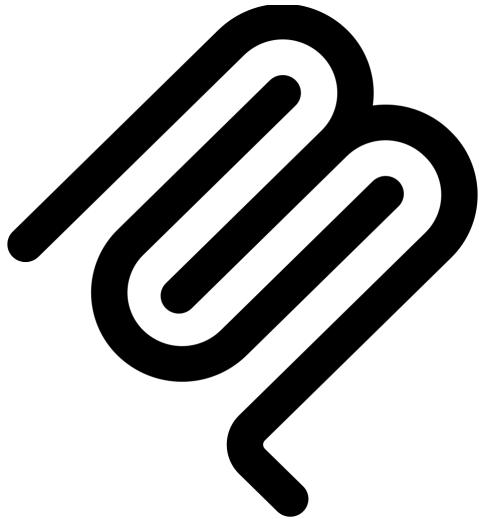


# Model Context Protocol - MCP



*Protocol Revision: 2025-06-18*

# Table of content

---

## [Table of content](#)

### [Disclaimer](#)

### [Connect your AI applications to the world](#)

How it works

### [Documentation](#)

Introduction

Choose Your Path

Ready to Build?

SDKs

Available SDKs

Getting Started

Next Steps

Architecture Overview

Scope

Concepts of MCP

Example

Server Concepts

Core Building Blocks

How It All Works Together

Client Concepts

Core Client Features

Versioning

Revisions

Negotiation

Connect to Remote MCP Servers

Understanding Remote MCP Servers

What are Custom Connectors?

Connecting to a Remote MCP Server

Best Practices for Using Remote MCP Servers

Next Steps

Connect to Local MCP Servers

Prerequisites

Understanding MCP Servers

Installing the Filesystem Server

Using the Filesystem Server

Troubleshooting

Next Steps

Build an MCP Server

What's happening under the hood

Troubleshooting

Next steps

Inspector

Getting started

Feature overview

Best practices

Next steps

Build an MCP Client

Next steps

FAQs

- What is MCP?
- Why does MCP matter?
- How does MCP work?
- Who creates and maintains MCP servers?

## Specification

- Overview

- Key Details

- Base Protocol

- Features

- Additional Utilities

- Security and Trust & Safety

- Key Principles

- Implementation Guidelines

- Learn More

- Major changes

- Other schema changes

- Full changelog

- Core Components

- Host

- Clients

- Servers

- Design Principles

- Capability Negotiation

- Messages

- Requests

- Responses

- Notifications

- Auth

- Schema

- General fields

- Lifecycle Phases

- Initialization

- Operation

- Shutdown

- Timeouts

- Error Handling

- Streamable HTTP

- Sending Messages to the Server

- Listening for Messages from the Server

- Multiple Connections

- Resumability and Redelivery

- Session Management

- Sequence Diagram

- Protocol Version Header

- Backwards Compatibility

- Custom Transports

- Introduction

- Purpose and Scope

- Protocol Requirements

- Standards Compliance

- Authorization Flow

- Roles

- Overview

- Authorization Server Discovery
- Dynamic Client Registration
- Authorization Flow Steps
- Access Token Usage
- Error Handling
- Security Considerations
  - Token Audience Binding and Validation
  - Token Theft
  - Communication Security
  - Authorization Code Protection
  - Open Redirection
  - Confused Deputy Problem
  - Access Token Privilege Restriction
- Introduction
  - Purpose and Scope
- Attacks and Mitigations
  - Confused Deputy Problem
  - Token Passthrough
  - Session Hijacking
- Cancellation Flow
- Behavior Requirements
- Timing Considerations
- Implementation Notes
- Error Handling
- Overview
- Message Format
- Behavior Requirements
- Usage Patterns
- Implementation Considerations
- Error Handling
- Progress Flow
- Behavior Requirements
- Implementation Notes
- User Interaction Model
- Capabilities
- Protocol Messages
  - Listing Roots
  - Root List Changes
- Message Flow
- Data Types
  - Root
- Error Handling
- Security Considerations
- Implementation Guidelines
- User Interaction Model
- Capabilities
- Protocol Messages
  - Creating Messages
- Message Flow
- Data Types
  - Messages
  - Model Preferences
- Error Handling

- Security Considerations
- User Interaction Model
- Capabilities
- Protocol Messages
  - Creating Elicitation Requests
- Message Flow
- Request Schema
  - Supported Schema Types
- Response Actions
- Security Considerations

## Server Features - Overview

- User Interaction Model
- Capabilities
- Protocol Messages
  - Listing Prompts
  - Getting a Prompt
  - List Changed Notification
- Message Flow
- Data Types
  - Prompt
  - PromptMessage
- Error Handling
- Implementation Considerations
- Security
- User Interaction Model
- Capabilities
- Protocol Messages
  - Listing Resources
  - Reading Resources
  - Resource Templates
  - List Changed Notification
  - Subscriptions
- Message Flow
- Data Types
  - Resource
  - Resource Contents
  - Annotations
- Common URI Schemes
  - https://
  - file://
  - git://
- Custom URI Schemes
- Error Handling
- Security Considerations
- User Interaction Model
- Capabilities
- Protocol Messages
  - Listing Tools
  - Calling Tools
  - List Changed Notification
- Message Flow
- Data Types
  - Tool

