `params.name`, `params.uri`).
2. Append the `Mcp-Method` header and, if applicable, `Mcp-Name` header to the request.
3. Inspect the tool's `inputSchema` for properties marked with `x-mcp-header` and extract the value at each annotated property's exact property path, omitting the header when no value is present (see Schema Extension).
4. Encode the values according to the Value Encoding rules.
5. Append a `Mcp-Param-{Name}: {Value}` header to the request.

If the server rejects a request with a `HeaderMismatch` error because required `Mcp-Param-*` headers are missing or do not match the body, the client **SHOULD** call `tools/list` to check for changes to the tool's `inputSchema`, then retry the original request with the appropriate headers.

Server Behavior for Custom Headers

Intermediate servers that do not recognize an `Mcp-Param-{Name}` header **MUST** forward it and otherwise ignore it, as required by the HTTP Semantics RFC.

Servers **MUST** reject requests with a recognized `Mcp-Param-{Name}` header that contains invalid characters (see Value Encoding).

Any server that processes the message body **MUST** validate that encoded header values, after decoding if Base64-encoded, match the corresponding values in the request body. Servers **MUST** reject requests with a `400 Bad Request` HTTP status and JSON-RPC error code `-32020` (`HeaderMismatch`) if any validation fails.

Scenario	Client Behavior	Server Behavior
Parameter value provided	Client MUST include the header	Server MUST validate header matches body
Parameter value is `null`	Client MUST omit the header	Server MUST NOT expect the header
Parameter not in arguments	Client MUST omit the header	Server MUST NOT expect the header
Client omits header but value is in body	Non-conforming client	Server MUST reject the request

Case Sensitivity

Header names (called "field names" in RFC 9110) are case-insensitive. Clients and servers **MUST** use case-insensitive comparisons for header names. Header *values* (such as method names) are case-sensitive.

Server Validation

Servers that process the request body **MUST** reject requests where the values specified in the headers do not match the corresponding values in the request body. This prevents potential security vulnerabilities when different components in the network rely on different sources of truth (e.g., a load balancer routing on the header value while the MCP server executes based on the body value).

> **NOTE**
>
> When validating integer parameter values, servers **SHOULD** compare the header value and the body value numerically rather than as strings (e.g., `42.0` and `42` are considered equal).

When rejecting a request due to header validation failure, servers **MUST** return HTTP status `400 Bad Request` and **MUST** include a JSON-RPC error response using the following error code:

Code	Name	Description
`-32020`	`HeaderMismatch`	The HTTP headers do not match the corresponding values in the request body, or required headers are missing/malformed.

This error code is allocated from the sub-range the MCP specification reserves for protocol-defined errors. See Error Codes.

Example error response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "error": {
    "code": -32020,
    "message": "Header mismatch: Mcp-Name header value 'foo' does not match body value
'bar'"
  }
}
```

Validation failure conditions include:

- A required standard header (`MCP-Protocol-Version` , `Mcp-Method` , `Mcp-Name`) is missing.
- A header value does not match the corresponding request body value. For headers that permit the Base64 sentinel encoding (`Mcp-Name` and `Mcp-Param-{Name}`), servers **MUST** decode encoded values (see Value Encoding) before comparing them to the body value.
- A header value contains invalid characters.

> **NOTE**
>
> Intermediaries **MUST** return an appropriate HTTP error status (e.g., `400 Bad Request`) for validation failures but are not required to return a JSON-RPC error response.

> **NOTE**
>
> Intermediaries that enforce policy based on mirrored headers (e.g., routing or rate-limiting by tenant) **SHOULD** verify that the `MCP-Protocol-Version` header indicates a version that requires header–body validation. If the version is older or the header is absent, the intermediary **SHOULD** reject the request rather than trusting unvalidated header values.

Backward Compatibility

A client that supports both modern (per-request-metadata) MCP versions and a legacy version that requires an `initialize` handshake **MAY** detect which era the server implements by attempting a modern request first. On `400 Bad Request` , the client **SHOULD** inspect the response body before falling back: modern servers also use `400` for `UnsupportedProtocolVersionError` , `MissingRequiredClientCapabilityError` , and header-validation failures.

- If the body contains a recognized modern JSON-RPC error, the server speaks a modern version of MCP — retry using the advertised `supported` versions or correct the request, rather than falling back.
- If the body is empty or is not a recognized modern JSON-RPC error, fall back to `initialize` and continue with the legacy version for subsequent requests.

See Versioning: Backward Compatibility for the era model and a compatibility matrix for implementors.

Earlier Streamable HTTP Revisions

Protocol versions `2025-03-26` through `2025-11-25` also used the Streamable HTTP transport, but in a different shape: servers could assign a session via the `Mcp-Session-Id` header (terminated with HTTP DELETE), clients could open a standalone SSE stream with HTTP GET to receive server-initiated messages, servers could send JSON-RPC *requests* on SSE streams, and streams were resumable via `Last-Event-ID` . None of these mechanisms are part of this revision.

A server that supports only this revision and receives such traffic from an older client **SHOULD** respond as follows:

- HTTP GET or DELETE to the MCP endpoint: respond with `405 Method Not Allowed`.
- An `Mcp-Session-Id` header on a request: ignore it, and do not mint or echo session IDs.
- A `Last-Event-ID` header: ignore it; streams are not resumable.

Servers and clients that need to interoperate with counterparts speaking those protocol versions implement the behavior described in the corresponding revision (for example, 2025-11-25: Streamable HTTP), in addition to the version-negotiation fallback described above.

HTTP+SSE Transport (2024-11-05)

> **WARNING**
>
> **Deprecated**: The HTTP+SSE transport from protocol version 2024-11-05 has been deprecated since protocol version `2025-03-26` and is classified as Deprecated under the feature lifecycle policy (SEP-2596). New implementations **SHOULD NOT** adopt it; existing implementations **SHOULD** migrate to Streamable HTTP. It is eligible for removal in a future revision; see the deprecated features registry.

Clients and servers can maintain backward 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 a request 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 HTTP status code `400 Bad Request`, `404 Not Found`, or `405 Method Not Allowed` **and** the response body is not a recognized modern JSON-RPC error (a modern server returns one for unsupported version, unknown method, or header-validation failure):
     - 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.

5.5 Authorization

5.5.1 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 Bearer Token Usage (RFC6750)
- OAuth 2.0 Authorization Server Metadata (RFC8414)
- OAuth 2.0 Dynamic Client Registration Protocol (RFC7591)
- Resource Indicators for OAuth 2.0 (RFC8707)
- OAuth 2.0 Protected Resource Metadata (RFC9728)
- OAuth 2.0 Authorization Server Issuer Identification (RFC9207)
- OAuth Client ID Metadata Documents (draft-ietf-oauth-client-id-metadata-document-00)
- OpenID Connect Discovery 1.0
- OpenID Connect Dynamic Client Registration 1.0 (OpenID Connect Registration)

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. Authorization Server Discovery 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). Note that Dynamic Client Registration is deprecated and retained for backwards compatibility with authorization servers that do not support Client ID Metadata Documents.

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

MCP servers advertise their associated authorization servers through OAuth 2.0 Protected Resource Metadata, and MCP clients determine authorization server endpoints and supported capabilities through authorization server metadata discovery. Implementations **MUST** follow the normative discovery requirements defined in Authorization Server Discovery.

Client Registration

Before initiating the authorization flow, MCP clients **MUST** obtain a client ID through one of three registration mechanisms: Client ID Metadata Documents, pre-registration, or Dynamic Client Registration, following the requirements and selection priority defined in Client Registration.

Scope Selection Strategy

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 the current operation. These scopes are required to satisfy the current request. When re-authorizing, clients **SHOULD** include these scopes alongside any previously granted scopes to avoid losing permissions needed for other operations (see Step-Up Authorization Flow). 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"
```

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:

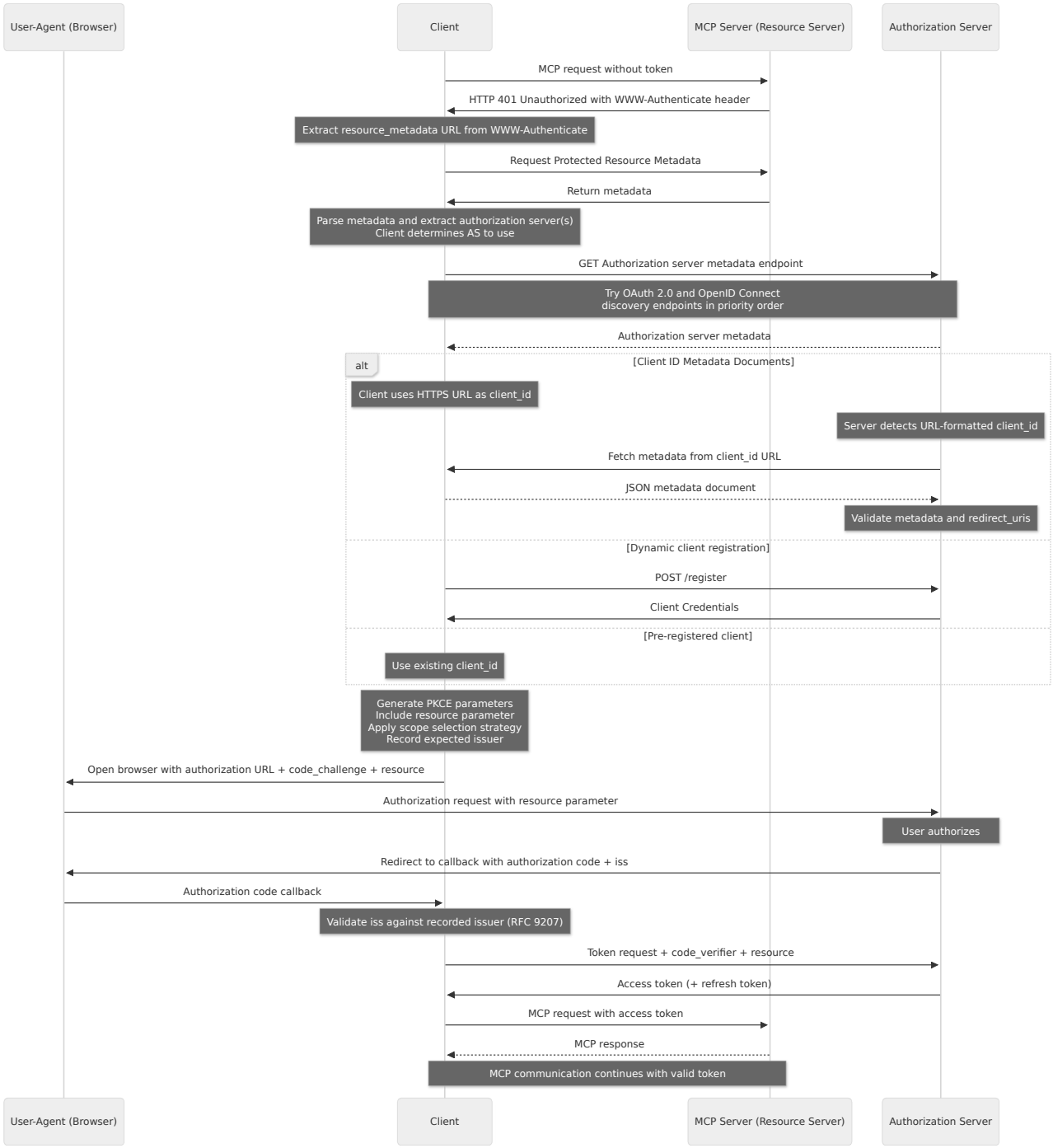
1. **Use `scope` parameter** from the initial `WWW-Authenticate` header in the 401 response, if provided
2. **If `scope` is not available**, use all scopes defined in `scopes_supported` from the Protected Resource Metadata document, omitting the `scope` parameter if `scopes_supported` is undefined.

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 registration step shown in the flow uses one of the mechanisms defined in Client Registration.

The complete Authorization flow proceeds as follows:



Authorization Response Validation

Before redirecting the user-agent, the client **MUST** record the `issuer` value from the selected authorization server's validated metadata document (see Authorization Server Metadata Discovery) and associate it with the same per-request record used to store the PKCE code verifier (and the `state` value, if used). The validation in this section depends on that recorded value being authentic; it provides no protection if the expected issuer was obtained from an unvalidated source.

MCP authorization servers **SHOULD** include the `iss` parameter in authorization responses, including error responses, as defined in RFC9207 Section 2. Authorization servers that include the `iss` parameter **MUST** advertise this by setting `authorization_response_iss_parameter_supported` to `true` in their metadata (RFC9207 Section 2.3).

On receiving the authorization response, MCP clients **MUST** apply the validation in RFC9207 Section 2.4 before transmitting the authorization code to any token endpoint:

`authorization_response_iss_parameter_supported`	`iss` in response	Client action
`true`	present	Compare to the recorded issuer using simple string comparison (RFC3986 Section 6.2.1)
`true`	absent	Reject the response
`false` or absent	present	Compare to the recorded issuer using simple string comparison (RFC3986 Section 6.2.1)
`false` or absent	absent	Proceed

The third row applies the local-policy provision in RFC9207 Section 2.4: this specification compares a present `iss` against the recorded issuer regardless of metadata advertisement, to accommodate authorization servers that emit `iss` before updating their metadata.

A future revision of this specification is expected to upgrade authorization server inclusion of `iss` from **SHOULD** to **MUST**. Implementers are encouraged to emit and validate `iss` now to ease that transition; client rejection behavior on `iss` absence will continue to be keyed on `authorization_response_iss_parameter_supported` until that revision defines the upgrade path.

After decoding the `iss` value from the `application/x-www-form-urlencoded` response per RFC 9207 Section 2.4, clients **MUST NOT** apply scheme or host case folding, default-port elision, trailing-slash, or percent-encoding normalization (RFC 3986 Sections 6.2.2-6.2.3) before comparison.

This validation applies equally to error responses - on mismatch the client **MUST NOT** act on or display `error`, `error_description`, or `error_uri`.

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.

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.

Refresh Tokens

This section provides guidance for MCP Clients and MCP Servers when handling or issuing refresh tokens for both OAuth and OpenID Connect.

MCP Clients that desire refresh tokens:

- **MUST** keep refresh tokens confidential in transit and storage as specified in OAuth 2.1 Section 4.3
- **SHOULD** include `refresh_token` in their `grant_types` client metadata
- **MAY** add `offline_access` to the `scope` parameter of the authorization and token requests when the Authorization Server metadata contains it in `scopes_supported`
- **MUST NOT** assume refresh tokens will be issued; the AS retains discretion

MCP Servers (Protected Resources) **SHOULD NOT** include `offline_access` in `WWW-Authenticate` scope or Protected Resource Metadata `scopes_supported`, as refresh tokens are not a resource requirement.

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 operation in the `scope` parameter, consistent with RFC 6750 Section 3.1. The `scope` attribute describes the scopes necessary to access the requested resource — servers are not required to include the client's previously granted scopes.

Whatever scope-inclusion strategy a server adopts, servers **SHOULD** include all scopes required for the current operation in a single challenge. Challenging incrementally (returning one missing scope, then another on the subsequent retry) forces multiple authorization round-trips for a single operation and degrades user experience. The required scopes may be determined dynamically based on the specific request arguments and context, but once determined, they should be emitted together.

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.

Scope accumulation across operations is a client-side responsibility. See the Step-Up Authorization Flow for the scope-union requirement.

Example insufficient scope response:

```
HTTP/1.1 403 Forbidden
WWW-Authenticate: Bearer error="insufficient_scope",
                        scope="files:write",
                        resource_metadata="https://mcp.example.com/.well-known/oauth-
protected-resource",
                        error_description="File write permission required for this
operation"
```

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** by computing the union of the client's previously requested scope set and the scopes from the current challenge. This ensures previously granted permissions are preserved when servers emit per-operation scope challenges per RFC 6750 Section 3.1. Clients **MAY** also consult the Scope Selection Strategy for initial scope selection guidance.
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.

Servers **MUST** account for scope hierarchies, where a broader scope implies narrower ones, when deciding whether a token is sufficient for an operation.

Security Considerations

Implementations of this specification **MUST** follow the normative security requirements in Security Considerations, covering token audience binding and validation, token theft, communication security, authorization code protection, mix-up and confused deputy attacks, open redirection, and Client ID Metadata Document security.

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.

5.5.2 Authorization Server Discovery

This document 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".

When multiple authorization servers are listed in `authorization_servers`, each is an independent OAuth 2.0 authorization server. Consistent with RFC 6749 Section 2.2, client identifiers are unique to the authorization server that issued them. Clients **MUST** maintain separate registration state (client credentials, tokens) per authorization server and **MUST NOT** assume that credentials valid for one authorization server will be accepted by another. See Authorization Server Binding for the requirements on associating client credentials with the authorization server that issued them.

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

Servers can also include a `scope` parameter in the `WWW-Authenticate` challenge to indicate the scopes required for accessing the resource; the scope semantics and the associated client behavior are defined in the Scope Selection Strategy section.

Authorization Server Metadata Discovery

MCP uses the default `oauth-authorization-server` well-known URI suffix defined in RFC 8414 Section 3.1 for authorization server metadata discovery. MCP does not define an application-specific well-known URI suffix.

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 RFC 8414 Section 3.1 "Authorization Server Metadata Request" for OAuth 2.0 Authorization Server Metadata discovery and RFC 8414 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`

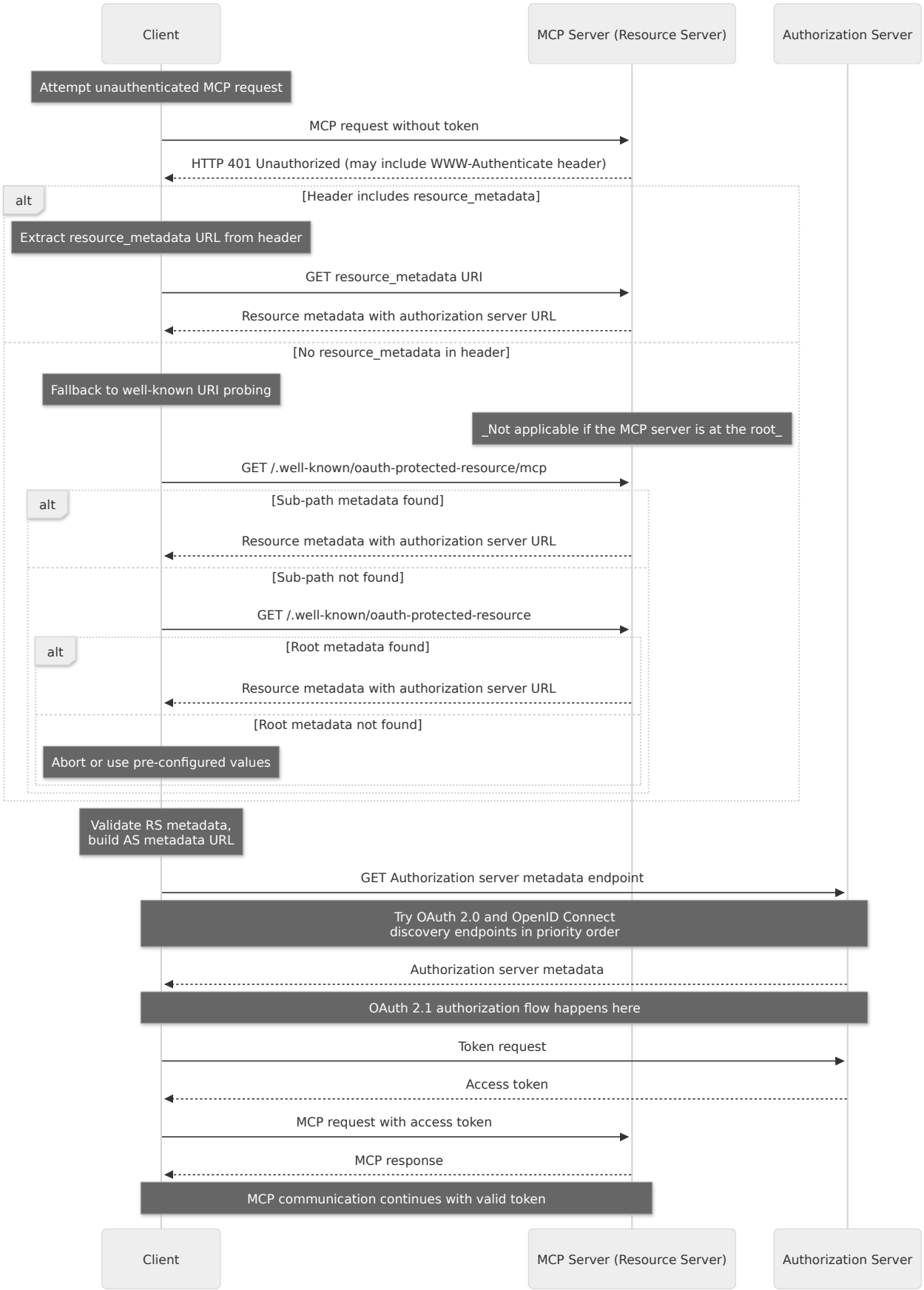
For issuer URLs without path components (e.g., `https://auth.example.com`), clients **MUST** try:

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`

After retrieving a metadata document, MCP clients **MUST** validate it as required by RFC8414 Section 3.3 or OpenID Connect Discovery Section 4.3: the `issuer` value in the document **MUST** be identical to the issuer identifier used to construct the well-known URL. If they differ, the client **MUST NOT** use the metadata. For example, a document fetched from `https://attacker.example/.well-known/oauth-authorization-server` that contains `"issuer": "https://honest.example"` **MUST** be rejected.

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

5.5.3 Client Registration

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** use 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 that it supports them (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 for client registration.