- Tool Result
- Error Handling
- Security Considerations
- User Interaction Model
- Capabilities
- Protocol Messages
  - Requesting Completions
  - Reference Types
  - Completion Results
- Message Flow
- Data Types
  - CompleteRequest
  - CompleteResult
- Error Handling
- Implementation Considerations
- Security
- User Interaction Model
- Capabilities
- Log Levels
- Protocol Messages
  - Setting Log Level
  - Log Message Notifications
- Message Flow
- Error Handling
- Implementation Considerations
- Security
- Pagination Model
- Response Format
- Request Format
- Pagination Flow
- Operations Supporting Pagination
- Implementation Guidelines
- Error Handling
- Common Types
  - Annotations
  - AudioContent
  - BlobResourceContents
  - BooleanSchema
  - ClientCapabilities
  - ContentBlock
  - Cursor
  - EmbeddedResource
  - EmptyResult
  - EnumSchema
  - ImageContent
  - Implementation
  - JSONRPCError
  - JSONRPCNotification
  - JSONRPCRequest
  - JSONRPCResponse
  - LogLevel
  - ModelHint

`ModelPreferences`  
`NumberSchema`  
`PrimitiveSchemaDefinition`  
`ProgressToken`  
`Prompt`  
`PromptArgument`  
`PromptMessage`  
`PromptReference`  
`RequestId`  
`Resource`  
`ResourceContents`  
`ResourceLink`  
`ResourceTemplate`  
`ResourceTemplateReference`  
`Result`  
`Role`  
`Root`  
`SamplingMessage`  
`ServerCapabilities`  
`StringSchema`  
`TextContent`  
`TextResourceContents`  
`Tool`  
`ToolAnnotations`  
`completion/complete`  
    `CompleteRequest`  
    `CompleteResult`  
`elicitation/create`  
    `ElicitRequest`  
    `ElicitResult`  
`initialize`  
    `InitializeRequest`  
    `InitializeResult`  
`logging/setLevel`  
    `SetLevelRequest`  
`notifications/cancelled`  
    `CancelledNotification`  
`notifications/initialized`  
    `InitializedNotification`  
`notifications/message`  
    `LoggingMessageNotification`  
`notifications/progress`  
    `ProgressNotification`  
`notifications/prompts/list_changed`  
    `PromptListChangedNotification`  
`notifications/resources/list_changed`  
    `ResourceListChangedNotification`  
`notifications/resources/updated`  
    `ResourceUpdatedNotification`  
`notifications/roots/list_changed`

```
    RootsListChangedNotification  
    notifications/tools/list_changed  
        ToolListChangedNotification  
  
    ping  
        PingRequest  
  
    prompts/get  
        GetPromptRequest  
        GetPromptResult  
  
    prompts/list  
        ListPromptsRequest  
        ListPromptsResult  
  
    resources/list  
        ListResourcesRequest  
        ListResourcesResult  
  
    resources/read  
        ReadResourceRequest  
        ReadResourceResult  
  
    resources/subscribe  
        SubscribeRequest  
  
    resources/templates/list  
        ListResourceTemplatesRequest  
        ListResourceTemplatesResult  
  
    resources/unsubscribe  
        UnsubscribeRequest  
  
    roots/list  
        ListRootsRequest  
        ListRootsResult  
  
    sampling/createMessage  
        CreateMessageRequest  
        CreateMessageResult  
  
    tools/call  
        CallToolRequest  
        CallToolResult  
  
    tools/list  
        ListToolsRequest  
        ListToolsResult
```

# Disclaimer

---

This eBook has been compiled and published by Nicolas Riousset for the sole purpose of facilitating access to and dissemination of the Model Context Protocol specifications. All original content, including text, diagrams, and other materials, remains the property of their respective authors and copyright holders.

While every effort has been made to ensure accuracy, this eBook is provided "as is", without warranties of any kind, express or implied, including but not limited to accuracy, completeness, or fitness for a particular purpose.

The compiler, Nicolas Riousset, is not affiliated with, endorsed by, or representing the official maintainers of the Model Context Protocol. Any errors, omissions, or formatting changes introduced during the compilation process are solely the responsibility of the compiler.

By using this eBook, you acknowledge that you do so at your own risk, and you agree that the compiler shall not be held liable for any damages, direct or indirect, arising from its use.

# Connect your AI applications to the world

---

AI-enabled tools are powerful, but they're often limited to the information you manually provide or require bespoke integrations.

Whether it's reading files from your computer, searching through an internal or external knowledge base, or updating tasks in a project management tool, MCP provides a secure, standardized, *simple* way to give AI systems the context they need.

# How it works

---

## 1 Choose MCP servers

Pick from pre-built servers for popular tools like GitHub, Google Drive, Slack and hundreds of others. Combine multiple servers for complete workflows, or easily build your own for custom integrations.

## 2 Connect your AI application

Configure your AI application (like Claude, VS Code, or ChatGPT) to connect to your MCP servers. The application can now see available tools, resources and prompts from all connected servers.

## 3 Work with context

Your AI-powered application can now access real data, execute actions, and provide more helpful responses based on your actual context.

# Documentation

---

# Introduction

---

Get started with the Model Context Protocol (MCP)

MCP is an open protocol that standardizes how applications provide context to large language models (LLMs). Think of MCP like a USB-C port for AI applications. Just as USB-C provides a standardized way to connect your devices to various peripherals and accessories, MCP provides a standardized way to connect AI models to different data sources and tools. MCP enables you to build agents and complex workflows on top of LLMs and connects your models with the world.