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:

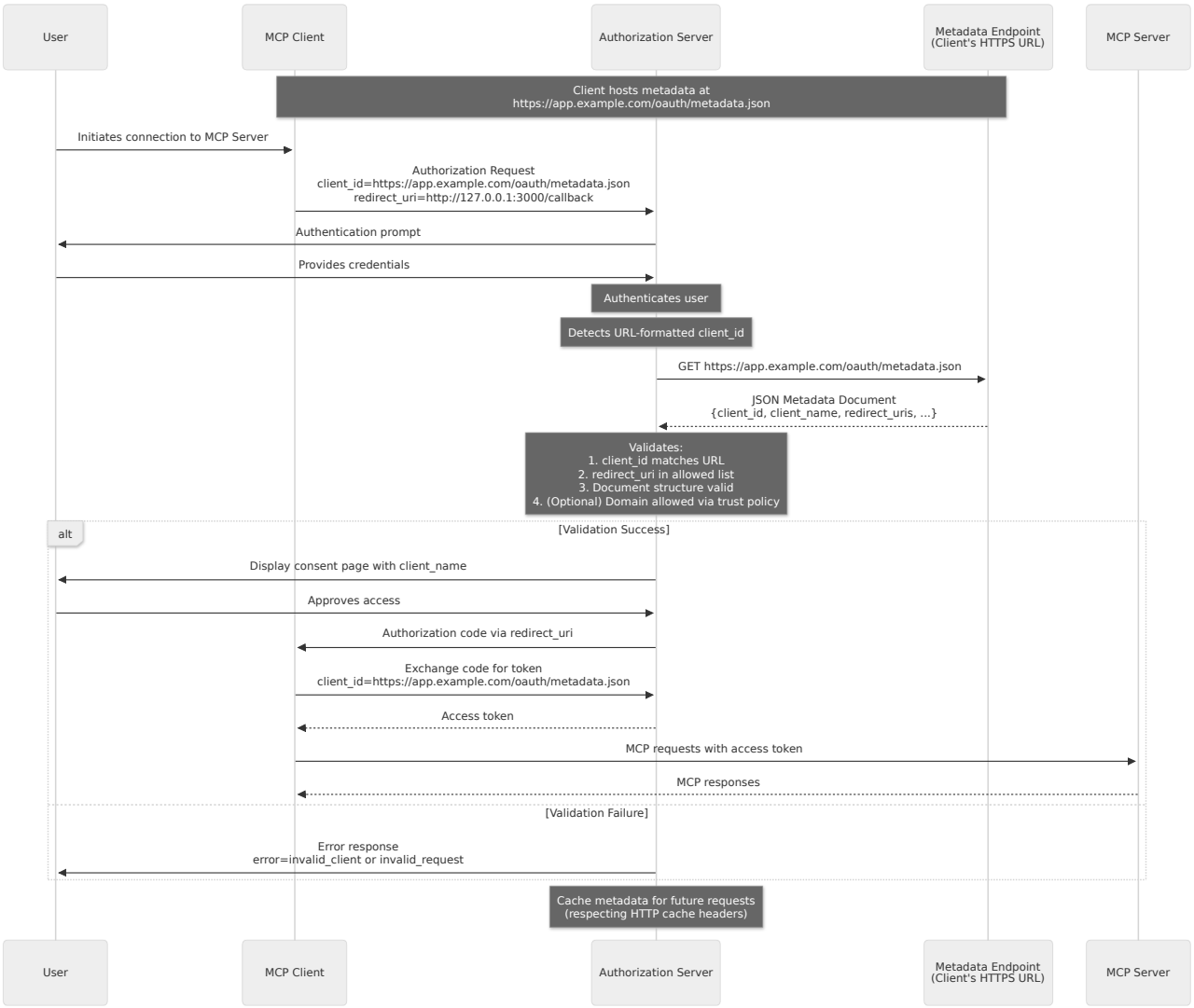
- **SHOULD** fetch metadata documents when encountering URL-formatted client_ids
- **MUST** validate that the fetched document's `client_id` matches the URL exactly
- **SHOULD** cache metadata respecting HTTP cache headers
- **MUST** validate redirect URIs presented in an authorization request against 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 and in Client ID Metadata Document Security

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://[::1]: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:



Advertising CIMD Support

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.

Pre-registration

MCP clients **SHOULD** support an option for static client credentials such as those supplied by a pre-registration 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

> **WARNING**
>
> Dynamic Client Registration is deprecated. New implementations should use Client ID Metadata Documents instead. This option remains available for backwards compatibility with authorization servers that do not support Client ID Metadata Documents.

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.

Application Type and Redirect URI Constraints

When authorization servers support OpenID Connect (OIDC) and Dynamic Client Registration, they may enforce additional constraints on redirect URIs based on the `application_type` parameter as defined in OpenID Connect Dynamic Client Registration 1.0.

MCP clients **MUST** specify an appropriate `application_type` during Dynamic Client Registration. Omitting it defaults to `"web"` under OIDC, which can conflict with native-style redirect URIs; non-OIDC servers safely ignore the parameter.

- **Native applications** (desktop applications, mobile apps, CLI tools, and locally-hosted web applications accessed via `localhost`) **SHOULD** use `application_type: "native"`
- **Web applications** (remote browser-based applications served from a non-local host) **SHOULD** use `application_type: "web"`

MCP clients **MUST** be prepared to handle registration failures due to redirect URI constraints when authorization servers implement OIDC. When a registration request is rejected, clients **SHOULD** surface a meaningful error to the user or developer. Clients **MAY** retry registration with an adjusted `application_type` or with redirect URIs that conform to the authorization server's requirements for the given application type.

Authorization Server Binding

Clients that use pre-registered credentials, or persist client credentials obtained via Dynamic Client Registration, **MUST** associate those credentials with the specific authorization server that issued them, keyed by the authorization server's `issuer` identifier. When the authorization server changes (detected via updated protected resource metadata), clients **MUST NOT** reuse client credentials from a different authorization server and **MUST** re-register with the new authorization server.

Pre-registered credentials are inherently specific to a particular authorization server. If the authorization server indicated by protected resource metadata no longer matches the one the credentials were registered with, clients **SHOULD** surface an error rather than silently attempting to use mismatched credentials.

Client IDs based on Client ID Metadata Documents are portable across authorization servers, since they are self-hosted HTTPS URLs resolved by the authorization server on demand. No re-registration is needed when the authorization server changes.

5.5.4 Authorization Security Considerations

This document outlines security requirements that implementers **MUST** consider when building MCP clients and servers.

Additionally, implementors **MUST** follow OAuth 2.1 security best practices as outlined 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.

Mix-Up Attacks

An attacker that controls one of the authorization servers an MCP client interacts with may attempt to have the client send it an authorization code or token issued by a different, honest authorization server (a mix-up attack, described in RFC9207 Section 1). Authorization Response Validation specifies the required mitigation.

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

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.

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 for accepting Client ID Metadata Documents, as described in Section 6.4 and Section 6.8 of the Client ID Metadata Document specification.

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.

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.

6 Client Features

6.1 Roots

> **WARNING**
>
> **Deprecated**: The Roots feature is deprecated as of protocol version `2026-07-28` (SEP-2577). Under the feature lifecycle policy, it remains in the specification for at least twelve months after this revision's release before it becomes eligible for removal. New implementations **SHOULD NOT** adopt it; existing implementations **SHOULD** migrate to passing directories or files via tool parameters, resource URIs, or server configuration. See the deprecated features registry.

The Model Context Protocol (MCP) provides a standardized way for clients to expose filesystem "roots" to servers. Roots inform servers about the directories and files the client considers relevant, so that servers can focus their operations accordingly. They are informational guidance rather than an access-control mechanism. The protocol does not enforce that servers stay within roots. Servers can request the list of roots from supporting clients.

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 in `_meta.io.modelcontextprotocol/clientCapabilities` on each request:

```
{
  "_meta": {
    "io.modelcontextprotocol/clientCapabilities": {
      "roots": {}
    }
  }
}
```

Protocol Messages

Listing Roots

To retrieve roots during the processing of a client request, servers send an `InputRequiredResult` containing a `roots/list` request:

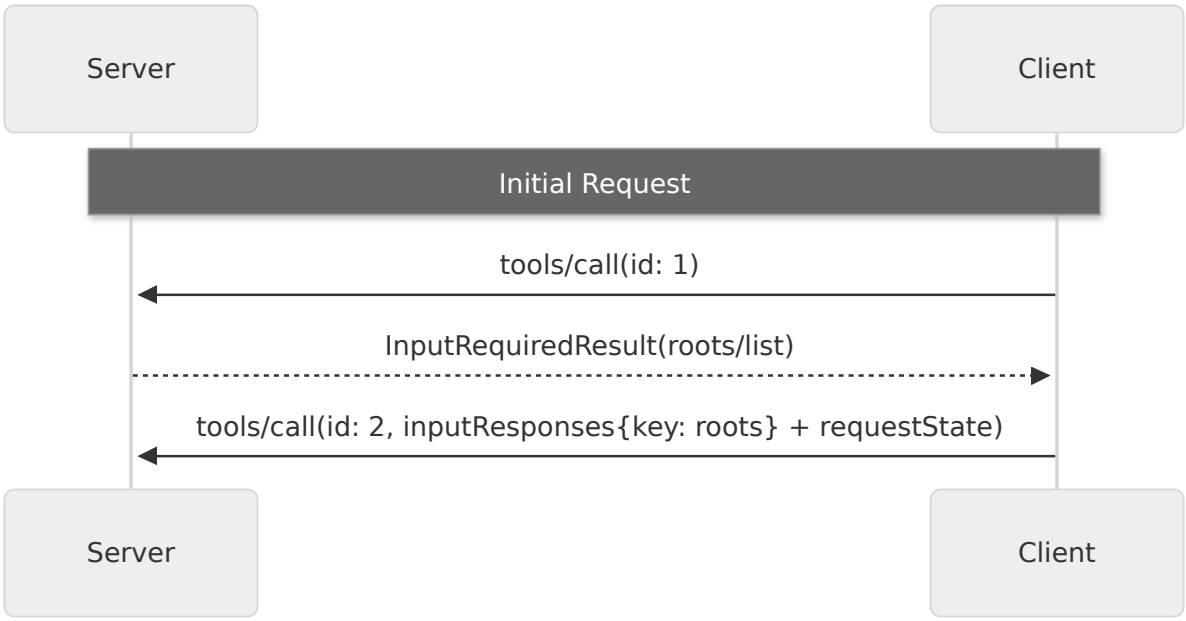
Input request (delivered inside `InputRequiredResult.inputRequests`):

```
{
  "method": "roots/list"
}
```

Client result (returned inside `inputResponses` on the retried request):

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

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

If an error occurs, the client does not need to replay the initial call with an error message as the server is not waiting for a response with the `InputRequiredResult` pattern.

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
    - Respect root boundaries in operations
    - Cache root information appropriately

6.2 Sampling

> **WARNING**
>
> **Deprecated**: The Sampling feature is deprecated as of protocol version `2026-07-28` (SEP-2577). Under the feature lifecycle policy, it remains in the specification for at least twelve months after this revision's release before it becomes eligible for removal. New implementations **SHOULD NOT** adopt it; existing implementations **SHOULD** migrate to integrating directly with LLM provider APIs. See the deprecated features registry.

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. The tool definitions in the `tools` array are scoped to the sampling request — they don't need to correspond to registered tools. This enables servers to implement agentic behaviors where the LLM can call specially designated 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 in `_meta.io.modelcontextprotocol/clientCapabilities` on each request:

Basic sampling:

```
{
  "_meta": {
    "io.modelcontextprotocol/clientCapabilities": {
      "sampling": {}
    }
  }
}
```

With tool use support:

```
{
  "_meta": {
    "io.modelcontextprotocol/clientCapabilities": {
      "sampling": {
        "tools": {}
      }
    }
  }
}
```

With context inclusion support (deprecated):

```
{
  "_meta": {
    "io.modelcontextprotocol/clientCapabilities": {
      "sampling": {
        "context": {}
      }
    }
  }
}
```

> **NOTE**
>
> The `includeContext` parameter values `"thisServer"` and `"allServers"` are deprecated under the feature lifecycle policy (SEP-2596); they will be removed no later than the Sampling feature itself. 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. See the deprecated features registry.

Protocol Messages

Creating Messages

To request a language model generation during the processing of a client request, servers send an `InputRequiredResult` containing a `sampling/createMessage` request:

Input request (delivered inside `InputRequiredResult.inputRequests`):

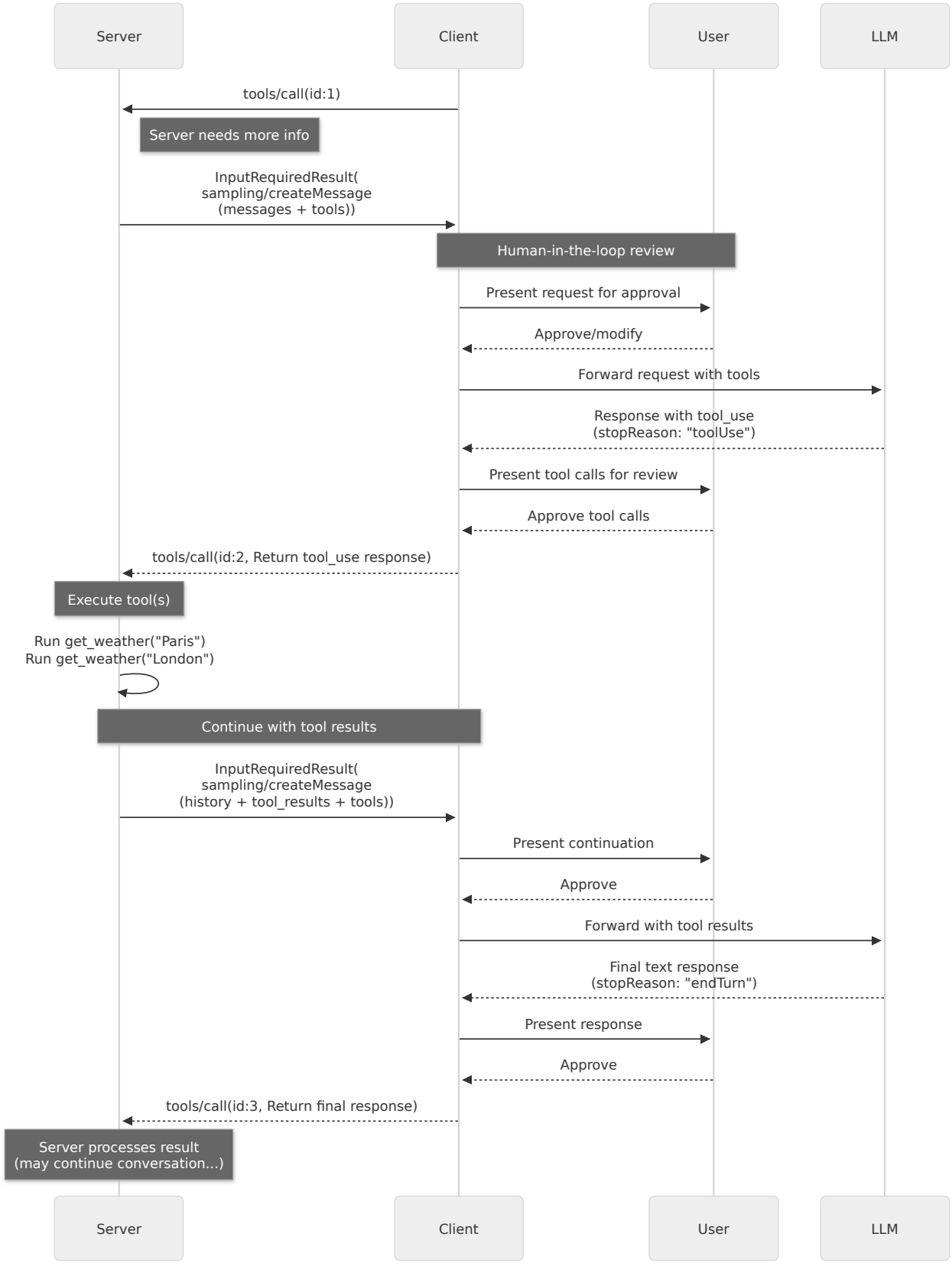
```
{
  "method": "sampling/createMessage",
  "params": {
    "messages": [
      {
        "role": "user",
        "content": {
          "type": "text",
          "text": "What is the capital of France?"
        }
      }
    ],
    "modelPreferences": {
      "hints": [
        {
          "name": "claude-3-sonnet"
        }
      ],
      "costPriority": 0.3,
      "intelligencePriority": 0.8,
      "speedPriority": 0.5
    },
    "temperature": 0.1,
    "systemPrompt": "You are a helpful assistant.",
    "includeContext": "thisServer",
    "maxTokens": 100
  }
}
```

Client result (returned inside `inputResponses` on the retried request):

```
{
  "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:

Input request (Server -> Client, delivered inside `InputRequiredResult.inputRequests`):

```
{
  "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
  }
}
```

Client result (Client -> Server, returned inside `inputResponses` on the retried request):

```
{
  "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 input request (Server -> Client, delivered inside `InputRequiredResult.inputRequests`) with tool results:

```
{
  "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 client result (Client -> Server, returned inside `inputResponses` on the retried request):

```
{
  "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

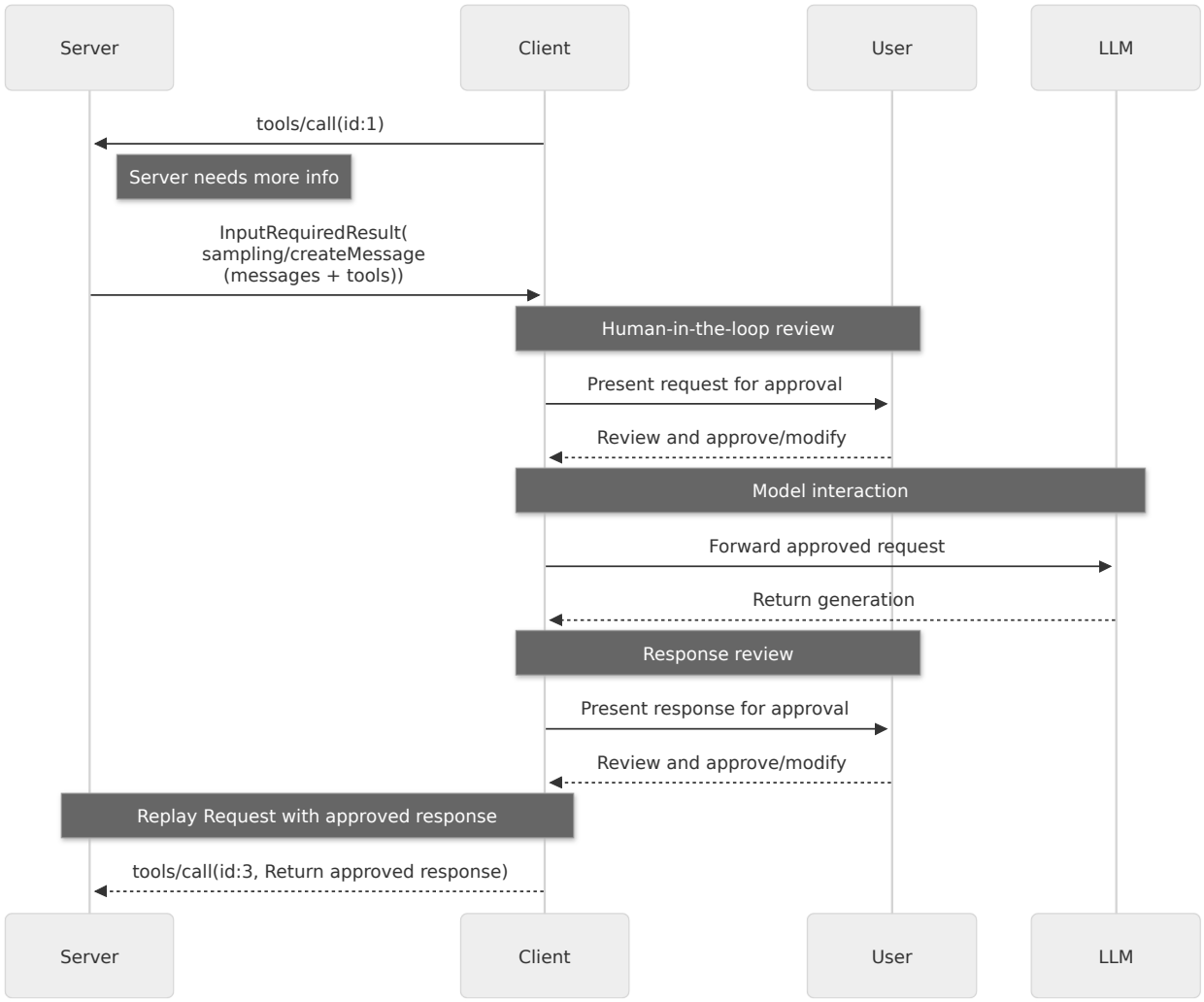
Parallel Tool Use

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 **MUST** contain a `role` field of `"user"` or `"assistant"`; and a `content` field representing the message data.

The list of messages in a sampling request **SHOULD NOT** be retained between separate requests.

The `content` field 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.

System Prompt

The optional `systemPrompt` field allows servers to request a specific system prompt. The client **MAY** modify or ignore this field without communicating this to the server.

Context Inclusion

The `includeContext` parameter specifies what context information the client is expected to include in its response:

- `"none"`: No additional context.
- `"thisServer"`: Include context from the requesting server.
- `"allServers"`: Include context from all connected MCP servers.

The `"thisServer"` and `"allServers"` values are deprecated; see Capabilities.

The client **MAY** modify or ignore this field without communicating this to the server. For example, a client could determine that respecting this field in a particular request would require sharing sensitive information with a server, and constrain its response accordingly.

Sampling Parameters

LLM sampling can be fine-tuned with the following parameters:

- `temperature`: Controls randomness in model responses. Higher values produce higher randomness, and lower values produce more stable output. Valid range depends upon the model provider.
- `maxTokens`: Maximum tokens to generate; required.
- `stopSequences`: Array of sequences that stop generation.
- `metadata`: Additional provider-specific parameters.

The client **MUST** respect the `maxTokens` parameter.

The client **MAY** modify or ignore `temperature`, `stopSequences` and `metadata`. For example, a client could use a model that does not support one or more of these parameters, and would therefore be unable to leverage them.

Result Fields

Sampling results will contain the following fields:

- `role`: The message role; see Messages.
- `content`: The message content. This can be either:
  - A single content block when the response contains only one content block, such as a single text response.
  - An array of content blocks when the response contains one or more content blocks, such as multiple tool uses or mixed content.

  See Messages for content block types.
- `model`: The name of the model that generated the message.
- `stopReason`: The reason why sampling stopped, if known. The specification defines the following (non-exhaustive) stop reasons, although implementations **MAY** provide their own arbitrary values:
  - `"endTurn"`: The participant is yielding the conversation to the other party.
  - `"stopSequence"`: Message generation encountered one of the requested `stopSequences`.
  - `"maxTokens"`: The token limit was reached.
  - `"toolUse"`: The model wants to use one or more tools.

Error Handling

If an error occurs or the user declines the sampling request, the client does not need to replay the initial call with an error message, as the server is not waiting for a response with the `InputRequiredResult` pattern.

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

6.3 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 in `_meta.io.modelcontextprotocol/clientCapabilities` on each request:

```
{
  "_meta": {
    "io.modelcontextprotocol/clientCapabilities": {
      "elicitation": {
        "form": {},
        "url": {}
      }
    }
  }
}
```

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

```
{
  "_meta": {
    "io.modelcontextprotocol/clientCapabilities": {
      "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

Servers **MAY** request information from a user during the processing of a client request, by sending an `InputRequiredResult` containing 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,
     "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

Input request (delivered inside `InputRequiredResult.inputRequests`):

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

Client result (returned inside `inputResponses` on the retried request):

```
{
  "action": "accept",
  "content": {
    "name": "octocat"
  }
}
```

Example: Structured Data Request

Input request (delivered inside `InputRequiredResult.inputRequests`):

```
{
  "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"]
    }
  }
}
```

Client result (returned inside `inputResponses` on the retried request):

```
{
  "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.

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.