MCP provides:

- **A growing list of pre-built integrations** that your LLM can directly plug into
- **A standardized way** to build custom integrations for AI applications
- **An open protocol** that everyone is free to implement and use
- **The flexibility to change** between different apps and take your context with you

## Choose Your Path

Learn the core concepts and architecture of MCP

{ " " }

Connect to existing MCP servers and start using them

{ " " }

Create MCP servers to expose your data and tools

Develop applications that connect to MCP servers

## Ready to Build?

MCP provides official **SDKs** in multiple languages, see the [SDK documentation](#) to find the right SDK for your project. The SDKs handle the protocol details so you can focus on building your features.

# SDKs

---

Official SDKs for building with the Model Context Protocol

Build MCP servers and clients using our official SDKs. Choose the SDK that matches your technology stack - all SDKs provide the same core functionality and full protocol support.

## Available SDKs

## Getting Started

Each SDK provides the same functionality but follows the idioms and best practices of its language. All SDKs support:

- Creating MCP servers that expose tools, resources, and prompts
- Building MCP clients that can connect to any MCP server
- Local and Remote transport protocols
- Protocol compliance with type safety

Visit the SDK page for your chosen language to find installation instructions, documentation, and examples.

## Next Steps

Ready to start building with MCP? Choose your path:

[Learn how to create your first MCP server](#)

[Create applications that connect to MCP servers](#)

Browse pre-built servers for inspiration

Dive deeper into how MCP works

# Architecture Overview

---

This overview of the Model Context Protocol (MCP) discusses its [scope](#) and [core concepts](#), and provides an [example](#) demonstrating each core concept.

Because MCP SDKs abstract away many concerns, most developers will likely find the [data layer protocol](#) section to be the most useful. It discusses how MCP servers can provide context to an AI application.

For specific implementation details, please refer to the documentation for your [language-specific SDK](#).

## Scope

The Model Context Protocol includes the following projects:

- [MCP Specification](#): A specification of MCP that outlines the implementation requirements for clients and servers.
- [MCP SDKs](#): SDKs for different programming languages that implement MCP.
- **MCP Development Tools**: Tools for developing MCP servers and clients, including the [MCP Inspector](#)
- [MCP Reference Server Implementations](#): Reference implementations of MCP servers.

MCP focuses solely on the protocol for context exchange—it does not dictate how AI applications use LLMs or manage the provided context.

## Concepts of MCP

### Participants

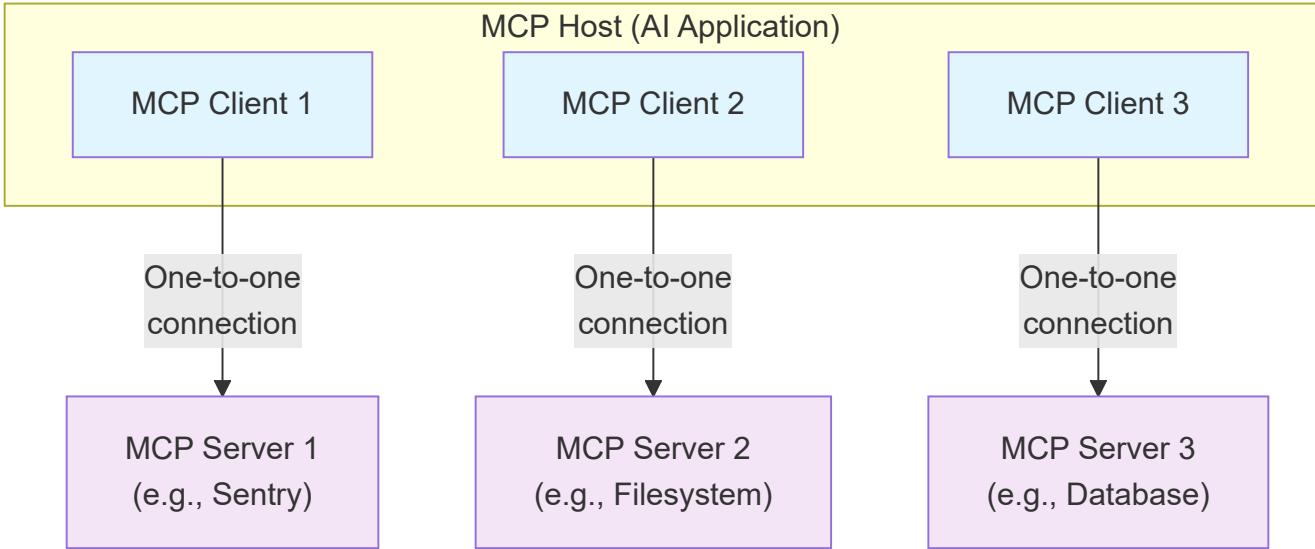
MCP follows a client-server architecture where an MCP host — an AI application like [Claude Code](#) or [Claude Desktop](#) — establishes connections to one or more MCP servers. The MCP host accomplishes this by creating one MCP client for each MCP server. Each MCP client maintains a dedicated one-to-one connection with its corresponding MCP server.

The key participants in the MCP architecture are:

- **MCP Host**: The AI application that coordinates and manages one or multiple MCP clients
- **MCP Client**: A component that maintains a connection to an MCP server and obtains context from an MCP server for the MCP host to use
- **MCP Server**: A program that provides context to MCP clients

**For example:** Visual Studio Code acts as an MCP host. When Visual Studio Code establishes a connection to an MCP server, such as the [Sentry MCP server](#), the Visual Studio Code runtime instantiates an MCP client object that maintains the connection to the Sentry MCP server.