Input request (delivered inside `InputRequiredResult.inputRequests`):

```
{
  "method": "elicitation/create",
  "params": {
    "mode": "url",
    "url": "https://mcp.example.com/ui/set_api_key",
    "message": "Please provide your API key to continue."
  }
}
```

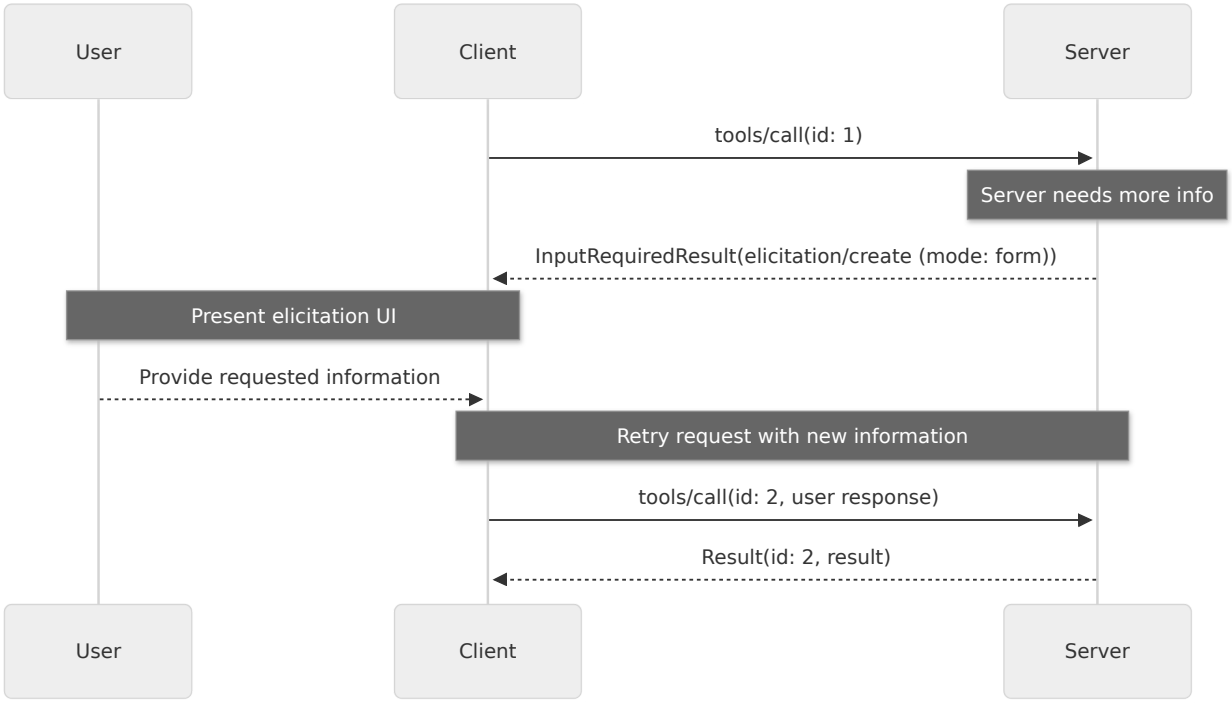
Client result (returned inside `inputResponses` on the retried request):

```
{
  "action": "accept"
}
```

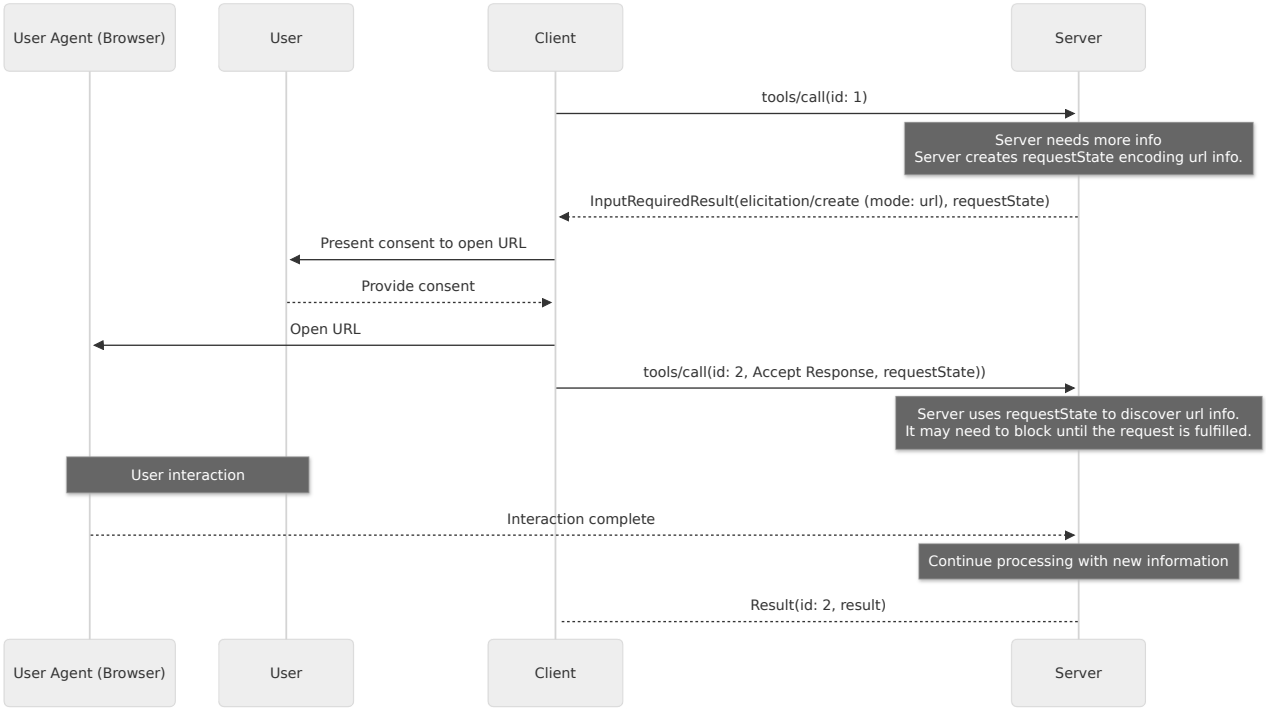
The response with `action: "accept"` indicates that the user has consented to the interaction. It does not mean that the interaction is complete. The interaction occurs out of band and the client is not directly informed of the outcome. When the client retries the original request, the server determines from the echoed `requestState` (or its own stored state) whether the out-of-band interaction has completed, and either returns the final result or responds with another `InputRequiredResult`. Clients **SHOULD** provide manual controls that let the user retry or cancel the original request (or otherwise resume interacting with the client).

Message Flow

Form Mode Flow



URL Mode 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.

```
{
  "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

Elicitations do not require that the server maintain state about users with the multi round-trip requests mechanism.

However, if state is stored, servers implementing elicitation **MUST** securely associate this state with individual users following the guidelines in the security best practices document. Specifically:

- 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.

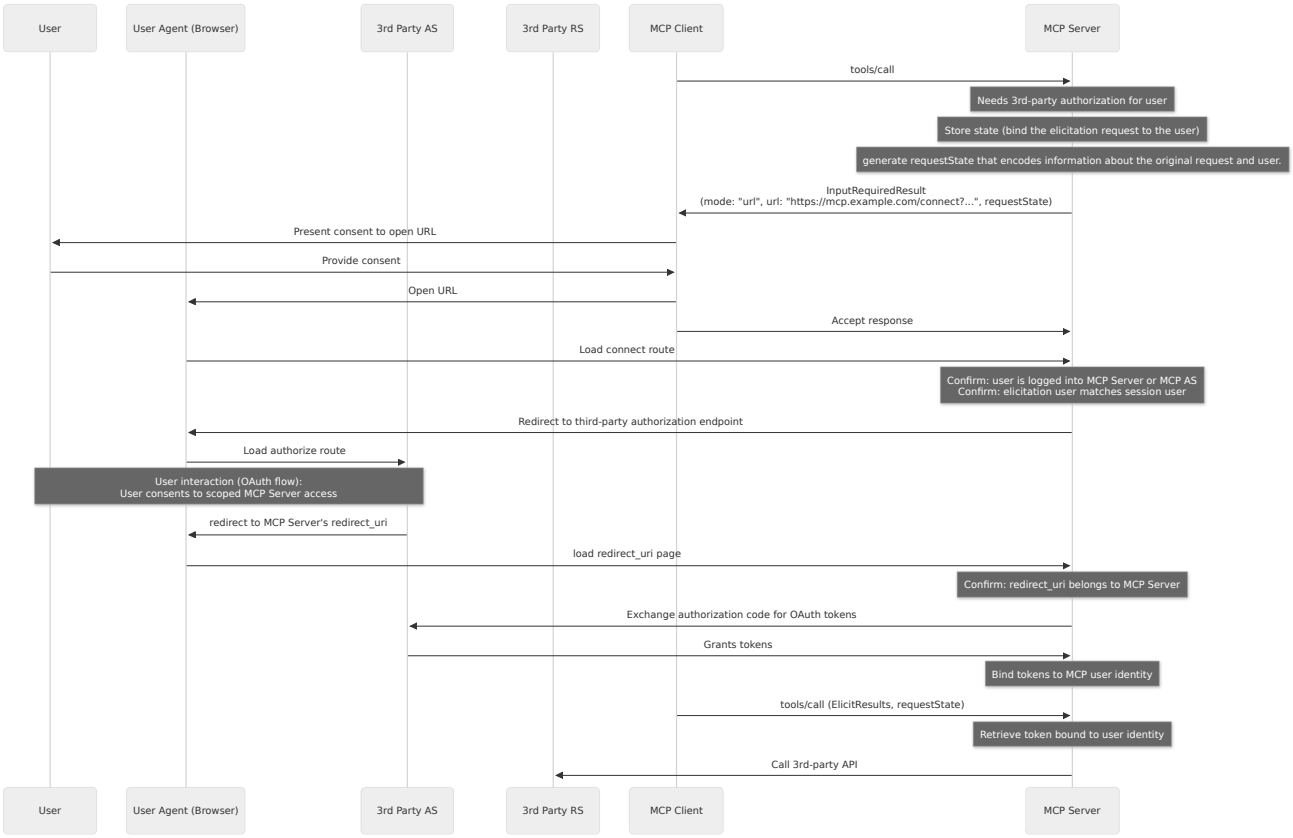
> **NOTE**
>
> For additional background, refer to the token passthrough section of the Security Best Practices document to understand why MCP servers cannot act as pass-through proxies.

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 and an optional `requestState` that encodes information about the elicitation request and user (if needed).
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 **SHOULD NOT** assume that elicitation requests will always succeed, and **MUST** handle cases where the user declines or cancels the elicitation, or where the client fails to process the request.

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** 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, personally 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 from 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?...` rather than the third-party authorization endpoint. This "connect URL" must ensure the user who opened the page is the same user for whom the elicitation was generated. 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 that the user who opens the elicitation URL is the same user for whom the elicitation was generated.

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.

7 Server Features

7.1 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

7.2 Discovery

`server/discover` lets a client query a server's supported protocol versions, capabilities, and identity before sending any other requests. Servers **MUST** implement it.

Request

The request carries no body parameters beyond the standard `_meta`:

```
{
  "jsonrpc": "2.0",
  "id": "discover-1",
  "method": "server/discover",
  "params": {
    "_meta": {
      "io.modelcontextprotocol/protocolVersion": "2026-07-28",
      "io.modelcontextprotocol/clientInfo": {
        "name": "ExampleClient",
        "version": "1.0.0"
      },
      "io.modelcontextprotocol/clientCapabilities": {}
    }
  }
}
```

Response

The server replies with its supported protocol versions, capabilities, and identity. This operation supports caching.

```
{
  "jsonrpc": "2.0",
  "id": "discover-1",
  "result": {
    "resultType": "complete",
    "supportedVersions": ["2026-07-28"],
    "capabilities": {
      "tools": {},
      "resources": {}
    },
    "_meta": {
      "io.modelcontextprotocol/serverInfo": {
        "name": "ExampleServer",
        "version": "1.0.0"
      }
    },
    "instructions": "This server provides weather and resource utilities.",
    "ttlMs": 3600000,
    "cacheScope": "public"
  }
}
```

When to Call

Calling `server/discover` is optional for clients — a client may invoke any RPC inline and handle `UnsupportedProtocolVersionError` if the server does not support the requested version. However, `server/discover` is useful in two scenarios:

- **Presenting server information.** While a client doesn't need to call `server/discover` to use the server, it's a convenient way to retrieve the server's identity, capabilities, and supported versions in a single request. For example, a client can present the capabilities a server supports from a single `server/discover` response instead of probing with separate `tools/list`, `prompts/list`, and `resources/list` requests.
- **stdio backward-compatibility probe.** On stdio, there is no per-request HTTP status code to drive fallback. A client that supports both modern (per-request `_meta`) and legacy (`initialize` handshake) servers **SHOULD** send `server/discover` first; see stdio: Backward Compatibility for the fallback rules.

See Protocol Version Negotiation for the full version-selection flow. For HTTP-specific status codes returned for unknown methods, see the Protocol Version Header section in Transports.

Data Types

DiscoverResult

A discovery result includes:

- `supportedVersions`: Protocol versions the server supports. The client should choose one of these for subsequent requests.
- `capabilities`: Capabilities the server supports (tools, resources, prompts, etc.)
- `_meta['io.modelcontextprotocol/serverInfo']`: Name and version of the server software. Servers **SHOULD** include this field.
- `instructions`: Optional natural-language guidance for LLMs on how to use this server effectively

> **NOTE**
>
> `serverInfo` is self-reported by the server and is not verified by the protocol. It is intended for display, logging, and debugging. Clients **SHOULD NOT** use it to change their behavior, and **SHOULD NOT** rely on it for security decisions.

7.3 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.

> **NOTE**
>
> For brevity, the request examples on this page omit the `_meta` request metadata (`io.modelcontextprotocol/protocolVersion` , `io.modelcontextprotocol/clientInfo` , and `io.modelcontextprotocol/clientCapabilities`). Every request **MUST** include the required `_meta` fields; see `_meta` .

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. This refers to who decides when the prompt is used, not who authors its content. Prompt content is defined by the server.

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 in their `DiscoverResult` :

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

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

Servers that declare the `prompts` capability **MUST** respond to `prompts/list` requests with the set of prompts currently available to the requesting client. This set **MAY** be empty and **MAY** change over time (see List Changed Notification), but **MUST NOT** vary per-connection or as a side effect of other requests on the connection. The set **MAY** vary by the authorization presented on the request — for example, returning only the prompts the caller's granted scopes permit — since credentials are per-request input, not connection state.

Protocol Messages

Listing Prompts

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

Request:

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

Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "resultType": "complete",
    "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",
    "ttlMs": 600000,
    "cacheScope": "public"
  }
}
```

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

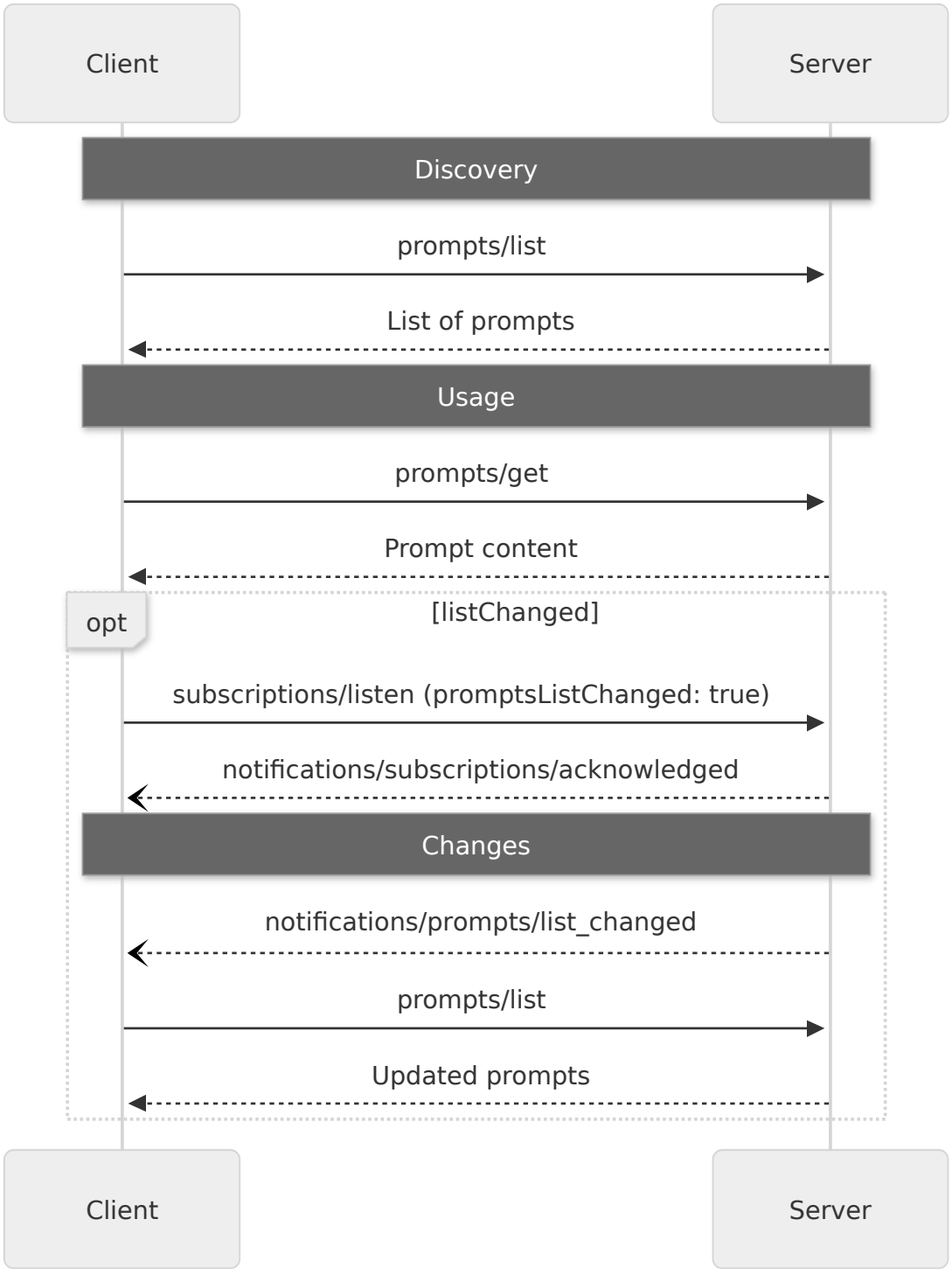
Servers **MAY** also respond to `prompts/get` with an `InputRequiredResult` to indicate that additional input is needed before the prompt can be resolved. This follows the multi round-trip requests mechanism. When retrying the request, clients include `inputResponses` and, if provided by the server, `requestState` in the request parameters.

List Changed Notification

When the list of available prompts changes, servers that declared the `listChanged` capability **SHOULD** send a notification to clients that have opened a `subscriptions/listen` stream with `promptsListChanged: true`:

```
{
  "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.

Resource Links

Prompt messages **MAY** include links to Resources, to provide additional context or data without embedding the resource contents directly. In this case, the prompt message returns a URI that can be 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.

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.

7.4 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.

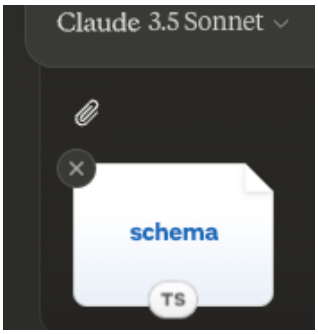
> **NOTE**
>
> For brevity, the request examples on this page omit the `_meta` request metadata (`io.modelcontextprotocol/protocolVersion` , `io.modelcontextprotocol/clientInfo` , and `io.modelcontextprotocol/clientCapabilities`). Every request **MUST** include the required `_meta` fields; see `_meta` .

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

The capability supports two optional features:

- `listChanged` : whether the server will emit notifications when the list of available resources changes.
- `subscribe` : whether the server supports resource-specific update notifications for resources requested through subscriptions/listen using the resourceSubscriptions filter.

Servers may advertise either feature independently, together or neither.

Servers that support neither `listChanged` nor `subscribe` may omit it:

```
{
  "capabilities": {
    "resources": {}
  }
}
```

Servers that declare the `resources` capability **MUST** respond to `resources/list` requests with the set of resources currently available to the requesting client. This set **MAY** be empty and **MAY** change over time (see List Changed Notification), but **MUST NOT** vary per-connection or as a side effect of other requests on the connection. The set **MAY** vary by the authorization presented on the request — for example, returning only the resources the caller's granted scopes permit — since credentials are per-request input, not connection state.

Protocol Messages

Listing Resources

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

Request:

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

Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "resultType": "complete",
    "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",
    "ttlMs": 300000,
    "cacheScope": "public"
  }
}
```

Reading Resources

To retrieve resource contents, clients send a `resources/read` request. This operation supports caching.