When Visual Studio Code subsequently connects to another MCP server, such as the [local filesystem server](#), the Visual Studio Code runtime instantiates an additional MCP client object to maintain this connection, hence maintaining a one-to-one relationship of MCP clients to MCP servers.



Note that **MCP server** refers to the program that serves context data, regardless of where it runs. MCP servers can execute locally or remotely. For example, when Claude Desktop launches the [filesystem server](#), the server runs locally on the same machine because it uses the STDIO transport. This is commonly referred to as a "local" MCP server. The official [Sentry MCP server](#) runs on the Sentry platform, and uses the Streamable HTTP transport. This is commonly referred to as a "remote" MCP server.

## Layers

MCP consists of two layers:

- **Data layer:** Defines the JSON-RPC based protocol for client-server communication, including lifecycle management, and core primitives, such as tools, resources, prompts and notifications.
- **Transport layer:** Defines the communication mechanisms and channels that enable data exchange between clients and servers, including transport-specific connection establishment, message framing, and authorization.

Conceptually the data layer is the inner layer, while the transport layer is the outer layer.

### Data layer

The data layer implements a [JSON-RPC 2.0](#) based exchange protocol that defines the message structure and semantics.

This layer includes:

- **Lifecycle management:** Handles connection initialization, capability negotiation, and connection termination between clients and servers
- **Server features:** Enables servers to provide core functionality including tools for AI actions, resources for context data, and prompts for interaction templates from and to the client
- **Client features:** Enables servers to ask the client to sample from the host LLM, elicit input from the user, and log messages to the client
- **Utility features:** Supports additional capabilities like notifications for real-time updates and progress tracking for long-running operations

## Transport layer

The transport layer manages communication channels and authentication between clients and servers. It handles connection establishment, message framing, and secure communication between MCP participants.

MCP supports two transport mechanisms:

- **Stdio transport:** Uses standard input/output streams for direct process communication between local processes on the same machine, providing optimal performance with no network overhead.
- **Streamable HTTP transport:** Uses HTTP POST for client-to-server messages with optional Server-Sent Events for streaming capabilities. This transport enables remote server communication and supports standard HTTP authentication methods including bearer tokens, API keys, and custom headers. MCP recommends using OAuth to obtain authentication tokens.

The transport layer abstracts communication details from the protocol layer, enabling the same JSON-RPC 2.0 message format across all transport mechanisms.

## Data Layer Protocol

A core part of MCP is defining the schema and semantics between MCP clients and MCP servers. Developers will likely find the data layer — in particular, the set of [primitives](#) — to be the most interesting part of MCP. It is the part of MCP that defines the ways developers can share context from MCP servers to MCP clients.

MCP uses [JSON-RPC 2.0](#) as its underlying RPC protocol. Client and servers send requests to each other and respond accordingly. Notifications can be used when no response is required.

### Lifecycle management

MCP is a stateful protocol that requires lifecycle management. The purpose of lifecycle management is to negotiate the capabilities that both client and server support. Detailed information can be found in the [specification](#), and the [example](#) showcases the initialization sequence.

### Primitives

MCP primitives are the most important concept within MCP. They define what clients and servers can offer each other. These primitives specify the types of contextual information that can be shared with AI applications and the range of actions that can be performed.

MCP defines three core primitives that *servers* can expose:

- **Tools:** Executable functions that AI applications can invoke to perform actions (e.g., file operations, API calls, database queries)
- **Resources:** Data sources that provide contextual information to AI applications (e.g., file contents, database records, API responses)
- **Prompts:** Reusable templates that help structure interactions with language models (e.g., system prompts, few-shot examples)

Each primitive type has associated methods for discovery (`*/list`), retrieval (`*/get`), and in some cases, execution (`tools/call`).

MCP clients will use the `*/list` methods to discover available primitives. For example, a client can first list all available tools (`tools/list`) and then execute them. This design allows listings to be dynamic.

As a concrete example, consider an MCP server that provides context about a database. It can expose tools for querying the database, a resource that contains the schema of the database, and a prompt that includes few-shot examples for interacting with the tools.

For more details about server primitives see [server concepts](#).

MCP also defines primitives that *clients* can expose. These primitives allow MCP server authors to build richer interactions.

- **Sampling:** Allows servers to request language model completions from the client's AI application. This is useful when servers' authors want access to a language model, but want to stay model independent and not include a language model SDK in their MCP server. They can use the `sampling/complete` method to request a language model completion from the client's AI application.
- **Elicitation:** Allows servers to request additional information from users. This is useful when servers' authors want to get more information from the user, or ask for confirmation of an action. They can use the `elicitation/request` method to request additional information from the user.
- **Logging:** Enables servers to send log messages to clients for debugging and monitoring purposes.

For more details about client primitives see [client concepts](#).

## Notifications

The protocol supports real-time notifications to enable dynamic updates between servers and clients. For example, when a server's available tools change—such as when new functionality becomes available or existing tools are modified—the server can send tool update notifications to inform connected clients about these changes. Notifications are sent as JSON-RPC 2.0 notification messages (without expecting a response) and enable MCP servers to provide real-time updates to connected clients.