Request:

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

Response:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "resultType": "complete",
    "contents": [
      {
        "uri": "file:///project/src/main.rs",
        "mimeType": "text/x-rust",
        "text": "fn main() {\n    println!(\"Hello world!\");\n}"
      }
    ],
    "ttlMs": 60000,
    "cacheScope": "private"
  }
}
```

Servers **MAY** return multiple resource contents in response to a single `resources/read` request. For example, a server could return the contents of several files when a directory resource is read.

Servers **MAY** also respond to `resources/read` with an `InputRequiredResult` to indicate that additional input is needed before the resource can be read. This follows the multi round-trip requests mechanism. When retrying the request, clients include `inputResponses` and, if provided by the server, `requestState` in the request parameters.

Alternatively, if the scheme of `uri` is `https://`, clients may fetch the resource directly from the web. See the Common URI Schemes section for more information.

Resource Templates

Resource templates allow servers to expose parameterized resources using URI templates. Arguments may be auto-completed through the completion API. This operation supports pagination and caching.

Request:

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

Response:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "result": {
    "resultType": "complete",
    "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"]
          }
        ]
      }
    ],
    "nextCursor": "next-page-cursor",
    "ttlMs": 300000,
    "cacheScope": "public"
  }
}
```

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"
}
```

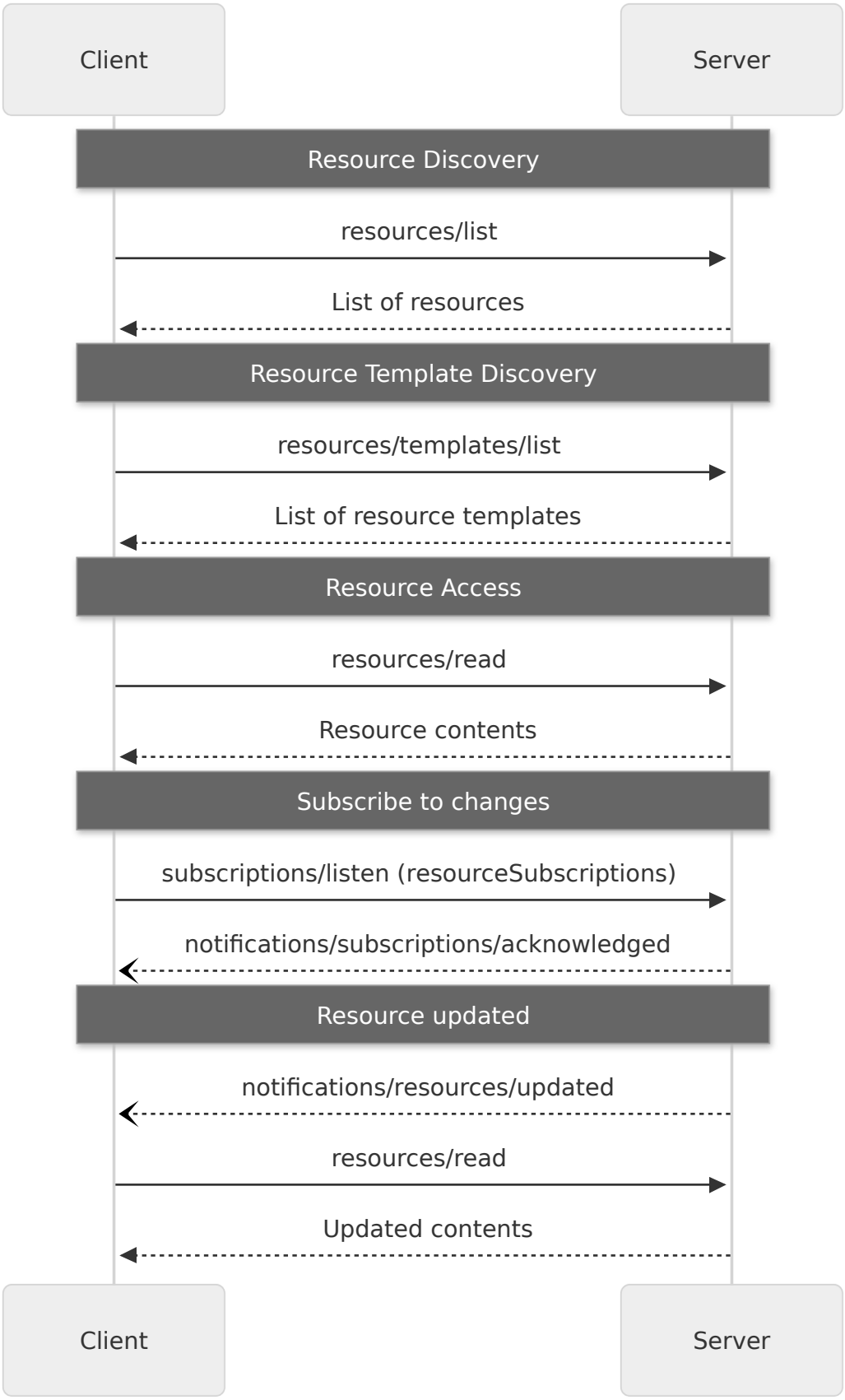
Subscriptions

Clients subscribe to change notifications for specific resources by sending a `subscriptions/listen` request with the resource URIs listed in `notifications.resourceSubscriptions`. The server delivers `notifications/resources/updated` on the resulting stream whenever a watched resource changes.

```
{
  "jsonrpc": "2.0",
  "method": "notifications/resources/updated",
  "params": {
    "_meta": { "io.modelcontextprotocol/subscriptionId": 4 },
    "uri": "file:///project/src/main.rs"
  }
}
```

See Subscriptions for the full protocol mechanics (acknowledgment, `subscriptionId` correlation, and cancellation).

Message Flow



Data Types

Resource

A resource definition includes:

- `uri`: Unique identifier for the resource
- `name`: The name of the resource.
- `title`: Optional human-readable name of the resource for display purposes.
- `description`: Optional description
- `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 is 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

If the requested resource does not exist, servers **MUST** return a JSON-RPC error with code `-32602` (Invalid Params). Servers **SHOULD** return `-32603` for internal errors.

For backwards compatibility, clients **SHOULD** also accept `-32002` as a resource not found error, as earlier protocol versions used this code.

Servers **MUST NOT** return an empty `contents` array for a non-existent resource. An empty array is ambiguous—it could mean the resource exists but has no content, or that it doesn't exist at all.

Example error:

```
{
  "jsonrpc": "2.0",
  "id": 5,
  "error": {
    "code": -32602,
    "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
5. Servers **MUST** sanitize file paths to prevent directory traversal attacks when serving `file://` resources

7.5 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.

> **NOTE**
>
> For brevity, the request examples on this page omit the `_meta` request metadata (`io.modelcontextprotocol/protocolVersion` , `io.modelcontextprotocol/clientInfo` , and `io.modelcontextprotocol/clientCapabilities`). Every request **MUST** include the required `_meta` fields; see `_meta` .

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.

Servers that declare the `tools` capability **MUST** respond to `tools/list` requests with the set of tools currently available to the requesting client. This set **MAY** be empty and **MAY** change over time (see List Changed Notification), but **MUST NOT** vary per-connection or as a side effect of other requests on the connection. The set **MAY** vary by the authorization presented on the request — for example, returning only the tools the caller's granted scopes permit — since credentials are per-request input, not connection state.

Servers **SHOULD** return tools in a deterministic order (i.e., the same ordering across requests when the underlying set of tools has not changed). Deterministic ordering enables clients to reliably cache the tool list and improves LLM prompt cache hit rates when tools are included in model context.

Protocol Messages

Listing Tools

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

Request:

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

Response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "resultType": "complete",
    "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"]
          }
        ]
      }
    ],
    "nextCursor": "next-page-cursor",
    "ttlMs": 300000,
    "cacheScope": "public"
  }
}
```

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": {
    "resultType": "complete",
    "content": [
      {
        "type": "text",
        "text": "Current weather in New York:\nTemperature: 72°F\nConditions: Partly
cloudy"
      }
    ],
    "isError": false
  }
}
```

Input Required Tool Results

Servers **MAY** respond to `tools/call` with an `InputRequiredResult` to indicate that additional input is needed before the tool call can be completed. This follows the multi round-trip requests mechanism.

When retrying the request with input responses, clients include `inputResponses` and, if provided by the server, `requestState` in the request parameters:

Input Required Response:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "resultType": "input_required",
    "inputRequests": {
      "github_login": {
        "method": "elicitation/create",
        "params": {
          "mode": "form",
          "message": "Please provide your GitHub username",
          "requestedSchema": {
            "type": "object",
            "properties": {
              "name": { "type": "string" }
            },
            "required": ["name"]
          }
        }
      }
    },
    "requestState": "eyJsb2NhdGlvbiI6Ik5ldyBZb3JrIn0..."
  }
}
```

Retry with Input Responses:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "tools/call",
  "params": {
    "name": "get_weather",
    "arguments": {
      "location": "New York"
    },
    "inputResponses": {
      "github_login": {
        "action": "accept",
        "content": {
          "name": "octocat"
        }
      }
    },
    "requestState": "eyJsb2NhdGlvbiI6Ik5ldyBZb3JrIn0..."
  }
}
```

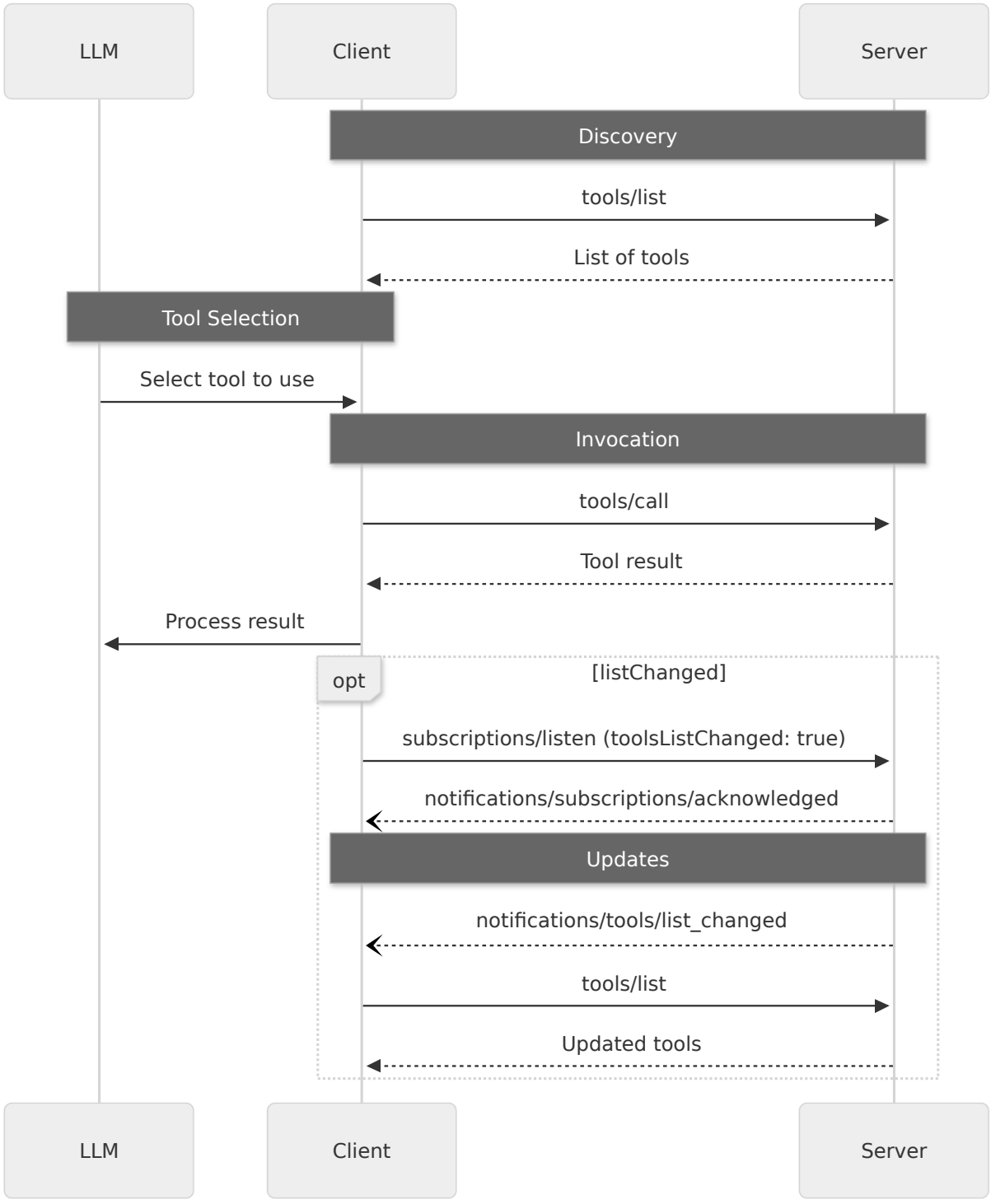
Note that the JSON-RPC `id` **MUST** be different between the initial request and the retry.

List Changed Notification

When the list of available tools changes, servers that declared the `listChanged` capability **SHOULD** send a notification to clients that have opened a `subscriptions/listen` stream with `toolsListChanged: true`:

```
{
  "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 `$schema` field is present
  - **MUST** be a valid JSON Schema object (not `null`)
  - For tools with no parameters, use one of these valid approaches:
    - `{ "type": "object", "additionalProperties": false }` - **Recommended**: explicitly accepts only empty objects
    - `{ "type": "object" }` - accepts any object (including with properties)
  - Properties **MAY** include an `x-mcp-header` annotation to expose parameter values as HTTP headers
- `outputSchema`: Optional JSON Schema defining expected output structure
  - Follows the JSON Schema usage guidelines
  - Defaults to 2020-12 if no `$schema` field is present
- `annotations`: Optional properties describing tool behavior

> **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`

> **NOTE**
>
> Tool name uniqueness is scoped to a single server. Clients or proxies that aggregate tools from multiple servers **MAY** encounter naming collisions (for example, two servers each exposing a `search` tool) and **SHOULD** implement a disambiguation strategy such as prefixing tool names with a server identifier.
>
> The server `name` (from `serverInfo`) is not guaranteed to be unique across servers and **SHOULD NOT** be relied upon for disambiguation.

x-mcp-header

The `x-mcp-header` extension property allows servers to designate specific tool parameters to be mirrored into HTTP headers when using the Streamable HTTP transport. This enables network intermediaries (load balancers, proxies, WAFs) to route and process requests based on parameter values without parsing the request body.

The `x-mcp-header` property is placed directly within the JSON Schema of the property to be mirrored. Its value specifies the name portion of the resulting `Mcp-Param-{name}` HTTP header.

Constraints on `x-mcp-header` values:

- **MUST NOT** be empty
- **MUST** match HTTP field-name token syntax (`1*tchar` , RFC 9110 Section 5.1)
- **MUST NOT** contain control characters, including carriage return (CR, `\r`) or line feed (LF, `\n`)
- **MUST** be case-insensitively unique among all `x-mcp-header` values in the `inputSchema`
- **MUST** only be applied to parameters with primitive types (integer, string, boolean). Parameters with type `number` are not permitted. Integer values **MUST** be within the safe range for integers represented using IEEE754 double-precision floating point numbers ($-2^{53}+1$ to $2^{53}-1$)
- **MUST** only be applied to properties that are *statically reachable* from the schema root, as defined in Custom Headers from Tool Parameters, which also defines how header values are extracted from call arguments

Clients using the Streamable HTTP transport **MUST** reject tool definitions where any `x-mcp-header` value violates these constraints. Rejection means the client **MUST** exclude the invalid tool from the result of `tools/list` . Clients **SHOULD** log a warning when rejecting a tool definition, including the tool name and the reason for rejection. This ensures that a single malformed tool definition does not prevent other valid tools from being used. Clients using other transports (e.g., stdio) **MAY** ignore `x-mcp-header` annotations entirely.

Example tool definition with `x-mcp-header` :

```
{
  "name": "execute_sql",
  "description": "Execute SQL on Google Cloud Spanner",
  "inputSchema": {
    "type": "object",
    "properties": {
      "region": {
        "type": "string",
        "description": "The region to execute the query in",
        "x-mcp-header": "Region"
      },
      "query": {
        "type": "string",
        "description": "The SQL query to execute"
      }
    },
    "required": ["region", "query"]
  }
}
```

In this example, when the tool is called with `"region": "us-west1"` , the client adds the header `Mcp-Param-Region: us-west1` to the HTTP request.

> **WARNING**
>
> Server developers **SHOULD NOT** mark sensitive parameters (passwords, API keys, tokens, PII) with `x-mcp-header` , as header values are visible to network intermediaries.

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 value in the `structuredContent` field of a result. This can be any JSON value (object, array, string, number, boolean, or null) that conforms to the tool's `outputSchema` if one is defined.

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": {
    "resultType": "complete",
    "content": [
      {
        "type": "text",
        "text": "{\"temperature\": 22.5, \"conditions\": \"Partly cloudy\",
\"humidity\": 65}"
      }
    ],
    "structuredContent": {
      "temperature": 22.5,
      "conditions": "Partly cloudy",
      "humidity": 65
    }
  }
}
```

Example tool with array output schema:

```
{
  "name": "list_users",
  "title": "User List",
  "description": "Returns a list of all users",
  "inputSchema": {
    "type": "object",
    "properties": {}
  },
  "outputSchema": {
    "type": "array",
    "items": {
      "type": "object",
      "properties": {
        "id": { "type": "string" },
        "name": { "type": "string" },
        "email": { "type": "string" }
      },
      "required": ["id", "name", "email"]
    }
  }
}
```

Example valid response for a tool with array output:

```
{
  "jsonrpc": "2.0",
  "id": 6,
  "result": {
    "resultType": "complete",
    "content": [
      {
        "type": "text",
        "text": "Found 2 users: Alice (alice@example.com) and Bob (bob@example.com)."
      }
    ],
    "structuredContent": [
      { "id": "1", "name": "Alice", "email": "alice@example.com" },
      { "id": "2", "name": "Bob", "email": "bob@example.com" }
    ]
  }
}
```

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
  }
}
```

Stateful Tools

> **NOTE**
>
> This section is non-normative guidance for tool design. The protocol has no concept of a state handle; from the wire's perspective a handle is an ordinary string in a tool result and an ordinary argument to subsequent tool calls.

MCP has no protocol-level session, so a server cannot rely on implicit per-connection state to relate one tool call to the next. Servers that need to maintain state across calls — a shopping cart, an open browser context, a database transaction — should do so by returning an explicit handle from a creation tool and accepting that handle as an argument on subsequent calls.

For example, a server that manages a shopping cart might expose:

```
// → tools/call
{ "name": "create_basket", "arguments": {} }

// ← result
{
  "content": [{ "type": "text", "text": "Created basket bsk_a1b2c3" }],
  "structuredContent": { "basket_id": "bsk_a1b2c3" }
}

// → tools/call
{
  "name": "add_item",
  "arguments": { "basket_id": "bsk_a1b2c3", "sku": "..." }
}
```

The model is responsible for carrying `basket_id` forward; the server stores the cart contents under that key and looks them up on each call.

When designing handles, servers should consider:

- **Authorization.** For authenticated servers, a handle is a name, not a capability. The server should validate the caller's authorization against the handle on every call. For unauthenticated servers, where the handle is necessarily a bearer token, it should be generated with sufficient entropy (e.g., a UUIDv4) and given a bounded lifetime.
- **Opacity.** Handles that encode internal structure invite parsing or guessing; opaque identifiers do not.
- **Lifetime.** Because handles outlive any single connection, the server's retention policy should be stated in the creation tool's description (e.g., "baskets expire after 24 hours of inactivity") so the model can see it when deciding to create state.
- **Expiry errors.** A call against an expired or unknown handle should return a tool execution error that says so, so the model can recover by creating a new one.

Error Handling

Tools use two error reporting mechanisms:

1. **Protocol Errors** indicate issues with the request structure itself that models are less likely to be able to fix:

   - Unknown tool
   - Malformed requests (requests that fail to satisfy CallToolRequest schema)
   - Server errors

   They are returned as standard JSON-RPC errors:

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

2. **Tool Execution Errors** contain actionable feedback that language models can use to self-correct and retry with adjusted parameters:

    - API failures
    - Input validation errors (e.g., date in wrong format, value out of range)
    - Business logic errors

    They are reported in tool results with `isError: true`:

```
{
  "jsonrpc": "2.0",
  "id": 4,
  "result": {
    "resultType": "complete",
    "content": [
      {
        "type": "text",
        "text": "Invalid departure date: must be in the future. Current date is
08/08/2025."
      }
    ],
    "isError": true
  }
}
```

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

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
    - Follow the `$ref` resolution requirements when validating tool inputs and outputs against `inputSchema` and `outputSchema`
    - Implement timeouts for tool calls
    - Log tool usage for audit purposes

7.6 Utilities

7.6.1 Caching

The Model Context Protocol (MCP) supports caching for some results. This allows clients to cache responses and reduce unnecessary re-fetching. Caching is complementary to change notifications—both mechanisms can coexist.

Cacheable Results

Servers MUST include caching hints on results with `resultType: "complete"` returned by the following operations:

- `server/discover`
- `tools/list`
- `prompts/list`
- `resources/list`
- `resources/templates/list`
- `resources/read`

Interim results with `resultType: "input_required"` (see multi round-trip requests) are not cacheable and carry no caching hints.

Cache Key

A cached response is identified by the request method together with the request parameters that affect the result (for example, the `uri` for `resources/read`, or the `cursor` for paginated list requests). Clients **MUST NOT** serve a cached response for a request whose method or parameters differ from the request that produced it.

Results produced by retrying a request through the multi round-trip requests mechanism—that is, requests carrying `inputResponses` or `requestState`—**MUST NOT** be cached, as they depend on inputs that are not part of the cache key.

Cacheable Model

Cacheable Results in MCP use two fields to provide caching hints to clients:

- The **Time-to-live (TTL) Field**, `ttlMs`, is an integer value in milliseconds specifying how long the client MAY consider the result fresh.
- The **Cache Scope Field**, `cacheScope`, indicates the intended scope of the cached response, either `"public"` or `"private"`.

Time-to-Live (TTL) Field

The `ttlMs` field is a hint from the server indicating how long, in milliseconds, the client MAY consider the result fresh. Semantics are analogous to HTTP `Cache-Control: max-age`.

- If `ttlMs` is `0`, the response **SHOULD** be considered immediately stale. The client MAY re-fetch every time the result is needed.
- If `ttlMs` is positive, the client **SHOULD** consider the result fresh for that many milliseconds after receiving the response.

- If `ttlMs` is absent, clients **SHOULD** assume a default of `0` (immediately stale) and rely on their own caching heuristics or notifications. This should only occur in older server versions.
- If `ttlMs` is negative, clients **SHOULD** ignore it and treat it as `0`.

Servers **MUST** provide a `ttlMs` value that is `>= 0`.

> **NOTE**
>
> TTL is a **freshness hint**, not a guarantee. Servers MAY change the underlying data before the TTL expires. The TTL tells the client how long it can reasonably avoid re-fetching, not how long the data is guaranteed to remain unchanged.

Freshness Calculation

A client records the local time at which the response was received (`t_received`). The response is considered **fresh** while:

```
now < t_received + ttlMs
```

Once the TTL expires, the response is **stale** and the client **SHOULD** re-fetch on next access.

Clients **SHOULD NOT** treat TTL as a polling interval that triggers automatic background refetches. The TTL is a freshness hint: the client checks freshness when it needs the data, and re-fetches only if stale. Implementations that do choose to poll **MUST** apply jitter and backoff.

Clients **MAY** re-fetch before the TTL expires if they have reason to believe the data has changed (e.g., receiving an unexpected error on a tool call indicating the method was not found or the parameters were invalid).

Clients **MAY** serve stale responses if errors occur during re-fetching (e.g., network issues, server downtime).

Cache Scope Field

The `cacheScope` field controls who may cache a response, analogous to HTTP `Cache-Control: public` vs `Cache-Control: private`.

Value	Meaning
`"public"`	The response does not contain user-specific data. Any client, shared gateway, or caching proxy **MAY** store and serve the cached response to any user.
`"private"`	The response contains private data that is not meant to be shared between callers. Cached responses **MAY** be reused for the same authorization context. Caches **MUST NOT** be shared across authorization contexts (e.g. a different access token requires a different cache).

Choosing a Cache Scope

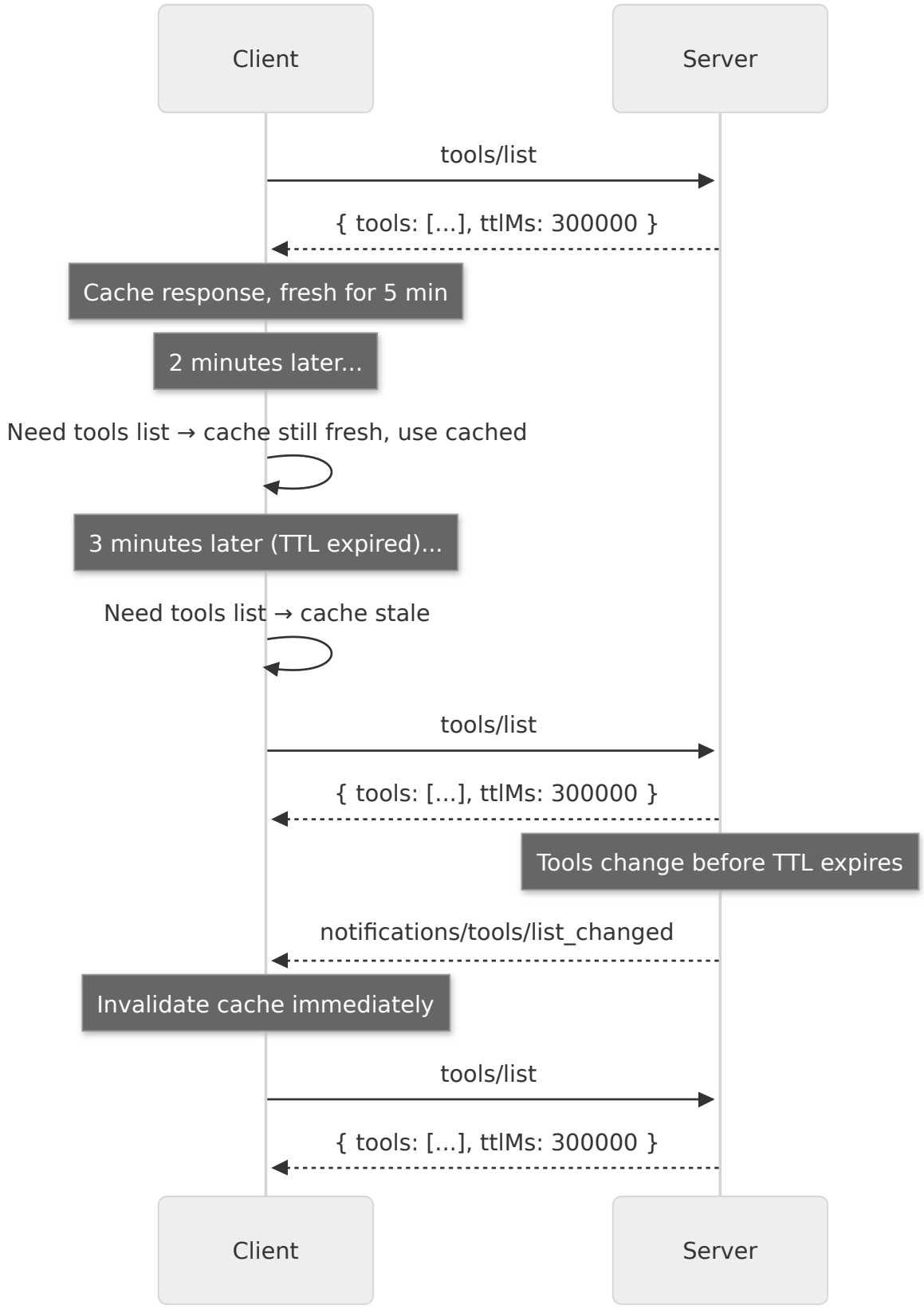
- **`"public"`** is appropriate for lists of tools, prompts, and resource templates when they are identical for all users.
- **`"private"`** is appropriate for `resources/read` results that depend on the authenticated user, or for filtered list results that vary per user.

Interaction with Notifications

TTL and server-push notifications are complementary:

- A server **MAY** provide `ttlMs` without advertising `listChanged: true` in its capabilities. In this case, the client relies entirely on TTL-based freshness.
- A server **MAY** advertise `listChanged: true` **and** provide `ttlMs`. In this case, the client can use the TTL to avoid unnecessary refetches between notifications, and the notification acts as an immediate invalidation signal.

When a relevant notification is received while a cached response is still fresh, the notification **invalidates** the cached response and it should be considered immediately stale.



Interaction with Pagination

When a list result is paginated, each page is an independently cacheable response—consistent with how HTTP `Cache-Control` treats paginated resources.

- Each page response carries its own `ttlMs` value. The freshness clock for each page starts at the time that page was received.
- Servers **MAY** return different `ttlMs` values on different pages (e.g., a longer TTL for early pages of a stable list, a shorter TTL for the final page).
- When a cached page expires, the client **SHOULD** re-fetch that page using its cursor.
- There is no cross-page consistency guarantee. If the underlying data changes between page fetches, clients may observe duplicates or gaps.
- Clients that require a consistent snapshot of the full list **SHOULD** re-fetch from the beginning (without a cursor).
- If a cursor becomes invalid (e.g., the server returns an error for a previously valid cursor), the client **SHOULD** discard all cached pages and re-fetch from the beginning.

Servers **MUST** apply the same `cacheScope` to all response pages for a given list request. For example, if the first page of a `tools/list` response has `cacheScope: "private"`, all subsequent pages for that request **MUST** also be `"private"`.

Security Considerations

A `cacheScope` of `"public"` indicates that the response does not contain user-specific data and can be safely shared. Servers MUST be aware that responses with a `"public"` `cacheScope` may be shared between callers even if the Result is coming from an authenticated endpoint. For example, the Result from an authenticated `tools/list` call with a `"public"` `cacheScope` may be cached by a client and may be shared outside of the initial requests authorization context. (i.e. different access tokens can leverage the same cache).

Server implementors:

- should ensure that the `cacheScope` correctly reflects the intended visibility of the primitive.
- MUST apply appropriate per-primitive access controls, and MUST NOT rely on `cacheScope` alone to prevent unauthorized access to primitives.

7.6.2 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.

> **NOTE**
>
> For brevity, the request examples on this page omit the `_meta` request metadata (`io.modelcontextprotocol/protocolVersion` , `io.modelcontextprotocol/clientInfo` , and `io.modelcontextprotocol/clientCapabilities`). Every request **MUST** include the required `_meta` fields; see `_meta` .

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": {
    "resultType": "complete",
    "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": {
    "resultType": "complete",
    "completion": {
      "values": ["flask"],
      "total": 1,
      "hasMore": false
    }
  }
}
```

Reference Types

The protocol supports two types of completion references:

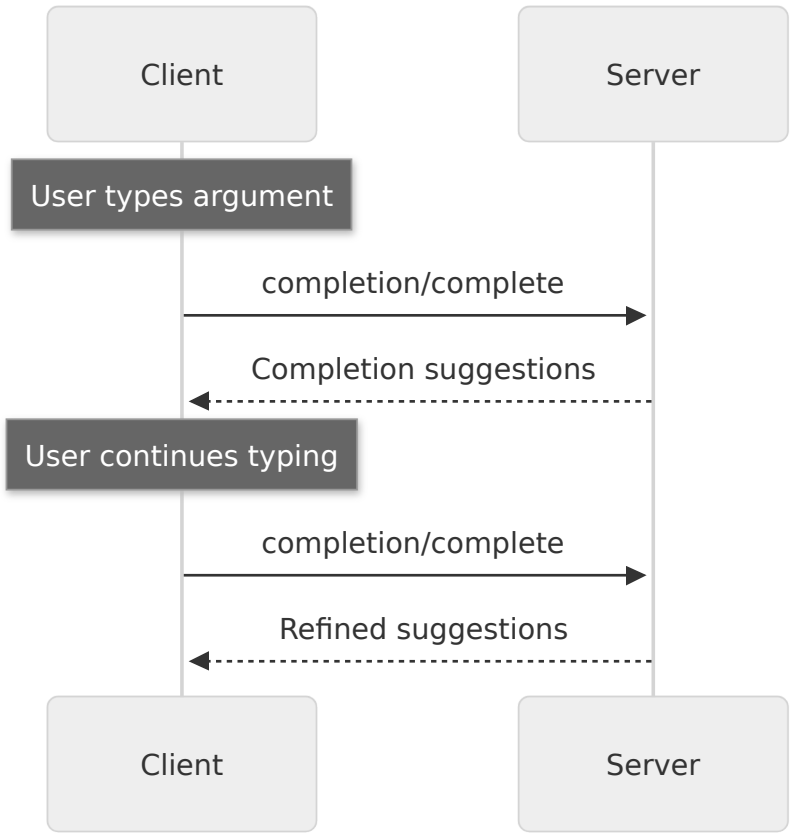
Type	Description	Example
`ref/prompt`	References a prompt by name	`{"type": "ref/prompt", "name": "code_review"}`
`ref/resource`	References a resource URI or URI template	`{"type": "ref/resource", "uri": "file:///{path}"}`

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 `ResourceTemplateReference`. For `ResourceTemplateReference`, `uri` is a URI or URI template.
- `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

7.6.3 Logging

> **WARNING**
>
> **Deprecated**: The Logging feature is deprecated as of protocol version `2026-07-28` (SEP-2577). Under the feature lifecycle policy, it remains in the specification for at least twelve months after this revision's release before it becomes eligible for removal. New implementations **SHOULD NOT** adopt it; existing implementations **SHOULD** migrate to logging to `stderr` for stdio transports, or to OpenTelemetry for structured observability. See the deprecated features registry.

The Model Context Protocol (MCP) provides a standardized way for servers to send structured log messages to clients. Clients control logging verbosity per-request via `_meta`, 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

Requesting Log Messages

Per-request log level

To receive log messages for a specific request, include `io.modelcontextprotocol/logLevel` in the request's `_meta`. The server **MUST NOT** emit `notifications/message` for a request that does not include this field.

When the field is present, the server **MAY** send `notifications/message` notifications at or above the requested level on the response stream of that request, before the final response. `notifications/message` is request-scoped: the server **MUST NOT** deliver it on a `subscriptions/listen` stream or on any stream other than the one carrying the response to the request that set the log level.

Protocol Messages

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
      }
    }
  }
}
```

Error Handling

If the `io.modelcontextprotocol/logLevel` value carried in a request's `_meta` is not a recognized log level, the server **SHOULD** reject that request with a standard JSON-RPC error:

- Invalid log level: `-32602` (Invalid params)
- Internal 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

7.6.4 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.

> **NOTE**
>
> For brevity, the request examples on this page omit the `_meta` request metadata (`io.modelcontextprotocol/protocolVersion` , `io.modelcontextprotocol/clientInfo` , and `io.modelcontextprotocol/clientCapabilities`). Every request **MUST** include the required `_meta` fields; see `_meta` .

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

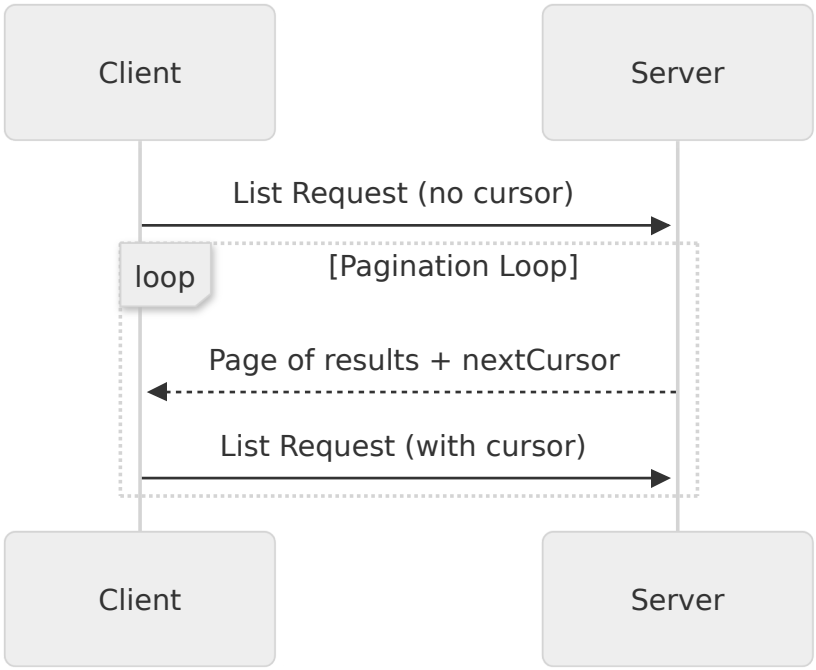
```
{
  "jsonrpc": "2.0",
  "id": "123",
  "result": {
    "resultType": "complete",
    "resources": [...],
    "nextCursor": "eyJwYWdlIjogM30=",
    "ttlMs": 300000,
    "cacheScope": "public"
  }
}
```

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 make any determination based on cursor value other than whether a non-null value was provided (e.g. an empty string is a valid cursor and thus **MUST NOT** be treated as the end of results)

Error Handling

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

8 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`

```ts
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

```ts
JSONRPCResponse: JSONRPCResultResponse | JSONRPCErrorResponse
```

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

JSONRPCResultResponse

```ts
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

```ts
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 result that indicates success but carries no data.

`Icon`

```
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 take 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.

`InputResponseRequestParams`

```ts
interface InputResponseRequestParams {
  _meta: RequestMetaObject;
  inputResponses?: InputResponses;
  requestState?: string;
}
```

Common params for any request.

> **_meta: RequestMetaObject**
> Inherited from RequestParams._meta

> **inputResponses?: InputResponses**

> **requestState?: string**

`JSONArray`

```ts
JSONArray: JSONValue[]
```

`JSONObject`

```ts
JSONObject: { [key: string]: JSONValue }
```

Type Declaration

- [key: string]: JSONValue

`JSONValue`

```
JSONValue: string | number | boolean | null | JSONObject | JSONArray
```

`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

Deprecated

Deprecated as of protocol version 2026-07-28 (SEP-2577). Remains in the specification for at least twelve months; see the deprecated features registry.

`MetaObject`

```
MetaObject: Record<string, unknown>
```

Represents the contents of a `_meta` field, which clients and servers use to attach additional metadata to their interactions.

Certain key names are reserved by MCP for protocol-level metadata; implementations MUST NOT make assumptions about values at these keys. Additionally, specific schema definitions may reserve particular names for purpose-specific metadata, as declared in those definitions.

Valid keys have two segments:

Prefix:

- Optional — 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 may 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 start and end with an alphanumeric character (`[a-z0-9A-Z]`).
- Interior characters may be alphanumeric, hyphens (`-`), underscores (`_`), or dots (`.`).

See

General fields: `_meta` for more details.

`NotificationMetaObject`

```
interface NotificationMetaObject {
  "io.modelcontextprotocol/subscriptionId"?: RequestId;
  [key: string]: unknown;
}
```

Extends MetaObject with additional notification-specific fields. All key naming rules from `MetaObject` apply.

See

- MetaObject for key naming rules and reserved prefixes.
- General fields: `_meta` for more details.

> **`"io.modelcontextprotocol/subscriptionId"?: RequestId`**
>
> Identifies the subscription stream a notification was delivered on. The server MUST include this key on every notification delivered via a subscriptions/listen stream, so the client can correlate the notification with the originating subscription. The key is absent on notifications not delivered via a subscription stream (e.g. progress notifications for an in-flight request), which is why it is optional here.
>
> The value is the JSON-RPC ID of the `subscriptions/listen` request that opened the stream.

`NotificationParams`

```
interface NotificationParams {
  _meta?: NotificationMetaObject;
}
```

Common params for any notification.

> **`_meta?: NotificationMetaObject`**

`PaginatedRequestParams`

```
interface PaginatedRequestParams {
  _meta: RequestMetaObject;
  cursor?: string;
}
```

Common params for paginated requests.

▼ **Example: List request with cursor**

```
{
  "_meta": {
    "io.modelcontextprotocol/protocolVersion": "2026-07-28",
    "io.modelcontextprotocol/clientInfo": {
      "name": "ExampleClient",
      "version": "1.0.0"
    },
    "io.modelcontextprotocol/clientCapabilities": {}
  },
  "cursor": "eyJwYWdlIjogMn0="
}
```
Copy

> **_meta: RequestMetaObject**
>
> Inherited from RequestParams._meta

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

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.

RequestMetaObject

```
interface RequestMetaObject {
  progressToken?: ProgressToken;
  "io.modelcontextprotocol/protocolVersion": string;
  "io.modelcontextprotocol/clientInfo"?: Implementation;
  "io.modelcontextprotocol/clientCapabilities": ClientCapabilities;
  "io.modelcontextprotocol/logLevel"?: LoggingLevel;
  [key: string]: unknown;
}
```

Extends MetaObject with additional request-specific fields. All key naming rules from `MetaObject` apply.

See

- MetaObject for key naming rules and reserved prefixes.
- General fields: `_meta` for more details.

> **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.

> **"io.modelcontextprotocol/protocolVersion": string**
>
> The MCP Protocol Version being used for this request. Required.
>
> For the HTTP transport, this value MUST match the `MCP-Protocol-Version` header; otherwise the server MUST return a `400 Bad Request`. If the server does not support the requested version, it MUST return an UnsupportedProtocolVersionError.

> **"io.modelcontextprotocol/clientInfo"?: Implementation**
>
> Identifies the client software making the request. Clients SHOULD include this field on every request unless specifically configured not to do so.
>
> The Implementation schema requires `name` and `version`; other fields are optional.
>
> The value is self-reported by the client and is not verified by the protocol. It is intended for display, logging, and debugging. Servers SHOULD NOT use it to change their behavior, and SHOULD NOT rely on it for security decisions.

> **"io.modelcontextprotocol/clientCapabilities": ClientCapabilities**
>
> The client's capabilities for this specific request. Required.
>
> Capabilities are declared per-request rather than once at initialization; an empty object means the client supports no optional capabilities. Servers MUST NOT infer capabilities from prior requests.

> **"io.modelcontextprotocol/logLevel"?: LoggingLevel**
>
> The desired log level for this request. Optional.
>
> If absent, the server MUST NOT send any notifications/message notifications for this request. The client opts in to log messages by explicitly setting a level. Replaces the former `logging/setLevel` RPC.
>
> **Deprecated**
>
> Deprecated as of protocol version 2026-07-28 (SEP-2577). Remains in the specification for at least twelve months; see the deprecated features registry.

`RequestParams`

```
interface RequestParams {
  _meta: RequestMetaObject;
}
```

Common params for any request.

> **_meta: RequestMetaObject**

`Result`

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

Common result fields.

> **`_meta?: ResultMetaObject`**

> **`resultType: string`**
>
> Indicates the type of the result, which allows the client to determine how to parse the result object.
>
> Servers implementing this protocol version MUST include this field. For backward compatibility, when a client receives a result from a server implementing an earlier protocol version (which does not include `resultType`), the client MUST treat the absent field as `"complete"`.

`ResultMetaObject`

```ts
interface ResultMetaObject {
  "io.modelcontextprotocol/serverInfo"?: Implementation;
  [key: string]: unknown;
}
```

Extends MetaObject with additional result-specific fields. All key naming rules from `MetaObject` apply.

See

- MetaObject for key naming rules and reserved prefixes.
- General fields: `_meta` for more details.

> **`"io.modelcontextprotocol/serverInfo"?: Implementation`**
>
> Identifies the server software producing the response. Servers SHOULD include this field on every response unless specifically configured not to do so.
>
> The Implementation schema requires `name` and `version`; other fields are optional.
>
> The value is self-reported by the server and is not verified by the protocol. It is intended for display, logging, and debugging. Clients SHOULD NOT use it to change their behavior, and SHOULD NOT rely on it for security decisions.

`ResultType`

```ts
ResultType: "complete" | "input_required" | string
```

Indicates the type of a Result object, allowing the client to determine how to parse the response.

complete - the request completed successfully and the result contains the final content. input_required - the request requires additional input and the result contains an InputRequiredResult object with instructions for the client to provide additional input before retrying the original request.

`Role`

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

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

Errors

`Error`

```
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.).

`HEADER_MISMATCH`

```
HEADER_MISMATCH: -32020
```

Error code returned when the HTTP headers of a request do not match the corresponding values in the request body, or required headers are missing or malformed.

`HeaderMismatchError`

```
interface HeaderMismatchError {
  jsonrpc: "2.0";
  id?: RequestId;
  error: Error & { code: -32020 };
}
```

Returned when a server rejects a request because the values in the HTTP headers do not match the corresponding values in the request body, or because required headers are missing or malformed. For HTTP, the response status code MUST be `400 Bad Request`.

▼ **Example: Header mismatch**

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "error": {
    "code": -32020,
    "message": "Header mismatch: Mcp-Name header value 'foo' does not match body value
'bar'"
  }
}
```

Copy

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

> **id?: RequestId**
> Inherited from JSONRPCErrorResponse.id

> **error: Error & { code: -32020 }**

`InternalError`

```
interface InternalError {
  message: string;
  data?: unknown;
  code: -32603;
}
```

A JSON-RPC error indicating that an internal error occurred on the receiver. This error is returned when the receiver encounters an unexpected condition that prevents it from fulfilling the request.

See

JSON-RPC 2.0 Error Object

▼ **Example: Unexpected error**


```
{
  "code": -32603,
  "message": "Internal error"
}
```

Copy

> **message: string**
>
> A short description of the error. The message SHOULD be limited to a concise single sentence.
>
> Inherited from Error.message

> **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.).
>
> Inherited from Error.data

> **code: -32603**

> The error type that occurred.
>
> Overrides Error.code

`InvalidParamsError`

```
interface InvalidParamsError {
  message: string;
  data?: unknown;
  code: -32602;
}
```

A JSON-RPC error indicating that the method parameters are invalid or malformed.

In MCP, this error is returned in various contexts when request parameters fail validation:

- **Tools**: Unknown tool name or invalid tool arguments
- **Prompts**: Unknown prompt name or missing required arguments
- **Pagination**: Invalid or expired cursor values
- **Logging**: Invalid log level
- **Elicitation**: Server requests an elicitation mode not declared in client capabilities
- **Sampling**: Missing tool result or tool results mixed with other content

See

JSON-RPC 2.0 Error Object

▼ **Example: Unknown tool**

```
{
  "code": -32602,
  "message": "Unknown tool: invalid_tool_name"
} Copy
```

▼ **Example: Invalid tool arguments**

```
{
  "code": -32602,
  "message": "Invalid arguments for tool calculate: Missing required property
'expression'"
} Copy
```

▼ **Example: Unknown prompt**

```
{
  "code": -32602,
  "message": "Unknown prompt: invalid_prompt_name"
} Copy
```

▼ **Example: Invalid cursor**

```
{
  "code": -32602,
  "message": "Invalid cursor"
} Copy
```

> **message: string**
>
> A short description of the error. The message SHOULD be limited to a concise single sentence.
>
> Inherited from Error.message

> **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.).
>
> Inherited from Error.data

> **code: -32602**
>
> The error type that occurred.
>
> Overrides Error.code

`InvalidRequestError`

```
interface InvalidRequestError {
  message: string;
  data?: unknown;
  code: -32600;
}
```

A JSON-RPC error indicating that the request is not a valid request object. This error is returned when the message structure does not conform to the JSON-RPC 2.0 specification requirements for a request (e.g., missing required fields like `jsonrpc` or `method`, or using invalid types for these fields).

See

JSON-RPC 2.0 Error Object

> **message: string**
>
> A short description of the error. The message SHOULD be limited to a concise single sentence.
>
> Inherited from Error.message

> **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.).
>
> Inherited from Error.data

> **code: -32600**
>
> The error type that occurred.
>
> Overrides Error.code

`MethodNotFoundError`

```ts
interface MethodNotFoundError {
  message: string;
  data?: unknown;
  code: -32601;
}
```

A JSON-RPC error indicating that the requested method does not exist or is not available.

In MCP, a server returns this error when a client invokes a method the server does not implement — either a genuinely unknown method, or one gated behind a server capability the server did not advertise (e.g., calling `prompts/list` when the `prompts` capability was not advertised).

A request that requires a client capability the client did not declare is signalled instead by MissingRequiredClientCapabilityError (`-32021`).

See

JSON-RPC 2.0 Error Object

▼ **Example: Prompts not supported**

```
{
  "code": -32601,
  "message": "Prompts not supported",
  "data": {
    "reason": "Server does not support the prompts capability"
  }
} Copy
```

> **`message: string`**
>
> A short description of the error. The message SHOULD be limited to a concise single sentence.
>
> Inherited from Error.message

> **`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.).
>
> Inherited from Error.data

> **`code: -32601`**
>
> The error type that occurred.
>
> Overrides Error.code

`MISSING_REQUIRED_CLIENT_CAPABILITY`

```ts
MISSING_REQUIRED_CLIENT_CAPABILITY: -32021
```

Error code returned when a server requires a client capability that was not declared in the request's `clientCapabilities`.

`MissingRequiredClientCapabilityError`

```
interface MissingRequiredClientCapabilityError {
  jsonrpc: "2.0";
  id?: RequestId;
  error: Error & {
    code: -32021;
    data: { requiredCapabilities: ClientCapabilities };
  };
}
```

Returned when processing a request requires a capability the client did not declare in `clientCapabilities`. For HTTP, the response status code MUST be `400 Bad Request`.

▼ **Example: Missing elicitation capability**

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "error": {
    "code": -32021,
    "message": "Server requires the elicitation capability for this request",
    "data": {
      "requiredCapabilities": {
        "elicitation": {}
      }
    }
  }
} Copy
```

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

> **id?: RequestId**
> Inherited from JSONRPCErrorResponse.id

> **error: Error & { code: -32021; data: { requiredCapabilities: ClientCapabilities }; }**

`ParseError`

```
interface ParseError {
  message: string;
  data?: unknown;
  code: -32700;
}
```

A JSON-RPC error indicating that invalid JSON was received by the server. This error is returned when the server cannot parse the JSON text of a message.