# Example

## Data Layer

This section provides a step-by-step walkthrough of an MCP client-server interaction, focusing on the data layer protocol. We'll demonstrate the lifecycle sequence, tool operations, and notifications using JSON-RPC 2.0 messages.

MCP begins with lifecycle management through a capability negotiation handshake. As described in the [lifecycle management](#) section, the client sends an `initialize` request to establish the connection and negotiate supported features.

```
<CodeGroup>
  ```json Request
  {
    "jsonrpc": "2.0",
    "id": 1,
    "method": "initialize",
    "params": {
      "protocolversion": "2025-06-18",
      "capabilities": {
        "elicitation": {}
      }
    }
  }```
  
```

```

        },
        "clientInfo": {
            "name": "example-client",
            "version": "1.0.0"
        }
    }
}
```
```json Response
{
    "jsonrpc": "2.0",
    "id": 1,
    "result": {
        "protocolversion": "2025-06-18",
        "capabilities": {
            "tools": {
                "listChanged": true
            },
            "resources": {}
        },
        "serverInfo": {
            "name": "example-server",
            "version": "1.0.0"
        }
    }
}
```
```
</CodeGroup>

```

#### ##### Understanding the Initialization Exchange

The initialization process is a key part of MCP's lifecycle management and serves several critical purposes:

- Protocol Version Negotiation**: The `protocolversion` field (e.g., "2025-06-18") ensures both client and server are using compatible protocol versions. This prevents communication errors that could occur when different versions attempt to interact. If a mutually compatible version is not negotiated, the connection should be terminated.
- Capability Discovery**: The `capabilities` object allows each party to declare what features they support, including which [primitives](#primitives) they can handle (tools, resources, prompts) and whether they support features like [notifications](#notifications). This enables efficient communication by avoiding unsupported operations.
- Identity Exchange**: The `clientInfo` and `serverInfo` objects provide identification and versioning information for debugging and compatibility purposes.

In this example, the capability negotiation demonstrates how MCP primitives are declared:

**Client Capabilities**:

```
* `{"elicitation": {}}` - The client declares it can work with user interaction requests (can receive `elicitation/create` method calls)
```

\*\*Server Capabilities\*\*:

```
* `{"tools": {"listChanged": true}}` - The server supports the tools primitive AND can send `tools/list_changed` notifications when its tool list changes
```

```
* `{"resources": {}}` - The server also supports the resources primitive (can handle `resources/list` and `resources/read` methods)
```

After successful initialization, the client sends a notification to indicate it's ready:

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

```

#### ##### How This Works in AI Applications

During initialization, the AI application's MCP client manager establishes connections to configured servers and stores their capabilities for later use. The application uses this information to determine which servers can provide specific types of functionality (tools, resources, prompts) and whether they support real-time updates.

```
```python Pseudo-code for AI application initialization
## Pseudo Code
async with stdio_client(server_config) as (read, write):
    async with ClientSession(read, write) as session:
        init_response = await session.initialize()
        if init_response.capabilities.tools:
            app.register_mcp_server(session, supports_tools=True)
        app.set_server_ready(session)
```

```

Now that the connection is established, the client can discover available tools by sending a `tools/list` request. This request is fundamental to MCP's tool discovery mechanism — it allows clients to understand what tools are available on the server before attempting to use them.

```
<CodeGroup>
    ```json Request
    {
        "jsonrpc": "2.0",
        "id": 2,
        "method": "tools/list"
    }
```
```json Response

```

```

{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "tools": [
      {
        "name": "calculator_arithmetic",
        "title": "Calculator",
        "description": "Perform mathematical calculations including basic arithmetic, trigonometric functions, and algebraic operations",
        "inputSchema": {
          "type": "object",
          "properties": {
            "expression": {
              "type": "string",
              "description": "Mathematical expression to evaluate (e.g., '2 + 3 * 4', 'sin(30)', 'sqrt(16)')"
            }
          },
          "required": ["expression"]
        }
      },
      {
        "name": "weather_current",
        "title": "Weather Information",
        "description": "Get current weather information for any location worldwide",
        "inputSchema": {
          "type": "object",
          "properties": {
            "location": {
              "type": "string",
              "description": "City name, address, or coordinates (latitude,longitude)"
            },
            "units": {
              "type": "string",
              "enum": ["metric", "imperial", "kelvin"],
              "description": "Temperature units to use in response",
              "default": "metric"
            }
          },
          "required": ["location"]
        }
      }
    ]
  }
}
```

```

</CodeGroup>

#### ##### Understanding the Tool Discovery Request

The `tools/list` request is simple, containing no parameters.

## ##### Understanding the Tool Discovery Response

The response contains a `tools` array that provides comprehensive metadata about each available tool. This array-based structure allows servers to expose multiple tools simultaneously while maintaining clear boundaries between different functionalities.

Each tool object in the response includes several key fields:

- \* \*\*`name`\*\*: A unique identifier for the tool within the server's namespace. This serves as the primary key for tool execution and should follow a clear naming pattern (e.g., `calculator\_arithmetic` rather than just `calculate`)
- \* \*\*`title`\*\*: A human-readable display name for the tool that clients can show to users
- \* \*\*`description`\*\*: Detailed explanation of what the tool does and when to use it
- \* \*\*`inputSchema`\*\*: A JSON Schema that defines the expected input parameters, enabling type validation and providing clear documentation about required and optional parameters

## ##### How This Works in AI Applications

The AI application fetches available tools from all connected MCP servers and combines them into a unified tool registry that the language model can access. This allows the LLM to understand what actions it can perform and automatically generates the appropriate tool calls during conversations.