See

JSON-RPC 2.0 Error Object

▼ **Example: Invalid JSON**

```
{
  "code": -32700,
  "message": "Parse error: Invalid JSON"
} Copy
```

> **message: string**
>
> A short description of the error. The message SHOULD be limited to a concise single sentence.
>
> Inherited from Error.message

> **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.).
>
> Inherited from Error.data

> **code: -32700**
>
> The error type that occurred.
>
> Overrides Error.code

`UNSUPPORTED_PROTOCOL_VERSION`

```
UNSUPPORTED_PROTOCOL_VERSION: -32022
```

Error code returned when the request's protocol version is not supported by the server.

`UnsupportedProtocolVersionError`

```
interface UnsupportedProtocolVersionError {
  jsonrpc: "2.0";
  id?: RequestId;
  error: Error & {
    code: -32022;
    data: { supported: string[]; requested: string };
  };
}
```

Returned when the request's protocol version is unknown to the server or unsupported (e.g., a known experimental or draft version the server has chosen not to implement). For HTTP, the response status code MUST be `400 Bad Request`.

▼ **Example: Unsupported protocol version**

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "error": {
    "code": -32022,
    "message": "Unsupported protocol version",
    "data": {
      "supported": [
        "2026-07-28",
        "2025-11-25"
      ],
      "requested": "1900-01-01"
    }
  }
}
```
Copy

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

> **id?: RequestId**
> Inherited from JSONRPCErrorResponse.id

> **error: Error & { code: -32022; data: { supported: string[]; requested: string }; }**

Content

`AudioContent`

```
interface AudioContent {
  type: "audio";
  data: string;
  mimeType: string;
  annotations?: Annotations;
  _meta?: MetaObject;
}
```

Audio provided to or from an LLM.

▼ **Example: `audio/wav` content**


```
{
  "type": "audio",
  "data": "UklGRiQAAABXQVZFZm10IBAAAAABAAEARKwAAIhYAQACABAAZGF0YQAAAAA=",
  "mimeType": "audio/wav"
}
```
Copy

> **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?: MetaObject**

BlobResourceContents

```
interface BlobResourceContents {
  uri: string;
  mimeType?: string;
  _meta?: MetaObject;
  blob: string;
}
```

▼ **Example: Image file contents**

```
{
  "uri": "file:///example.png",
  "mimeType": "image/png",
  "blob":
"iVBORw0KGgoAAAANSUhEUgAAAAEAAAABCAYAAAAfFcSJAAAADUlEQVR42mNk+M9QDwADhgGAWjR9awAAAABJRU
5ErkJggg=="
} Copy
```

> **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?: MetaObject**
> Inherited from ResourceContents._meta

> **blob: string**
>
> A base64-encoded string representing the binary data of the item.

ContentBlock

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

`EmbeddedResource`

```
interface EmbeddedResource {
  type: "resource";
  resource: TextResourceContents | BlobResourceContents;
  annotations?: Annotations;
  _meta?: MetaObject;
}
```

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.

▼ **Example: Embedded file resource with annotations**

```
{
  "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"
  }
} Copy
```

> **`type: "resource"`**

> **`resource: TextResourceContents | BlobResourceContents`**

> **`annotations?: Annotations`**
>
> Optional annotations for the client.

> **`_meta?: MetaObject`**

`ImageContent`

```ts
interface ImageContent {
  type: "image";
  data: string;
  mimeType: string;
  annotations?: Annotations;
  _meta?: MetaObject;
}
```

An image provided to or from an LLM.

▼ **Example: `image/png` content with annotations**

```
{
  "type": "image",
  "data":
"iVBORw0KGgoAAAANSUhEUgAAAAEAAAABCAYAAAAfFcSJAAAADUlEQVR42mNk+M9QDwADhgGAWjR9awAAAABJRU
5ErkJggg==",
  "mimeType": "image/png",
  "annotations": {
    "audience": ["user"],
    "priority": 0.9
  }
} Copy
```

> **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?: MetaObject**

`ResourceLink`

```ts
interface ResourceLink {
  icons?: Icon[];
  name: string;
  title?: string;
  uri: string;
  description?: string;
  mimeType?: string;
  annotations?: Annotations;
  size?: number;
  _meta?: MetaObject;
  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.

▼ **Example: File resource link**

```
{
  "type": "resource_link",
  "uri": "file:///project/src/main.rs",
  "name": "main.rs",
  "description": "Primary application entry point",
  "mimeType": "text/x-rust"
} Copy
```

> **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?: MetaObject

Inherited from Resource._meta

type: "resource_link"

`TextContent`

```
interface TextContent {
  type: "text";
  text: string;
  annotations?: Annotations;
  _meta?: MetaObject;
}
```

Text provided to or from an LLM.

▼ **Example: Text content**

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

Copy

type: "text"

text: string

The text content of the message.

annotations?: Annotations

Optional annotations for the client.

_meta?: MetaObject

TextResourceContents

```
interface TextResourceContents {
  uri: string;
  mimeType?: string;
  _meta?: MetaObject;
  text: string;
}
```

▼ **Example: Text file contents**

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

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?: MetaObject

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.

▼ **Example: Completion request**

```
{
  "jsonrpc": "2.0",
  "id": "completion-example",
  "method": "completion/complete",
  "params": {
    "_meta": {
      "io.modelcontextprotocol/protocolVersion": "2026-07-28",
      "io.modelcontextprotocol/clientInfo": {
        "name": "ExampleClient",
        "version": "1.0.0"
      },
      "io.modelcontextprotocol/clientCapabilities": {}
    },
    "ref": {
      "type": "ref/prompt",
      "name": "code_review"
    },
    "argument": {
      "name": "language",
      "value": "py"
    }
  }
} Copy
```

> **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`

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

Parameters for a `completion/complete` request.

▼ **Example: Prompt argument completion**

```
{
  "_meta": {
    "io.modelcontextprotocol/protocolVersion": "2026-07-28",
    "io.modelcontextprotocol/clientInfo": {
      "name": "ExampleClient",
      "version": "1.0.0"
    },
    "io.modelcontextprotocol/clientCapabilities": {}
  },
  "ref": {
    "type": "ref/prompt",
    "name": "code_review"
  },
  "argument": {
    "name": "language",
    "value": "py"
  }
} Copy
```

▼ **Example: Prompt argument completion with context**

```
{
  "_meta": {
    "io.modelcontextprotocol/protocolVersion": "2026-07-28",
    "io.modelcontextprotocol/clientInfo": {
      "name": "ExampleClient",
      "version": "1.0.0"
    },
    "io.modelcontextprotocol/clientCapabilities": {}
  },
  "ref": {
    "type": "ref/prompt",
    "name": "code_review"
  },
  "argument": {
    "name": "framework",
    "value": "fla"
  },
  "context": {
    "arguments": {
      "language": "python"
    }
  }
}
```
Copy

> **`_meta: RequestMetaObject`**
> 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.

`CompleteResultResponse`

```
interface CompleteResultResponse {
  jsonrpc: "2.0";
  id: RequestId;
  result: CompleteResult;
}
```

A successful response from the server for a completion/complete request.

▼ **Example: Completion result response**

```
{
  "jsonrpc": "2.0",
  "id": "completion-example",
  "result": {
    "resultType": "complete",
    "completion": {
      "values": ["flask"],
      "total": 1,
      "hasMore": false
    }
  }
} Copy
```

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

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

> **result: CompleteResult**
> Overrides JSONRPCResultResponse.result

`CompleteResult`

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

The result returned by the server for a completion/complete request.

▼ **Example: Single completion value**

```
{
  "resultType": "complete",
  "completion": {
    "values": ["flask"],
    "total": 1,
    "hasMore": false
  }
} Copy
```

▼ **Example: Multiple completion values with more available**

```
{
  "resultType": "complete",
  "completion": {
    "values": ["python", "pytorch", "pyside"],
    "total": 10,
    "hasMore": true
  }
} Copy
```

_meta?: ResultMetaObject
Inherited from Result._meta

resultType: string

Indicates the type of the result, which allows the client to determine how to parse the result object.

Servers implementing this protocol version MUST include this field. For backward compatibility, when a client receives a result from a server implementing an earlier protocol version (which does not include `resultType`), the client MUST treat the absent field as `"complete"`.

Inherited from Result.resultType

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`

```ts
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`

```ts
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 {
  method: "elicitation/create";
  params: ElicitRequestParams;
}
```

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

▼ **Example: Elicitation request**

```
{
  "method": "elicitation/create",
  "params": {
    "mode": "form",
    "message": "Please provide your GitHub username",
    "requestedSchema": {
      "type": "object",
      "properties": {
        "name": {
          "type": "string",
          "title": "GitHub Username",
          "description": "Your GitHub username"
        }
      },
      "required": ["name"]
    }
  }
} Copy
```

> **method: "elicitation/create"**

> **params: ElicitRequestParams**

`ElicitRequestParams`

```
ElicitRequestParams: ElicitRequestFormParams | ElicitRequestURLParams
```

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

`ElicitResult`

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

The result returned by the client for an elicitation/create request.

▼ **Example: Input single field**

```
{
  "action": "accept",
  "content": {
    "name": "octocat"
  }
} Copy
```

▼ **Example: Input multiple fields**

```
{
  "action": "accept",
  "content": {
    "name": "Monalisa Octocat",
    "email": "octocat@github.com",
    "age": 30
  }
} Copy
```

▼ **Example: Accept URL mode (no content)**

```
{
  "action": "accept"
} Copy
```

> **action: "accept" | "decline" | "cancel"**
>
> The user action in response to the elicitation.
>
> - `"accept"`: User submitted the form/confirmed the action
> - `"decline"`: User explicitly declined the action
> - `"cancel"`: User dismissed without making an explicit choice

> **content?: { [key: string]: string | number | boolean | 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;
}
```

▼ **Example: Boolean input schema**

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

> **type: "boolean"**

> **title?: string**

> **description?: string**

> **default?: boolean**

`ElicitRequestFormParams`

```ts
interface ElicitRequestFormParams {
  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.

▼ **Example: Elicit single field**

```
{
  "mode": "form",
  "message": "Please provide your GitHub username",
  "requestedSchema": {
    "type": "object",
    "properties": {
      "name": {
        "type": "string"
      }
    },
    "required": ["name"]
  }
} Copy
```

▼ **Example: Elicit multiple fields**

```
{
  "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"]
  }
} Copy
```

> **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 {
  mode: "url";
  message: string;
  url: string;
}
```

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

▼ **Example: Elicit sensitive data**

```
{
  "mode": "url",
  "url": "https://mcp.example.com/ui/set&#x5F;api&#x5F;key",
  "message": "Please provide your API key to continue."
}
```

Copy

> **mode: "url"**
>
> The elicitation mode.

> **message: string**
>
> The message to present to the user explaining why the interaction is needed.

> **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;
}
```

▼ **Example: Number input schema**

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

> **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;
}
```

▼ **Example: Email input schema**

```
{
  "type": "string",
  "title": "Display Name",
  "description": "Description text",
  "minLength": 3,
  "maxLength": 50,
  "format": "email",
  "default": "user@example.com"
} Copy
```

> **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.

▼ **Example: Titled color multi-select schema**

```
{
  "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"]
} Copy
```

> **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`

```
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.

▼ **Example: Titled color select schema**

```
{
  "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"
} Copy
```

> **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`

```
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.

▼ **Example: Color multi-select schema**

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

> **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.

▼ **Example: Color select schema**

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

> **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.

`notifications/cancelled`

`CancelledNotification`

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

This notification is sent by the client to indicate that it is cancelling a request it previously issued.

On stdio, the server also sends this notification, solely to terminate a subscriptions/listen stream: it references the ID of the `subscriptions/listen` request that opened the stream. Servers MUST NOT use this notification to cancel any other 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.

▼ **Example: User-requested cancellation**

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

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

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

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

`CancelledNotificationParams`

```
interface CancelledNotificationParams {
  _meta?: NotificationMetaObject;
  requestId: RequestId;
  reason?: string;
}
```

Parameters for a `notifications/cancelled` notification.

▼ **Example: User-requested cancellation**

```
{
  "requestId": "123",
  "reason": "User requested cancellation"
} Copy
```

> **_meta?: NotificationMetaObject**
> Inherited from NotificationParams._meta

> **requestId: RequestId**
>
> The ID of the request to cancel.
>
> This MUST correspond to the ID of a request the client previously issued.

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

notifications/message

LoggingMessageNotification

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

JSONRPCNotification of a log message passed from server to client. The client opts in by setting `"io.modelcontextprotocol/logLevel"` in a request's `_meta`.

Deprecated

Deprecated as of protocol version 2026-07-28 (SEP-2577). Remains in the specification for at least twelve months; see the deprecated features registry.

▼ **Example: Log database connection failed**

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

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

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

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

`LoggingMessageNotificationParams`

```
interface LoggingMessageNotificationParams {
  _meta?: NotificationMetaObject;
  level: LoggingLevel;
  logger?: string;
  data: unknown;
}
```

Parameters for a `notifications/message` notification.

Deprecated

Deprecated as of protocol version 2026-07-28 (SEP-2577). Remains in the specification for at least twelve months; see the deprecated features registry.

▼ **Example: Log database connection failed**

```
{
  "level": "error",
  "logger": "database",
  "data": {
    "error": "Connection failed",
    "details": {
      "host": "localhost",
      "port": 5432
    }
  }
} Copy
```

> **_meta?: NotificationMetaObject**
> 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.

▼ **Example: Progress message**

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

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

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

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

ProgressNotificationParams

```
interface ProgressNotificationParams {
  _meta?: NotificationMetaObject;
  progressToken: ProgressToken;
  progress: number;
  total?: number;
  message?: string;
}
```

Parameters for a notifications/progress notification.

▼ **Example: Progress message**

```
{
  "progressToken": "oivaizmir",
  "progress": 50,
  "total": 100,
  "message": "Reticulating splines..."
} Copy
```

> **_meta?: NotificationMetaObject**
> 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

```
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 is only delivered on a subscriptions/listen stream when the client requested it via the `promptsListChanged` filter field.

▼ **Example: Prompts list changed**

```
{
  "jsonrpc": "2.0",
  "method": "notifications/prompts/list_changed",
  "params": {
    "_meta": {
      "io.modelcontextprotocol/subscriptionId": "listen-1"
    }
  }
} Copy
```

> **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 is only delivered on a subscriptions/listen stream when the client requested it via the `resourcesListChanged` filter field.

▼ **Example: Resources list changed**

```
{
  "jsonrpc": "2.0",
  "method": "notifications/resources/list_changed",
  "params": {
    "_meta": {
      "io.modelcontextprotocol/subscriptionId": "listen-1"
    }
  }
} Copy
```

> **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 is only sent for resources the client opted in to via the `resourceSubscriptions` field of a subscriptions/listen request.

▼ **Example: File resource updated notification**

```
{
  "jsonrpc": "2.0",
  "method": "notifications/resources/updated",
  "params": {
    "_meta": {
      "io.modelcontextprotocol/subscriptionId": "listen-1"
    },
    "uri": "file:///project/src/main.rs"
  }
} Copy
```

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

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

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

ResourceUpdatedNotificationParams

```
interface ResourceUpdatedNotificationParams {
  _meta?: NotificationMetaObject;
  uri: string;
}
```

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

▼ **Example: File resource updated**

```
{
  "uri": "file:///project/src/main.rs"
}
```
Copy

> **_meta?: NotificationMetaObject**
> 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/subscriptions/acknowledged

SubscriptionsAcknowledgedNotification

```
interface SubscriptionsAcknowledgedNotification {
  jsonrpc: "2.0";
  method: "notifications/subscriptions/acknowledged";
  params: SubscriptionsAcknowledgedNotificationParams;
}
```

Sent by the server to acknowledge that a subscriptions/listen subscription has been established and to report which notification types it agreed to honor.

This notification MUST be the first message the server sends carrying the subscription's ID in `io.modelcontextprotocol/subscriptionId`. The server MUST NOT send any notification on the subscription before acknowledging it. On stdio, where every subscription shares one channel, this ordering is defined per subscription ID and not per channel: messages belonging to other subscriptions MAY be interleaved before it.

▼ **Example: Listen acknowledged**

```
{
  "jsonrpc": "2.0",
  "method": "notifications/subscriptions/acknowledged",
  "params": {
    "_meta": {
      "io.modelcontextprotocol/subscriptionId": "listen-1"
    },
    "notifications": {
      "toolsListChanged": true,
      "resourceSubscriptions": ["file:///project/config.json"]
    }
  }
} Copy
```

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

> **method: "notifications/subscriptions/acknowledged"**
> Overrides JSONRPCNotification.method

> **params: SubscriptionsAcknowledgedNotificationParams**
> Overrides JSONRPCNotification.params

SubscriptionsAcknowledgedNotificationParams

```
interface SubscriptionsAcknowledgedNotificationParams {
  _meta?: NotificationMetaObject;
  notifications: SubscriptionFilter;
}
```

Parameters for a notifications/subscriptions/acknowledged notification.

> **_meta?: NotificationMetaObject**
> Inherited from NotificationParams._meta

> **notifications: SubscriptionFilter**
>
> The subset of requested notification types the server agreed to honor. Only includes notification types the server actually supports; if the client requested an unsupported type (e.g., `promptsListChanged` when the server has no prompts), it is omitted from this set.

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 is only delivered on a subscriptions/listen stream when the client requested it via the `toolsListChanged` filter field.

▼ **Example: Tools list changed**

```
{
  "jsonrpc": "2.0",
  "method": "notifications/tools/list_changed",
  "params": {
    "_meta": {
      "io.modelcontextprotocol/subscriptionId": "listen-1"
    }
  }
} Copy
```

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

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

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

Multi Round-Trip

`InputRequests`

```
InputRequests: any
```

A map of server-initiated requests that the client must fulfill. Keys are server-assigned identifiers; values are the request objects.

▼ **Example: Elicitation and sampling input requests**

```
{
  "github_login": {
    "method": "elicitation/create",
    "params": {
      "mode": "form",
      "message": "Please provide your GitHub username",
      "requestedSchema": {
        "type": "object",
        "properties": {
          "name": {
            "type": "string"
          }
        },
        "required": ["name"]
      }
    }
  },
  "capital_of_france": {
    "method": "sampling/createMessage",
    "params": {
      "messages": [
        {
          "role": "user",
          "content": {
            "type": "text",
            "text": "What is the capital of France?"
          }
        }
      ],
      "maxTokens": 100
    }
  }
} Copy
```

`InputRequiredResult`

```
interface InputRequiredResult {
  _meta?: ResultMetaObject;
  resultType: string;
  inputRequests?: InputRequests;
  requestState?: string;
  [key: string]: unknown;
}
```

An InputRequiredResult sent by the server to indicate that additional input is needed before the request can be completed.

At least one of `inputRequests` or `requestState` MUST be present.

▼ Example: InputRequiredResult with elicitation and sampling input requests and request state

```
{
  "resultType": "input_required",
  "inputRequests": {
    "github_login": {
      "method": "elicitation/create",
      "params": {
        "message": "Please provide your GitHub username",
        "requestedSchema": {
          "type": "object",
          "properties": {
            "name": {
              "type": "string"
            }
          },
          "required": ["name"]
        }
      }
    },
    "capital_of_france": {
      "method": "sampling/createMessage",
      "params": {
        "messages": [
          {
            "role": "user",
            "content": {
              "type": "text",
              "text": "What is the capital of France?"
            }
          }
        ],
        "maxTokens": 100
      }
    }
  },
  "requestState": "eyJsb2NhdGlvbiI6Ik5ldyBZb3JrIn0"
} Copy
```

▼ **Example: InputRequiredResult with request state only (load shedding)**

```
{
  "resultType": "input_required",
  "requestState": "eyJwcm9ncmVzcyI6IjUwJSIsInN0YXRlIjoicHJvY2Vzc2luZyJ9"
} Copy
```

> **_meta?: ResultMetaObject**
>
> Inherited from Result._meta

> **resultType: string**
>
> Indicates the type of the result, which allows the client to determine how to parse the result object.
>
> Servers implementing this protocol version MUST include this field. For backward compatibility, when a client receives a result from a server implementing an earlier protocol version (which does not include

> `resultType`), the client MUST treat the absent field as `"complete"`.
>
> Inherited from Result.resultType

> **inputRequests?: InputRequests**

> **requestState?: string**

`InputResponses`

```
InputResponses: any
```

A map of client responses to server-initiated requests. Keys correspond to the keys in the InputRequests map; values are the client's result for each request.

▼ **Example: Elicitation and sampling input responses**

```
{
  "github_login": {
    "action": "accept",
    "content": {
      "name": "octocat"
    }
  },
  "capital_of_france": {
    "role": "assistant",
    "content": {
      "type": "text",
      "text": "The capital of France is Paris."
    },
    "model": "claude-3-sonnet-20240307",
    "stopReason": "endTurn"
  }
} Copy
```

`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.

▼ **Example: Get prompt request**

```
{
  "jsonrpc": "2.0",
  "id": "get-prompt-example",
  "method": "prompts/get",
  "params": {
    "_meta": {
      "io.modelcontextprotocol/protocolVersion": "2026-07-28",
      "io.modelcontextprotocol/clientInfo": {
        "name": "ExampleClient",
        "version": "1.0.0"
      },
      "io.modelcontextprotocol/clientCapabilities": {}
    },
    "name": "code_review",
    "arguments": {
      "code": "def hello():\n    print('world')"
    }
  }
}
```
Copy

> **`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: RequestMetaObject;
  inputResponses?: InputResponses;
  requestState?: string;
  name: string;
  arguments?: { [key: string]: string };
}
```

Parameters for a `prompts/get` request.

▼ **Example: Get code review prompt**

```
{
  "_meta": {
    "io.modelcontextprotocol/protocolVersion": "2026-07-28",
    "io.modelcontextprotocol/clientInfo": {
      "name": "ExampleClient",
      "version": "1.0.0"
    },
    "io.modelcontextprotocol/clientCapabilities": {}
  },
  "name": "code_review",
  "arguments": {
    "code": "def hello():\n    print('world')"
  }
} Copy
```

> **_meta: RequestMetaObject**
> Inherited from InputResponseRequestParams._meta

> **inputResponses?: InputResponses**
> Inherited from InputResponseRequestParams.inputResponses

> **requestState?: string**
> Inherited from InputResponseRequestParams.requestState

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

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

GetPromptResultResponse

```
interface GetPromptResultResponse {
  jsonrpc: "2.0";
  id: RequestId;
  result: InputRequiredResult | GetPromptResult;
}
```

A successful response from the server for a prompts/get request.

▼ **Example: Get prompt result response**

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

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

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

> **result: InputRequiredResult | GetPromptResult**
> Overrides JSONRPCResultResponse.result

`GetPromptResult`

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

The result returned by the server for a prompts/get request.

▼ **Example: Code review prompt**

```
{
  "resultType": "complete",
  "description": "Code review prompt",
  "messages": [
    {
      "role": "user",
      "content": {
        "type": "text",
        "text": "Please review this Python code:\ndef hello():\n    print('world')"
      }
    }
  ]
}
```
Copy

> **_meta?: ResultMetaObject**
> Inherited from Result._meta

> **resultType: string**
>
> Indicates the type of the result, which allows the client to determine how to parse the result object.
>
> Servers implementing this protocol version MUST include this field. For backward compatibility, when a client receives a result from a server implementing an earlier protocol version (which does not include `resultType`), the client MUST treat the absent field as `"complete"`.
>
> Inherited from Result.resultType

> **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.

▼ **Example: List prompts request**

```
{
  "jsonrpc": "2.0",
  "id": "list-prompts-example",
  "method": "prompts/list",
  "params": {
    "_meta": {
      "io.modelcontextprotocol/protocolVersion": "2026-07-28",
      "io.modelcontextprotocol/clientInfo": {
        "name": "ExampleClient",
        "version": "1.0.0"
      },
      "io.modelcontextprotocol/clientCapabilities": {}
    }
  }
} Copy
```

> **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

`ListPromptsResultResponse`

```
interface ListPromptsResultResponse {
  jsonrpc: "2.0";
  id: RequestId;
  result: ListPromptsResult;
}
```

A successful response from the server for a prompts/list request.

▼ **Example: List prompts result response**

```
{
  "jsonrpc": "2.0",
  "id": "list-prompts-example",
  "result": {
    "resultType": "complete",
    "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",
    "ttlMs": 600000,
    "cacheScope": "public"
  }
}
```
Copy

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

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

> **result: ListPromptsResult**
> Overrides JSONRPCResultResponse.result

`ListPromptsResult`

```ts
interface ListPromptsResult {
  _meta?: ResultMetaObject;
  resultType: string;
  nextCursor?: string;
  ttlMs: number;
  cacheScope: "public" | "private";
  prompts: Prompt[];
  [key: string]: unknown;
}
```

The result returned by the server for a prompts/list request.

▼ **Example: Prompts list with cursor and TTL**

```
{
  "resultType": "complete",
  "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",
  "ttlMs": 600000,
  "cacheScope": "public"
} Copy
```

> **_meta?: ResultMetaObject**
>
> Inherited from PaginatedResult._meta

> **resultType: string**
>
> Indicates the type of the result, which allows the client to determine how to parse the result object.
>
> Servers implementing this protocol version MUST include this field. For backward compatibility, when a client receives a result from a server implementing an earlier protocol version (which does not include `resultType`), the client MUST treat the absent field as `"complete"`.

Inherited from PaginatedResult.resultType

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

ttlMs: number

A hint from the server indicating how long (in milliseconds) the client MAY cache this response before re-fetching. Semantics are analogous to HTTP Cache-Control max-age.

- If 0, The response SHOULD be considered immediately stale, The client MAY re-fetch every time the result is needed.
- If positive, the client SHOULD consider the result fresh for this many milliseconds after receiving the response.

Inherited from CacheableResult.ttlMs

cacheScope: "public" | "private"

Indicates the intended scope of the cached response, analogous to HTTP `Cache-Control: public` vs `Cache-Control: private`.

- `"public"`: The response does not contain user-specific data. Any client or intermediary (e.g., shared gateway, caching proxy) MAY cache the response and serve it across authorization contexts.
- `"private"`: The response MAY be cached and reused only within the same authorization context. Caches MUST NOT be shared across authorization contexts (e.g., a different access token requires a different cache).

Inherited from CacheableResult.cacheScope

prompts: Prompt[]

`Prompt`

```
interface Prompt {
  icons?: Icon[];
  name: string;
  title?: string;
  description?: string;
  arguments?: PromptArgument[];
  _meta?: MetaObject;
}
```

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?: MetaObject`

`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.

▼ **Example: List resources request**

```
{
  "jsonrpc": "2.0",
  "id": "list-resources-example",
  "method": "resources/list",
  "params": {
    "_meta": {
      "io.modelcontextprotocol/protocolVersion": "2026-07-28",
      "io.modelcontextprotocol/clientInfo": {
        "name": "ExampleClient",
        "version": "1.0.0"
      },
      "io.modelcontextprotocol/clientCapabilities": {}
    }
  }
} Copy
```

> **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

ListResourcesResultResponse

```
interface ListResourcesResultResponse {
  jsonrpc: "2.0";
  id: RequestId;
  result: ListResourcesResult;
}
```

A successful response from the server for a resources/list request.

▼ **Example: List resources result response**

```
{
  "jsonrpc": "2.0",
  "id": "list-resources-example",
  "result": {
    "resultType": "complete",
    "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": "eyJwYWdlIjogM30=",
    "ttlMs": 600000,
    "cacheScope": "private"
  }
}
```

Copy

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

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

> **result: ListResourcesResult**
> Overrides JSONRPCResultResponse.result

`ListResourcesResult`

```ts
interface ListResourcesResult {
  _meta?: ResultMetaObject;
  resultType: string;
  nextCursor?: string;
  ttlMs: number;
  cacheScope: "public" | "private";
  resources: Resource[];
  [key: string]: unknown;
}
```

The result returned by the server for a resources/list request.

▼ **Example: Resources list with cursor and TTL**

```ts
{
  "resultType": "complete",
  "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": "eyJwYWdlIjogM30=",
  "ttlMs": 600000,
  "cacheScope": "private"
} Copy
```

> **_meta?: ResultMetaObject**
>
> Inherited from PaginatedResult._meta

> **resultType: string**
>
> Indicates the type of the result, which allows the client to determine how to parse the result object.
>
> Servers implementing this protocol version MUST include this field. For backward compatibility, when a client receives a result from a server implementing an earlier protocol version (which does not include `resultType`), the client MUST treat the absent field as `"complete"`.
>
> Inherited from PaginatedResult.resultType

> **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

> **`ttlMs: number`**
>
> A hint from the server indicating how long (in milliseconds) the client MAY cache this response before re-fetching. Semantics are analogous to HTTP Cache-Control max-age.
>
> - If 0, The response SHOULD be considered immediately stale, The client MAY re-fetch every time the result is needed.
> - If positive, the client SHOULD consider the result fresh for this many milliseconds after receiving the response.
>
> Inherited from CacheableResult.ttlMs

> **`cacheScope: "public" | "private"`**
>
> Indicates the intended scope of the cached response, analogous to HTTP `Cache-Control: public` vs `Cache-Control: private`.
>
> - `"public"`: The response does not contain user-specific data. Any client or intermediary (e.g., shared gateway, caching proxy) MAY cache the response and serve it across authorization contexts.
> - `"private"`: The response MAY be cached and reused only within the same authorization context. Caches MUST NOT be shared across authorization contexts (e.g., a different access token requires a different cache).
>
> Inherited from CacheableResult.cacheScope

> **`resources: Resource[]`**

`Resource`

```ts
interface Resource {
  icons?: Icon[];
  name: string;
  title?: string;
  uri: string;
  description?: string;
  mimeType?: string;
  annotations?: Annotations;
  size?: number;
  _meta?: MetaObject;
}
```

A known resource that the server is capable of reading.

▼ **Example: File 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"
  }
}
```
Copy

> **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?: MetaObject**

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.

▼ **Example: Read resource request**

```
{
  "jsonrpc": "2.0",
  "id": "read-resource-example",
  "method": "resources/read",
  "params": {
    "_meta": {
      "io.modelcontextprotocol/protocolVersion": "2026-07-28",
      "io.modelcontextprotocol/clientInfo": {
        "name": "ExampleClient",
        "version": "1.0.0"
      },
      "io.modelcontextprotocol/clientCapabilities": {}
    },
    "uri": "file:///project/src/main.rs"
  }
}
```

Copy

> **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: RequestMetaObject;
  inputResponses?: InputResponses;
  requestState?: string;
  uri: string;
}
```

Parameters for a `resources/read` request.

> **_meta: RequestMetaObject**
> Inherited from InputResponseRequestParams._meta

> **inputResponses?: InputResponses**
> Inherited from InputResponseRequestParams.inputResponses

> **requestState?: string**
> Inherited from InputResponseRequestParams.requestState

> **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

ReadResourceResultResponse

```
interface ReadResourceResultResponse {
  jsonrpc: "2.0";
  id: RequestId;
  result: InputRequiredResult | ReadResourceResult;
}
```

A successful response from the server for a resources/read request.

▼ **Example: Read resource result response**

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

Copy

▼ **Example: Read resource result response with TTL**

```
{
  "jsonrpc": "2.0",
  "id": "read-resource-with-ttl-example",
  "result": {
    "resultType": "complete",
    "contents": [
      {
        "uri": "file:///project/src/main.rs",
        "mimeType": "text/x-rust",
        "text": "fn main() {\n    println!(\"Hello world!\");\n}"
      }
    ],
    "ttlMs": 60000,
    "cacheScope": "private"
  }
} Copy
```

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

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

> **result: InputRequiredResult | ReadResourceResult**
> Overrides JSONRPCResultResponse.result

`ReadResourceResult`

```
interface ReadResourceResult {
  _meta?: ResultMetaObject;
  resultType: string;
  ttlMs: number;
  cacheScope: "public" | "private";
  contents: (TextResourceContents | BlobResourceContents)[];
  [key: string]: unknown;
}
```

The result returned by the server for a resources/read request.

▼ **Example: File resource contents**

```
{
  "resultType": "complete",
  "contents": [
    {
      "uri": "file:///project/src/main.rs",
      "mimeType": "text/x-rust",
      "text": "fn main() {\n    println!(\"Hello world!\");\n}"
    }
  ],
  "ttlMs": 60000,
  "cacheScope": "private"
}
```
Copy

`_meta?: ResultMetaObject`

Inherited from CacheableResult._meta

`resultType: string`

Indicates the type of the result, which allows the client to determine how to parse the result object.

Servers implementing this protocol version MUST include this field. For backward compatibility, when a client receives a result from a server implementing an earlier protocol version (which does not include `resultType`), the client MUST treat the absent field as `"complete"`.

Inherited from CacheableResult.resultType

`ttlMs: number`

A hint from the server indicating how long (in milliseconds) the client MAY cache this response before re-fetching. Semantics are analogous to HTTP Cache-Control max-age.

- If 0, The response SHOULD be considered immediately stale, The client MAY re-fetch every time the result is needed.
- If positive, the client SHOULD consider the result fresh for this many milliseconds after receiving the response.

Inherited from CacheableResult.ttlMs

`cacheScope: "public" | "private"`

Indicates the intended scope of the cached response, analogous to HTTP `Cache-Control: public` vs `Cache-Control: private`.

- `"public"`: The response does not contain user-specific data. Any client or intermediary (e.g., shared gateway, caching proxy) MAY cache the response and serve it across authorization contexts.
- `"private"`: The response MAY be cached and reused only within the same authorization context. Caches MUST NOT be shared across authorization contexts (e.g., a different access token requires a different cache).

Inherited from CacheableResult.cacheScope

`contents: (TextResourceContents | BlobResourceContents)[]`

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.

▼ **Example: List resource templates request**

```
{
  "jsonrpc": "2.0",
  "id": "list-resource-templates-example",
  "method": "resources/templates/list",
  "params": {
    "_meta": {
      "io.modelcontextprotocol/protocolVersion": "2026-07-28",
      "io.modelcontextprotocol/clientInfo": {
        "name": "ExampleClient",
        "version": "1.0.0"
      },
      "io.modelcontextprotocol/clientCapabilities": {}
    }
  }
} Copy
```

> **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

ListResourceTemplatesResultResponse

```
interface ListResourceTemplatesResultResponse {
  jsonrpc: "2.0";
  id: RequestId;
  result: ListResourceTemplatesResult;
}
```

A successful response from the server for a resources/templates/list request.

▼ **Example: List resource templates result response**

```
{
  "jsonrpc": "2.0",
  "id": "list-resource-templates-example",
  "result": {
    "resultType": "complete",
    "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"]
          }
        ]
      }
    ],
    "ttlMs": 3600000,
    "cacheScope": "public"
  }
} Copy
```

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

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

> **result: ListResourceTemplatesResult**
> Overrides JSONRPCResultResponse.result

`ListResourceTemplatesResult`

```
interface ListResourceTemplatesResult {
  _meta?: ResultMetaObject;
  resultType: string;
  nextCursor?: string;
  ttlMs: number;
  cacheScope: "public" | "private";
  resourceTemplates: ResourceTemplate[];
  [key: string]: unknown;
}
```

The result returned by the server for a resources/templates/list request.

▼ **Example: Resource templates list with cursor and TTL**

```
{
  "resultType": "complete",
  "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"]
        }
      ]
    }
  ],
  "nextCursor": "next-page-cursor",
  "ttlMs": 3600000,
  "cacheScope": "public"
}
```

Copy

_meta?: ResultMetaObject

Inherited from PaginatedResult._meta

resultType: string

Indicates the type of the result, which allows the client to determine how to parse the result object.

Servers implementing this protocol version MUST include this field. For backward compatibility, when a client receives a result from a server implementing an earlier protocol version (which does not include `resultType`), the client MUST treat the absent field as `"complete"`.

Inherited from PaginatedResult.resultType

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

ttlMs: number

A hint from the server indicating how long (in milliseconds) the client MAY cache this response before re-fetching. Semantics are analogous to HTTP Cache-Control max-age.

- If 0, The response SHOULD be considered immediately stale, The client MAY re-fetch every time the result is needed.
- If positive, the client SHOULD consider the result fresh for this many milliseconds after receiving the response.

Inherited from CacheableResult.ttlMs

cacheScope: "public" | "private"

> Indicates the intended scope of the cached response, analogous to HTTP `Cache-Control: public` vs `Cache-Control: private`.
>
> - `"public"`: The response does not contain user-specific data. Any client or intermediary (e.g., shared gateway, caching proxy) MAY cache the response and serve it across authorization contexts.
> - `"private"`: The response MAY be cached and reused only within the same authorization context. Caches MUST NOT be shared across authorization contexts (e.g., a different access token requires a different cache).
>
> Inherited from CacheableResult.cacheScope

> **`resourceTemplates: ResourceTemplate[]`**

`ResourceTemplate`

```
interface ResourceTemplate {
  icons?: Icon[];
  name: string;
  title?: string;
  uriTemplate: string;
  description?: string;
  mimeType?: string;
  annotations?: Annotations;
  _meta?: MetaObject;
}
```

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?: MetaObject**

roots/list

ListRootsRequest

```
interface ListRootsRequest {
  method: "roots/list";
  params?: { _meta?: MetaObject };
}
```

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.

▼ **Example: List roots request**

```
{
  "id": "list-roots-example",
  "method": "roots/list"
} Copy
```

Deprecated

Deprecated as of protocol version 2026-07-28 (SEP-2577). Remains in the specification for at least twelve months; see the deprecated features registry.

> **method: "roots/list"**

> **params?: { _meta?: MetaObject }**

`ListRootsResult`

```
interface ListRootsResult {
  roots: Root[];
}
```

The result returned by the client for a roots/list request. This result contains an array of Root objects, each representing a root directory or file that the server can operate on.

▼ **Example: Single root directory**

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

▼ **Example: Multiple root directories**

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

Deprecated

Deprecated as of protocol version 2026-07-28 (SEP-2577). Remains in the specification for at least twelve months; see the deprecated features registry.

> **`roots: Root[]`**

`Root`

```
interface Root {
  uri: string;
  name?: string;
  _meta?: MetaObject;
}
```

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

▼ **Example: Project directory root**

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

Deprecated

Deprecated as of protocol version 2026-07-28 (SEP-2577). Remains in the specification for at least twelve months; see the deprecated features registry.

> **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?: MetaObject**

sampling/createMessage

CreateMessageRequest

```
interface CreateMessageRequest {
  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.

▼ **Example: Sampling request**

```
{
  "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
  }
} Copy
```

Deprecated

Deprecated as of protocol version 2026-07-28 (SEP-2577). Remains in the specification for at least twelve months; see the deprecated features registry.

> **method: "sampling/createMessage"**

> **params: CreateMessageRequestParams**

`CreateMessageRequestParams`

```
interface CreateMessageRequestParams {
  messages: SamplingMessage[];
  modelPreferences?: ModelPreferences;
  systemPrompt?: string;
  includeContext?: "none" | "thisServer" | "allServers";
  temperature?: number;
  maxTokens: number;
  stopSequences?: string[];
  metadata?: JSONObject;
  tools?: Tool[];
  toolChoice?: ToolChoice;
}
```

Parameters for a `sampling/createMessage` request.

▼ **Example: Basic request**

```
{
  "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
}
```

Copy

▼ **Example: Request with tools**

```
{
  "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
}
```

Copy

▼ **Example: Follow-up request with tool results**

```
{
  "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
}
```
Copy

Deprecated

Deprecated as of protocol version 2026-07-28 (SEP-2577). Remains in the specification for at least twelve months; see the deprecated features registry.

> **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"`. The values `"thisServer"` and `"allServers"` are deprecated (SEP-2596): servers SHOULD omit this field or use `"none"`, and SHOULD only use the deprecated values if the client declares ClientCapabilities.sampling.context.
>
> **Deprecated**
>
> The `"thisServer"` and `"allServers"` values are deprecated as of protocol version 2025-11-25 (SEP-2596) and will be removed no later than the Sampling feature itself (SEP-2577). Omit this field or use `"none"`.

> **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?: JSONObject**
>
> 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`

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

The result returned by the client for a sampling/createMessage request. 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.

▼ **Example: Text response**

```
{
  "role": "assistant",
  "content": {
    "type": "text",
    "text": "The capital of France is Paris."
  },
  "model": "claude-3-sonnet-20240307",
  "stopReason": "endTurn"
} Copy
```

▼ **Example: Tool use response**

```
{
  "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"
} Copy
```

▼ **Example: Final response after tool use**

```
{
  "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"
} Copy
```

Deprecated

Deprecated as of protocol version 2026-07-28 (SEP-2577). Remains in the specification for at least twelve months; see the deprecated features registry.

> **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

> **`_meta?: MetaObject`**
> Inherited from SamplingMessage._meta

`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.

Deprecated

Deprecated as of protocol version 2026-07-28 (SEP-2577). Remains in the specification for at least twelve months; see the deprecated features registry.

> **`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.

▼ **Example: With hints and priorities**

```
{
  "hints": [
    { "name": "claude-3-sonnet" },
    { "name": "claude" }
  ],
  "costPriority": 0.3,
  "speedPriority": 0.8,
  "intelligencePriority": 0.5
} Copy
```

Deprecated

Deprecated as of protocol version 2026-07-28 (SEP-2577). Remains in the specification for at least twelve months; see the deprecated features registry.

> **`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`

```ts
interface SamplingMessage {
  role: Role;
  content: SamplingMessageContentBlock | SamplingMessageContentBlock[];
  _meta?: MetaObject;
}
```

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

▼ **Example: Single content block**

```
{
  "role": "user",
  "content": {
    "type": "text",
    "text": "What is the capital of France?"
  }
} Copy
```

▼ **Example: Multiple content blocks**

```
{
  "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" }]
    }
  ]
} Copy
```

Deprecated

Deprecated as of protocol version 2026-07-28 (SEP-2577). Remains in the specification for at least twelve months; see the deprecated features registry.

> **role: Role**

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

> **_meta?: MetaObject**

`SamplingMessageContentBlock`

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

Deprecated

Deprecated as of protocol version 2026-07-28 (SEP-2577). Remains in the specification for at least twelve months; see the deprecated features registry.

`ToolChoice`

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

Controls tool selection behavior for sampling requests.

Deprecated

Deprecated as of protocol version 2026-07-28 (SEP-2577). Remains in the specification for at least twelve months; see the deprecated features registry.

> **`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`

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

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

▼ **Example: `get_weather` tool result**

```
{
  "type": "tool_result",
  "toolUseId": "call_abc123",
  "content": [
    {
      "type": "text",
      "text": "Weather in Paris: 18°C, partly cloudy"
    }
  ]
}
```
Copy

Deprecated

Deprecated as of protocol version 2026-07-28 (SEP-2577). Remains in the specification for at least twelve months; see the deprecated features registry.

> **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?: unknown**
>
> An optional structured result value.
>
> This can be any JSON value (object, array, string, number, boolean, or null). If the tool defined an Tool.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?: MetaObject**
>
> Optional metadata about the tool result. Clients SHOULD preserve this field when including tool results in subsequent sampling requests to enable caching optimizations.

`ToolUseContent`

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

A request from the assistant to call a tool.

▼ **Example: `get_weather` tool use**

```
{
  "type": "tool_use",
  "id": "call_abc123",
  "name": "get_weather",
  "input": {
    "city": "Paris"
  }
} Copy
```

Deprecated

Deprecated as of protocol version 2026-07-28 (SEP-2577). Remains in the specification for at least twelve months; see the deprecated features registry.

> **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?: MetaObject**
>
> Optional metadata about the tool use. Clients SHOULD preserve this field when including tool uses in subsequent sampling requests to enable caching optimizations.

server/discover

DiscoverRequest

```
interface DiscoverRequest {
  jsonrpc: "2.0";
  id: RequestId;
  method: "server/discover";
  params: RequestParams;
}
```

A request from the client asking the server to advertise its supported protocol versions, capabilities, and other metadata. Servers **MUST** implement `server/discover`. Clients **MAY** call it but are not required to — version negotiation can also happen inline via per-request `_meta`.

▼ **Example: Discover request**

```
{
  "jsonrpc": "2.0",
  "id": "discover-1",
  "method": "server/discover",
  "params": {
    "_meta": {
      "io.modelcontextprotocol/protocolVersion": "2026-07-28",
      "io.modelcontextprotocol/clientInfo": {
        "name": "ExampleClient",
        "version": "1.0.0"
      },
      "io.modelcontextprotocol/clientCapabilities": {}
    }
  }
} Copy
```

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

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

> **method: "server/discover"**
> Overrides JSONRPCRequest.method

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

`DiscoverResultResponse`