```
```python Pseudo-code for AI application tool discovery
## Pseudo-code using MCP Python SDK patterns
available_tools = []
for session in app.mcp_server_sessions():
    tools_response = await session.list_tools()
    available_tools.extend(tools_response.tools)
conversation.register_available_tools(available_tools)
...```

```

The client can now execute a tool using the `tools/call` method. This demonstrates how MCP primitives are used in practice: after discovering available tools, the client can invoke them with appropriate arguments.

## ##### Understanding the Tool Execution Request

The `tools/call` request follows a structured format that ensures type safety and clear communication between client and server. Note that we're using the proper tool name from the discovery response (`weather\_current`) rather than a simplified name:

```
<CodeGroup>
  ```json Request
  {
    "jsonrpc": "2.0",
    "id": 3,
    "method": "tools/call",
    "params": {
      "name": "weather_current",
      "arguments": {```

```

```

        "location": "San Francisco",
        "units": "imperial"
    }
}
```
```json Response
{
    "jsonrpc": "2.0",
    "id": 3,
    "result": {
        "content": [
            {
                "type": "text",
                "text": "Current weather in San Francisco: 68°F, partly cloudy with light winds from the west at 8 mph. Humidity: 65%"
            }
        ]
    }
}
```
</CodeGroup>
```

#### ##### Key Elements of Tool Execution

The request structure includes several important components:

1. \*\*`name`\*\*: Must match exactly the tool name from the discovery response (`weather\_current`). This ensures the server can correctly identify which tool to execute.
2. \*\*`arguments`\*\*: Contains the input parameters as defined by the tool's `inputschema`. In this example:

```
* `location`: "San Francisco" (required parameter)
* `units`: "imperial" (optional parameter, defaults to "metric" if not specified)
```

3. \*\*JSON-RPC Structure\*\*: Uses standard JSON-RPC 2.0 format with unique `id` for request-response correlation.

#### ##### Understanding the Tool Execution Response

The response demonstrates MCP's flexible content system:

1. \*\*`content` Array\*\*: Tool responses return an array of content objects, allowing for rich, multi-format responses (text, images, resources, etc.)
2. \*\*Content Types\*\*: Each content object has a `type` field. In this example, `type`: "text" indicates plain text content, but MCP supports various content types for different use cases.
3. \*\*Structured Output\*\*: The response provides actionable information that the AI application can use as context for language model interactions.

This execution pattern allows AI applications to dynamically invoke server functionality and receive structured responses that can be integrated into conversations with language models.

#### ##### How This Works in AI Applications

When the language model decides to use a tool during a conversation, the AI application intercepts the tool call, routes it to the appropriate MCP server, executes it, and returns the results back to the LLM as part of the conversation flow. This enables the LLM to access real-time data and perform actions in the external world.

```
```python
## Pseudo-code for AI application tool execution
async def handle_tool_call(conversation, tool_name, arguments):
    session = app.find_mcp_session_for_tool(tool_name)
    result = await session.call_tool(tool_name, arguments)
    conversation.add_tool_result(result.content)
```

```

MCP supports real-time notifications that enable servers to inform clients about changes without being explicitly requested. This demonstrates the notification system, a key feature that keeps MCP connections synchronized and responsive.

#### ##### Understanding Tool List Change Notifications

When the server's available tools change—such as when new functionality becomes available, existing tools are modified, or tools become temporarily unavailable—the server can proactively notify connected clients:

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

```

#### ##### Key Features of MCP Notifications

1. **No Response Required**: Notice there's no `id` field in the notification. This follows JSON-RPC 2.0 notification semantics where no response is expected or sent.
2. **Capability-Based**: This notification is only sent by servers that declared `listChanged` in their tools capability during initialization (as shown in Step 1).
3. **Event-Driven**: The server decides when to send notifications based on internal state changes, making MCP connections dynamic and responsive.

#### ##### Client Response to Notifications

Upon receiving this notification, the client typically reacts by requesting the updated tool list. This creates a refresh cycle that keeps the client's understanding of available tools current:

```
```json Request
{
  "jsonrpc": "2.0",
  "id": 4,
  "method": "tools/list"
}
```

```

#### ##### Why Notifications Matter

This notification system is crucial for several reasons:

1. **Dynamic Environments**: Tools may come and go based on server state, external dependencies, or user permissions
2. **Efficiency**: Clients don't need to poll for changes; they're notified when updates occur
3. **Consistency**: Ensures clients always have accurate information about available server capabilities
4. **Real-time Collaboration**: Enables responsive AI applications that can adapt to changing contexts

This notification pattern extends beyond tools to other MCP primitives, enabling comprehensive real-time synchronization between clients and servers.

#### ##### How This Works in AI Applications

When the AI application receives a notification about changed tools, it immediately refreshes its tool registry and updates the LLM's available capabilities. This ensures that ongoing conversations always have access to the most current set of tools, and the LLM can dynamically adapt to new functionality as it becomes available.