```
interface DiscoverResultResponse {
  jsonrpc: "2.0";
  id: RequestId;
  result: DiscoverResult;
}
```

A successful response from the server for a server/discover request.

▼ **Example: Discover result response**

```
{
  "jsonrpc": "2.0",
  "id": "discover-1",
  "result": {
    "resultType": "complete",
    "supportedVersions": ["2026-07-28"],
    "capabilities": {
      "tools": {},
      "resources": {}
    },
    "_meta": {
      "io.modelcontextprotocol/serverInfo": {
        "name": "ExampleServer",
        "version": "1.0.0"
      }
    },
    "ttlMs": 3600000,
    "cacheScope": "public"
  }
}
```
Copy

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

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

> **result: DiscoverResult**
> Overrides JSONRPCResultResponse.result

`DiscoverResult`

```
interface DiscoverResult {
  _meta?: ResultMetaObject;
  resultType: string;
  supportedVersions: string[];
  capabilities: ServerCapabilities;
  instructions?: string;
  ttlMs: number;
  cacheScope: "public" | "private";
  [key: string]: unknown;
}
```

The result returned by the server for a server/discover request.

▼ **Example: Server capabilities discovery**

```
{
  "resultType": "complete",
  "supportedVersions": ["2026-07-28"],
  "capabilities": {
    "tools": {},
    "resources": {}
  },
  "_meta": {
    "io.modelcontextprotocol/serverInfo": {
      "name": "ExampleServer",
      "version": "1.0.0"
    }
  },
  "instructions": "This server provides weather and resource utilities. Prefer
`get_weather` for forecast lookups.",
  "ttlMs": 3600000,
  "cacheScope": "public"
}
```
Copy

`_meta?: ResultMetaObject`

Inherited from CacheableResult._meta

`resultType: string`

Indicates the type of the result, which allows the client to determine how to parse the result object.

Servers implementing this protocol version MUST include this field. For backward compatibility, when a client receives a result from a server implementing an earlier protocol version (which does not include `resultType`), the client MUST treat the absent field as `"complete"`.

Inherited from CacheableResult.resultType

`supportedVersions: string[]`

MCP Protocol Versions this server supports. The client should choose a version from this list for use in subsequent requests.

`capabilities: ServerCapabilities`

The capabilities of the server.

`instructions?: string`

Natural-language guidance describing the server and its features.

This can be used by clients to improve an LLM's understanding of available tools (e.g., by including it in a system prompt). It should focus on information that helps the model use the server effectively and should not duplicate information already in tool descriptions.

`ttlMs: number`

A hint from the server indicating how long (in milliseconds) the client MAY cache this response before re-fetching. Semantics are analogous to HTTP Cache-Control max-age.

- If 0, The response SHOULD be considered immediately stale, The client MAY re-fetch every time the result is needed.
- If positive, the client SHOULD consider the result fresh for this many milliseconds after receiving the response.

Inherited from CacheableResult.ttlMs

cacheScope: "public" | "private"

Indicates the intended scope of the cached response, analogous to HTTP `Cache-Control: public` vs `Cache-Control: private`.

- `"public"`: The response does not contain user-specific data. Any client or intermediary (e.g., shared gateway, caching proxy) MAY cache the response and serve it across authorization contexts.
- `"private"`: The response MAY be cached and reused only within the same authorization context. Caches MUST NOT be shared across authorization contexts (e.g., a different access token requires a different cache).

Inherited from CacheableResult.cacheScope

ClientCapabilities

```
interface ClientCapabilities {
  experimental?: { [key: string]: JSONObject };
  roots?: {};
  sampling?: { context?: JSONObject; tools?: JSONObject };
  elicitation?: { form?: JSONObject; url?: JSONObject };
  extensions?: { [key: string]: JSONObject };
}
```

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]: JSONObject }

Experimental, non-standard capabilities that the client supports.

roots?: {}

Present if the client supports listing roots.

Deprecated

Deprecated as of protocol version 2026-07-28 (SEP-2577). Remains in the specification for at least twelve months; see the deprecated features registry.

▼ **Example: Roots — minimum baseline support**

```
{
  "roots": {}
} Copy
```

sampling?: { context?: JSONObject; tools?: JSONObject }

Present if the client supports sampling from an LLM.

Type Declaration

- `Optional` context?: JSONObject

  Whether the client supports context inclusion via `includeContext` parameter. If not declared, servers SHOULD only use `includeContext: "none"` (or omit it).

- `Optional` tools?: JSONObject

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

Deprecated

Deprecated as of protocol version 2026-07-28 (SEP-2577). Remains in the specification for at least twelve months; see the deprecated features registry.

▼ **Example: Sampling — minimum baseline support**

```
{
  "sampling": {}
} Copy
```

▼ **Example: Sampling — tool use support**

```
{
  "sampling": {
    "tools": {}
  }
} Copy
```

▼ **Example: Sampling — context inclusion support (deprecated)**

```
{
  "sampling": {
    "context": {}
  }
} Copy
```

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

Present if the client supports elicitation from the server.

▼ **Example: Elicitation — form and URL mode support**

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

▼ **Example: Elicitation — form mode only (implicit)**

```
{
  "elicitation": {}
} Copy
```

extensions?: { [key: string]: JSONObject }

> Optional MCP extensions that the client supports. Keys are extension identifiers (e.g., "io.modelcontextprotocol/oauth-client-credentials"), and values are per-extension settings objects. An empty object indicates support with no settings.
>
> Keys MUST follow the `_meta` key naming rules, with a mandatory prefix.
>
> ▼ **Example: Extensions — MCP Apps (UI) extension with MIME type support**

```
{
  "extensions": {
    "io.modelcontextprotocol/ui": {
      "mimeTypes": ["text/html;profile=mcp-app"]
    }
  }
} Copy
```

`Implementation`

```
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`**
>
> The version of this implementation.

> **`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]: JSONObject };
  logging?: JSONObject;
  completions?: JSONObject;
  prompts?: { listChanged?: boolean };
  resources?: { subscribe?: boolean; listChanged?: boolean };
  tools?: { listChanged?: boolean };
  extensions?: { [key: string]: JSONObject };
}
```

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]: JSONObject }`**
>
> Experimental, non-standard capabilities that the server supports.

> **`logging?: JSONObject`**
>
> Present if the server supports sending log messages to the client.
>
> **Deprecated**
>
> Deprecated as of protocol version 2026-07-28 (SEP-2577). Remains in the specification for at least twelve months; see the deprecated features registry.
>
> ▼ **Example: Logging — minimum baseline support**
>
> ```
> {
>   "logging": {}
> }
> ```
> Copy

> **`completions?: JSONObject`**

Present if the server supports argument autocompletion suggestions.

▼ **Example: Completions — minimum baseline support**

```
{
  "completions": {}
}
```
Copy

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.

▼ **Example: Prompts — minimum baseline support**

```
{
  "prompts": {}
}
```
Copy

▼ **Example: Prompts — list changed notifications**

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

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.

▼ **Example: Resources — minimum baseline support**

```
{
  "resources": {}
}
```
Copy

▼ **Example: Resources — subscription to individual resource updates (only)**

```
{
  "resources": {
    "subscribe": true
  }
}
```
Copy

▼ **Example: Resources — list changed notifications (only)**

```
{
  "resources": {
    "listChanged": true
  }
}
```
Copy

▼ **Example: Resources — all notifications**

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

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.

▼ **Example: Tools — minimum baseline support**

```
{
  "tools": {}
}
```
Copy

▼ **Example: Tools — list changed notifications**

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

extensions?: { [key: string]: JSONObject }

Optional MCP extensions that the server supports. Keys are extension identifiers (e.g., "io.modelcontextprotocol/tasks"), and values are per-extension settings objects. An empty object indicates

> support with no settings.
>
> Keys MUST follow the `_meta` key naming rules, with a mandatory prefix.
>
> ▼ **Example: Extensions — Tasks extension support**
>
> ```
> {
>   "extensions": {
>     "io.modelcontextprotocol/tasks": {}
>   }
> } Copy
> ```

`subscriptions/listen`

`SubscriptionsListenRequest`

```
interface SubscriptionsListenRequest {
  jsonrpc: "2.0";
  id: RequestId;
  method: "subscriptions/listen";
  params: SubscriptionsListenRequestParams;
}
```

Sent from the client to open a long-lived channel for receiving notifications outside the context of a specific request. Replaces the previous HTTP GET endpoint and ensures consistent behavior between HTTP and STDIO.

▼ **Example: Listen for tools and resource list changes**

```
{
  "jsonrpc": "2.0",
  "id": "listen-1",
  "method": "subscriptions/listen",
  "params": {
    "_meta": {
      "io.modelcontextprotocol/protocolVersion": "2026-07-28",
      "io.modelcontextprotocol/clientInfo": {
        "name": "ExampleClient",
        "version": "1.0.0"
      },
      "io.modelcontextprotocol/clientCapabilities": {}
    },
    "notifications": {
      "toolsListChanged": true,
      "resourceSubscriptions": ["file:///project/config.json"]
    }
  }
} Copy
```

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

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

> **method: "subscriptions/listen"**
> Overrides JSONRPCRequest.method

> **params: SubscriptionsListenRequestParams**
> Overrides JSONRPCRequest.params

SubscriptionsListenRequestParams

```
interface SubscriptionsListenRequestParams {
  _meta: RequestMetaObject;
  notifications: SubscriptionFilter;
}
```

Parameters for a subscriptions/listen request.

> **_meta: RequestMetaObject**
> Inherited from RequestParams._meta

> **notifications: SubscriptionFilter**
>
> The notifications the client opts in to on this stream. The server **MUST NOT** send notification types the client has not explicitly requested.

SubscriptionsListenResultResponse

```
interface SubscriptionsListenResultResponse {
  jsonrpc: "2.0";
  id: RequestId;
  result: SubscriptionsListenResult;
}
```

A successful response from the server for a subscriptions/listen request, sent when the server tears the subscription down gracefully.

▼ **Example: Subscription closed gracefully response**

```
{
  "jsonrpc": "2.0",
  "id": "listen-1",
  "result": {
    "resultType": "complete",
    "_meta": {
      "io.modelcontextprotocol/subscriptionId": "listen-1"
    }
  }
} Copy
```

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

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

> **result: SubscriptionsListenResult**
> Overrides JSONRPCResultResponse.result

SubscriptionsListenResult

```
interface SubscriptionsListenResult {
  resultType: string;
  _meta: SubscriptionsListenResultMetaObject;
  [key: string]: unknown;
}
```

The response to a subscriptions/listen request, signalling that the subscription has ended gracefully (for example, during server shutdown). Because the listen stream is long-lived, this result is sent only when the server tears the subscription down; an abrupt transport close carries no response. The result body is otherwise empty.

▼ **Example: Subscription closed gracefully**

```
{
  "resultType": "complete",
  "_meta": {
    "io.modelcontextprotocol/subscriptionId": "listen-1"
  }
} Copy
```

> **resultType: string**
>
> Indicates the type of the result, which allows the client to determine how to parse the result object.
>
> Servers implementing this protocol version MUST include this field. For backward compatibility, when a client receives a result from a server implementing an earlier protocol version (which does not include `resultType`), the client MUST treat the absent field as `"complete"`.
>
> Inherited from Result.resultType

> **_meta: SubscriptionsListenResultMetaObject**
> Overrides Result._meta

SubscriptionFilter

```
interface SubscriptionFilter {
  toolsListChanged?: boolean;
  promptsListChanged?: boolean;
  resourcesListChanged?: boolean;
  resourceSubscriptions?: string[];
}
```

The set of notification types a client may opt in to on a subscriptions/listen request.

Each notification type is **opt-in**; the server **MUST NOT** send notification types the client has not explicitly requested here.

> **`toolsListChanged?: boolean`**
>
> If true, receive notifications/tools/list_changed.

> **`promptsListChanged?: boolean`**
>
> If true, receive notifications/prompts/list_changed.

> **`resourcesListChanged?: boolean`**
>
> If true, receive notifications/resources/list_changed.

> **`resourceSubscriptions?: string[]`**
>
> Subscribe to notifications/resources/updated for these resource URIs. Replaces the former `resources/subscribe` RPC.

`SubscriptionsListenResultMetaObject`

```
interface SubscriptionsListenResultMetaObject {
  "io.modelcontextprotocol/serverInfo"?: Implementation;
  "io.modelcontextprotocol/subscriptionId": RequestId;
  [key: string]: unknown;
}
```

Extends ResultMetaObject with the subscription-stream identifier carried by a SubscriptionsListenResult. All key naming rules from `MetaObject` apply.

See

MetaObject for key naming rules and reserved prefixes.

> **`"io.modelcontextprotocol/serverInfo"?: Implementation`**
>
> Identifies the server software producing the response. Servers SHOULD include this field on every response unless specifically configured not to do so.
>
> The Implementation schema requires `name` and `version`; other fields are optional.
>
> The value is self-reported by the server and is not verified by the protocol. It is intended for display, logging, and debugging. Clients SHOULD NOT use it to change their behavior, and SHOULD NOT rely on it for security decisions.
>
> Inherited from ResultMetaObject.io.modelcontextprotocol/serverInfo

> **`"io.modelcontextprotocol/subscriptionId": RequestId`**
>
> Identifies the subscription stream this response closes, so the client can correlate it with the originating subscription — mirroring the same key on the stream's notifications. The value is the JSON-RPC ID of the `subscriptions/listen` request that opened the stream (and equals this response's `id`).

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.

▼ **Example: Call tool request**

```
{
  "jsonrpc": "2.0",
  "id": "call-tool-example",
  "method": "tools/call",
  "params": {
    "_meta": {
      "io.modelcontextprotocol/protocolVersion": "2026-07-28",
      "io.modelcontextprotocol/clientInfo": {
        "name": "ExampleClient",
        "version": "1.0.0"
      },
      "io.modelcontextprotocol/clientCapabilities": {}
    },
    "name": "get_weather",
    "arguments": {
      "location": "New York"
    }
  }
}
```

Copy

> **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

```ts
interface CallToolRequestParams {
  _meta: RequestMetaObject;
  inputResponses?: InputResponses;
  requestState?: string;
  name: string;
  arguments?: { [key: string]: unknown };
}
```

Parameters for a `tools/call` request.

▼ **Example: `get_weather` tool call params**

```
{
  "_meta": {
    "io.modelcontextprotocol/protocolVersion": "2026-07-28",
    "io.modelcontextprotocol/clientInfo": {
      "name": "ExampleClient",
      "version": "1.0.0"
    },
    "io.modelcontextprotocol/clientCapabilities": {}
  },
  "name": "get_weather",
  "arguments": {
    "location": "New York"
  }
} Copy
```

▼ **Example: Tool call params with progress token**

```
{
  "_meta": {
    "io.modelcontextprotocol/protocolVersion": "2026-07-28",
    "io.modelcontextprotocol/clientInfo": {
      "name": "ExampleClient",
      "version": "1.0.0"
    },
    "io.modelcontextprotocol/clientCapabilities": {},
    "progressToken": "oivaizmir"
  },
  "name": "build_simulation",
  "arguments": {
    "city": "Micropolis"
  }
} Copy
```

> **`_meta: RequestMetaObject`**
> Inherited from InputResponseRequestParams._meta

> **`inputResponses?: InputResponses`**
> Inherited from InputResponseRequestParams.inputResponses

> **`requestState?: string`**

> Inherited from InputResponseRequestParams.requestState

> **name: string**
>
> The name of the tool.

> **arguments?: { [key: string]: unknown }**
>
> Arguments to use for the tool call.

CallToolResultResponse

```
interface CallToolResultResponse {
  jsonrpc: "2.0";
  id: RequestId;
  result: InputRequiredResult | CallToolResult;
}
```

A successful response from the server for a tools/call request.

▼ **Example: Call tool result response**

```
{
  "jsonrpc": "2.0",
  "id": "call-tool-example",
  "result": {
    "resultType": "complete",
    "content": [
      {
        "type": "text",
        "text": "Current weather in New York:\nTemperature: 72°F\nConditions: Partly
cloudy"
      }
    ],
    "isError": false
  }
} Copy
```

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

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

> **result: InputRequiredResult | CallToolResult**
> Overrides JSONRPCResultResponse.result

CallToolResult

```
interface CallToolResult {
  _meta?: ResultMetaObject;
  resultType: string;
  content: ContentBlock[];
  structuredContent?: unknown;
  isError?: boolean;
  [key: string]: unknown;
}
```

The result returned by the server for a tools/call request.

▼ **Example: Result with unstructured text**

```
{
  "resultType": "complete",
  "content": [
    {
      "type": "text",
      "text": "Current weather in New York:\nTemperature: 72°F\nConditions: Partly
cloudy"
    }
  ],
  "isError": false
} Copy
```

▼ **Example: Result with structured content**

```
{
  "resultType": "complete",
  "content": [
    {
      "type": "text",
      "text": "{\"temperature\": 22.5, \"conditions\": \"Partly cloudy\", \"humidity\":
65}"
    }
  ],
  "structuredContent": {
    "temperature": 22.5,
    "conditions": "Partly cloudy",
    "humidity": 65
  }
} Copy
```

▼ **Example: Invalid tool input error**

```
{
  "resultType": "complete",
  "content": [
    {
      "type": "text",
      "text": "Invalid departure date: must be in the future. Current date is
08/08/2025."
    }
  ],
  "isError": true
} Copy
```

> **_meta?: ResultMetaObject**
>
> Inherited from Result._meta

> **resultType: string**
>
> Indicates the type of the result, which allows the client to determine how to parse the result object.
>
> Servers implementing this protocol version MUST include this field. For backward compatibility, when a client receives a result from a server implementing an earlier protocol version (which does not include `resultType`), the client MUST treat the absent field as `"complete"`.
>
> Inherited from Result.resultType

> **content: ContentBlock[]**
>
> A list of content objects that represent the unstructured result of the tool call.

> **structuredContent?: unknown**
>
> An optional JSON value that represents the structured result of the tool call.
>
> This can be any JSON value (object, array, string, number, boolean, or null) that conforms to the tool's outputSchema if one is defined.

> **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.

▼ **Example: List tools request**

```
{
  "jsonrpc": "2.0",
  "id": "list-tools-example",
  "method": "tools/list",
  "params": {
    "_meta": {
      "io.modelcontextprotocol/protocolVersion": "2026-07-28",
      "io.modelcontextprotocol/clientInfo": {
        "name": "ExampleClient",
        "version": "1.0.0"
      },
      "io.modelcontextprotocol/clientCapabilities": {}
    }
  }
} Copy
```

> **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

`ListToolsResultResponse`

```
interface ListToolsResultResponse {
  jsonrpc: "2.0";
  id: RequestId;
  result: ListToolsResult;
}
```

A successful response from the server for a tools/list request.

▼ **Example: List tools result response**

```
{
  "jsonrpc": "2.0",
  "id": "list-tools-example",
  "result": {
    "resultType": "complete",
    "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"]
          }
        ]
      }
    ],
    "nextCursor": "next-page-cursor",
    "ttlMs": 3600000,
    "cacheScope": "public"
  }
}
```

Copy

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

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

> **result: ListToolsResult**
> Overrides JSONRPCResultResponse.result

`ListToolsResult`

```ts
interface ListToolsResult {
  _meta?: ResultMetaObject;
  resultType: string;
  nextCursor?: string;
  ttlMs: number;
  cacheScope: "public" | "private";
  tools: Tool[];
  [key: string]: unknown;
}
```

The result returned by the server for a tools/list request.

▼ **Example: Tools list with cursor and TTL**

```ts
{
  "resultType": "complete",
  "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"]
        }
      ]
    }
  ],
  "nextCursor": "next-page-cursor",
  "ttlMs": 300000,
  "cacheScope": "public"
} Copy
```

_meta?: ResultMetaObject

Inherited from PaginatedResult._meta

resultType: string

Indicates the type of the result, which allows the client to determine how to parse the result object.

> Servers implementing this protocol version MUST include this field. For backward compatibility, when a client receives a result from a server implementing an earlier protocol version (which does not include `resultType`), the client MUST treat the absent field as `"complete"`.
>
> Inherited from PaginatedResult.resultType

> **`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

> **`ttlMs: number`**
>
> A hint from the server indicating how long (in milliseconds) the client MAY cache this response before re-fetching. Semantics are analogous to HTTP Cache-Control max-age.
>
> - If 0, The response SHOULD be considered immediately stale, The client MAY re-fetch every time the result is needed.
> - If positive, the client SHOULD consider the result fresh for this many milliseconds after receiving the response.
>
> Inherited from CacheableResult.ttlMs

> **`cacheScope: "public" | "private"`**
>
> Indicates the intended scope of the cached response, analogous to HTTP `Cache-Control: public` vs `Cache-Control: private`.
>
> - `"public"`: The response does not contain user-specific data. Any client or intermediary (e.g., shared gateway, caching proxy) MAY cache the response and serve it across authorization contexts.
> - `"private"`: The response MAY be cached and reused only within the same authorization context. Caches MUST NOT be shared across authorization contexts (e.g., a different access token requires a different cache).
>
> Inherited from CacheableResult.cacheScope

> **`tools: Tool[]`**

`Tool`

```ts
interface Tool {
  icons?: Icon[];
  name: string;
  title?: string;
  description?: string;
  inputSchema: { $schema?: string; type: "object"; [key: string]: unknown };
  outputSchema?: { $schema?: string; [key: string]: unknown };
  annotations?: ToolAnnotations;
  _meta?: MetaObject;
}
```

Definition for a tool the client can call.

▼ **Example: With default 2020-12 input schema**

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

▼ **Example: With explicit draft-07 input 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"]
  }
}
```
Copy

▼ **Example: With no parameters**

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

▼ **Example: With output schema for structured content**

```
{
  "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"]
  }
}
```

Copy

> **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: { $schema?: string; type: "object"; [key: string]: unknown }`**
>
> A JSON Schema object defining the expected parameters for the tool.
>
> Tool arguments are always JSON objects, so `type: "object"` is required at the root. Beyond that, any JSON Schema 2020-12 keyword may appear alongside `type` — including composition keywords (`oneOf` , `anyOf` , `allOf` , `not`), conditional keywords (`if` / `then` / `else`), reference keywords (`$ref` , `$defs` , `$anchor`), and any other standard validation or annotation keywords.
>
> Property schemas may carry an `x-mcp-header` annotation to mirror the argument value into an HTTP header on the Streamable HTTP transport. See the Streamable HTTP transport specification for the validity and extraction rules.
>
> Defaults to JSON Schema 2020-12 when no explicit `$schema` is provided.

> **`outputSchema?: { $schema?: string; [key: string]: unknown }`**
>
> An optional JSON Schema object defining the structure of the tool's output returned in the structuredContent field of a CallToolResult. This can be any valid JSON Schema 2020-12.
>
> Defaults to JSON Schema 2020-12 when no explicit `$schema` is provided.

> **`annotations?: ToolAnnotations`**
>
> Optional additional tool information.
>
> Display name precedence order is: `title` , `annotations.title` , then `name` .

> **`_meta?: MetaObject`**

`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