```
```python
## Pseudo-code for AI application notification handling
async def handle_tools_changed_notification(session):
    tools_response = await session.list_tools()
    app.update_available_tools(session, tools_response.tools)
    if app.conversation.is_active():
        app.conversation.notify_llm_of_new_capabilities()
```

```

# Server Concepts

Understanding MCP server concepts

MCP servers are programs that expose specific capabilities to AI applications through standardized protocol interfaces. Each server provides focused functionality for a particular domain.

Common examples include file system servers for document management, email servers for message handling, travel servers for trip planning, and database servers for data queries. Each server brings domain-specific capabilities to the AI application.

## Core Building Blocks

Servers provide functionality through three building blocks:

| Building Block | Purpose                   | Who Controls It        | Real-World Example                                           |
|----------------|---------------------------|------------------------|--------------------------------------------------------------|
| Tools          | For AI actions            | Model-controlled       | Search flights, send messages, create calendar events        |
| Resources      | For context data          | Application-controlled | Documents, calendars, emails, weather data                   |
| Prompts        | For interaction templates | User-controlled        | "Plan a vacation", "Summarize my meetings", "Draft an email" |

## Tools - AI Actions

Tools enable AI models to perform actions through server-implemented functions. Each tool defines a specific operation with typed inputs and outputs. The model requests tool execution based on context.

### Overview

Tools are schema-defined interfaces that LLMs can invoke. MCP uses JSON Schema for validation. Each tool performs a single operation with clearly defined inputs and outputs. Most importantly, tool execution requires explicit user approval, ensuring users maintain control over actions taken by a model.

### Protocol operations:

| Method                  | Purpose                  | Returns                                |
|-------------------------|--------------------------|----------------------------------------|
| <code>tools/list</code> | Discover available tools | Array of tool definitions with schemas |
| <code>tools/call</code> | Execute a specific tool  | Tool execution result                  |

### Example tool definition:

```
{  
  name: "searchFlights",  
  description: "Search for available flights",  
  inputSchema: {  
    type: "object",  
    properties: {  
      origin: { type: "string", description: "Departure city" },  
      destination: { type: "string", description: "Arrival city" },  
      date: { type: "string", format: "date", description: "Travel date" }  
    },  
    required: ["origin", "destination", "date"]  
  },  
}
```

## Example: Taking Action

Tools enable AI applications to perform actions on behalf of users. In a travel planning scenario, the AI application might use several tools to help book a vacation.

First, it searches for flights using

```
searchFlights(origin: "NYC", destination: "Barcelona", date: "2024-06-15")
```

`searchFlights` queries multiple airlines and returns structured flight options. Once flights are selected, it creates a calendar event with

```
createCalendarEvent(title: "Barcelona Trip", startDate: "2024-06-15", endDate: "2024-06-22")
```

to mark the travel dates. Finally, it sends an out-of-office notification using

```
sendEmail(to: "team@work.com", subject: "Out of office", body: "...")
```

to inform colleagues about the absence.

Each tool execution requires explicit user approval, ensuring full control over actions taken.

## User Interaction Model

Tools are model-controlled, meaning AI models can discover and invoke them automatically. However, MCP emphasizes human oversight through several mechanisms. Applications should clearly display available tools in the UI and provide visual indicators when tools are being considered or used. Before any tool execution, users must be presented with clear approval dialogs that explain exactly what the tool will do.

For trust and safety, applications often enforce manual approval to give humans the ability to deny tool invocations. Applications typically implement this through approval dialogs, permission settings for pre-approving certain safe operations, and activity logs that show all tool executions with their results.

## Resources - Context Data

Resources provide structured access to information that the host application can retrieve and provide to AI models as context.

### Overview

Resources expose data from files, APIs, databases, or any other source that an AI needs to understand context. Applications can access this information directly and decide how to use it - whether that's selecting relevant portions, searching with embeddings, or passing it all to the model.

Resources use URI-based identification, with each resource having a unique URI such as `file:///path/to/document.md`. They declare MIME types for appropriate content handling and support two discovery patterns: **direct resources** with fixed URIs, and **resource templates** with parameterized URIs.

**Resource Templates** enable dynamic resource access through URI templates. A template like `travel://activities/{city}/{category}` would access filtered activity data by substituting both `{city}` and `{category}` parameters. For example, `travel://activities/barcelona/museums` would return all museums in Barcelona. Resource Templates include metadata such as title, description, and expected MIME type, making them discoverable and self-documenting.

### Protocol operations:

| Method                                | Purpose                         | Returns                                |
|---------------------------------------|---------------------------------|----------------------------------------|
| <code>resources/list</code>           | List available direct resources | Array of resource descriptors          |
| <code>resources/templates/list</code> | Discover resource templates     | Array of resource template definitions |
| <code>resources/read</code>           | Retrieve resource contents      | Resource data with metadata            |
| <code>resources/subscribe</code>      | Monitor resource changes        | Subscription confirmation              |

### Example: Accessing Context Data

Continuing with the travel planning example, resources provide the AI application with access to relevant information:

- **Calendar data** (`calendar://events/2024`) - To check availability
- **Travel documents** (`file:///Documents/Travel/passport.pdf`) - For important information
- **Previous itineraries** (`trips://history/barcelona-2023`) - User selects which past trip style to follow

Instead of manually copying this information, resources provide raw information to AI applications. The application can choose how to best handle the data. Applications might choose to select a subset of data, using embeddings or keyword search, or pass the raw data from a resource directly to a model. In our example, during the planning phase, the AI application can pass the calendar data, weather data and travel preferences, so that the model can check availability, look up weather patterns, and reference travel preferences.

### Resource Template Examples:

```
{  
  "uriTemplate": "weather://forecast/{city}/{date}",
```

```

    "name": "weather-forecast",
    "title": "Weather Forecast",
    "description": "Get weather forecast for any city and date",
    "mimeType": "application/json"
}

{
  "uriTemplate": "travel://flights/{origin}/{destination}",
  "name": "flight-search",
  "title": "Flight Search",
  "description": "Search available flights between cities",
  "mimeType": "application/json"
}

```

These templates enable flexible queries. For weather data, users can access forecasts for any city/date combination. For flights, they can search routes between any two airports. When a user has input "NYC" as the `origin` airport and begins to input "Bar" as the `destination` airport, the system can suggest "Barcelona (BCN)" or "Barbados (BGI)".

## Parameter Completion

Dynamic resources support parameter completion. For example:

- Typing "Par" as input for `weather://forecast/{city}` might suggest "Paris" or "Park City"
- The system helps discover valid values without requiring exact format knowledge

## User Interaction Model

Resources are application-driven, giving hosts flexibility in how they retrieve, process, and present available context. Common interaction patterns include tree or list views for browsing resources in familiar folder-like structures, search and filter interfaces for finding specific resources, automatic context inclusion based on heuristics or AI selection, and manual selection interfaces.

Applications are free to implement resource discovery through any interface pattern that suits their needs. The protocol doesn't mandate specific UI patterns, allowing for resource pickers with preview capabilities, smart suggestions based on current conversation context, bulk selection for including multiple resources, or integration with existing file browsers and data explorers.

## Prompts - Interaction Templates

Prompts provide reusable templates. They allow MCP server authors to provide parameterized prompts for a domain, or showcase how to best use the MCP server.

### Overview

Prompts are structured templates that define expected inputs and interaction patterns. They are user-controlled, requiring explicit invocation rather than automatic triggering. Prompts can be context-aware, referencing available resources and tools to create comprehensive workflows. Like resources, prompts support parameter completion to help users discover valid argument values.

### Protocol operations:

| Method        | Purpose                    | Returns                               |
|---------------|----------------------------|---------------------------------------|
| /prompts/list | Discover available prompts | Array of prompt descriptors           |
| /prompts/get  | Retrieve prompt details    | Full prompt definition with arguments |

## Example: Streamlined Workflows

Prompts provide structured templates for common tasks. In the travel planning context:

### "Plan a vacation" prompt:

```
{
  "name": "plan-vacation",
  "title": "Plan a vacation",
  "description": "Guide through vacation planning process",
  "arguments": [
    { "name": "destination", "type": "string", "required": true },
    { "name": "duration", "type": "number", "description": "days" },
    { "name": "budget", "type": "number", "required": false },
    { "name": "interests", "type": "array", "items": { "type": "string" } }
  ]
}
```

Rather than unstructured natural language input, the prompt system enables:

1. Selection of the "Plan a vacation" template
2. Structured input: Barcelona, 7 days, \$3000, ["beaches", "architecture", "food"]
3. Consistent workflow execution based on the template

## User Interaction Model

Prompts are user-controlled, requiring explicit invocation. Applications typically expose prompts through various UI patterns such as slash commands (typing "/" to see available prompts like /plan-vacation), command palettes for searchable access, dedicated UI buttons for frequently used prompts, or context menus that suggest relevant prompts.

The protocol gives implementers freedom to design interfaces that feel natural within their application. Key principles include easy discovery of available prompts, clear descriptions of what each prompt does, natural argument input with validation, and transparent display of the prompt's underlying template.

## How It All Works Together

The real power of MCP emerges when multiple servers work together, combining their specialized capabilities through a unified interface.

## Example: Multi-Server Travel Planning

Consider an AI application with three connected servers:

1. **Travel Server** - Handles flights, hotels, and itineraries
2. **Weather Server** - Provides climate data and forecasts

### 3. Calendar/Email Server - Manages schedules and communications

## The Complete Flow

### 1. User invokes a prompt with parameters:

```
{  
  "prompt": "plan-vacation",  
  "arguments": {  
    "destination": "Barcelona",  
    "departure_date": "2024-06-15",  
    "return_date": "2024-06-22",  
    "budget": 3000,  
    "travelers": 2  
  }  
}
```

### 2. User selects resources to include:

- o `calendar://my-calendar/June-2024` (from Calendar Server)
- o `travel://preferences/europe` (from Travel Server)
- o `travel://past-trips/spain-2023` (from Travel Server)

### 3. AI processes the request:

The AI first reads all selected resources to gather context. From the calendar, it identifies available dates. From travel preferences, it learns preferred airlines and hotel types. From past trips, it discovers previously enjoyed locations. From weather data, it checks climate conditions for the travel period.

Using this context, the AI then requests user approval to execute a series of coordinated actions: searching for flights from NYC to Barcelona, finding hotels within the specified budget, creating a calendar event for the trip duration, and sending confirmation emails with the trip details.

# Client Concepts

---

## Understanding MCP client concepts

MCP clients are instantiated by host applications to communicate with particular MCP servers. The host application, like Claude.ai or an IDE, manages the overall user experience and coordinates multiple clients. Each client handles one direct communication with one server.

Understanding the distinction is important: the *host* is the application users interact with, while *clients* are the protocol-level components that enable server connections.

## Core Client Features

In addition to making use of context provided by servers, clients may provide several features to servers. These client features allow server authors to build richer interactions. For example, clients can allow MCP servers to request additional information from the user via elicitations. Clients can offer the following capabilities:

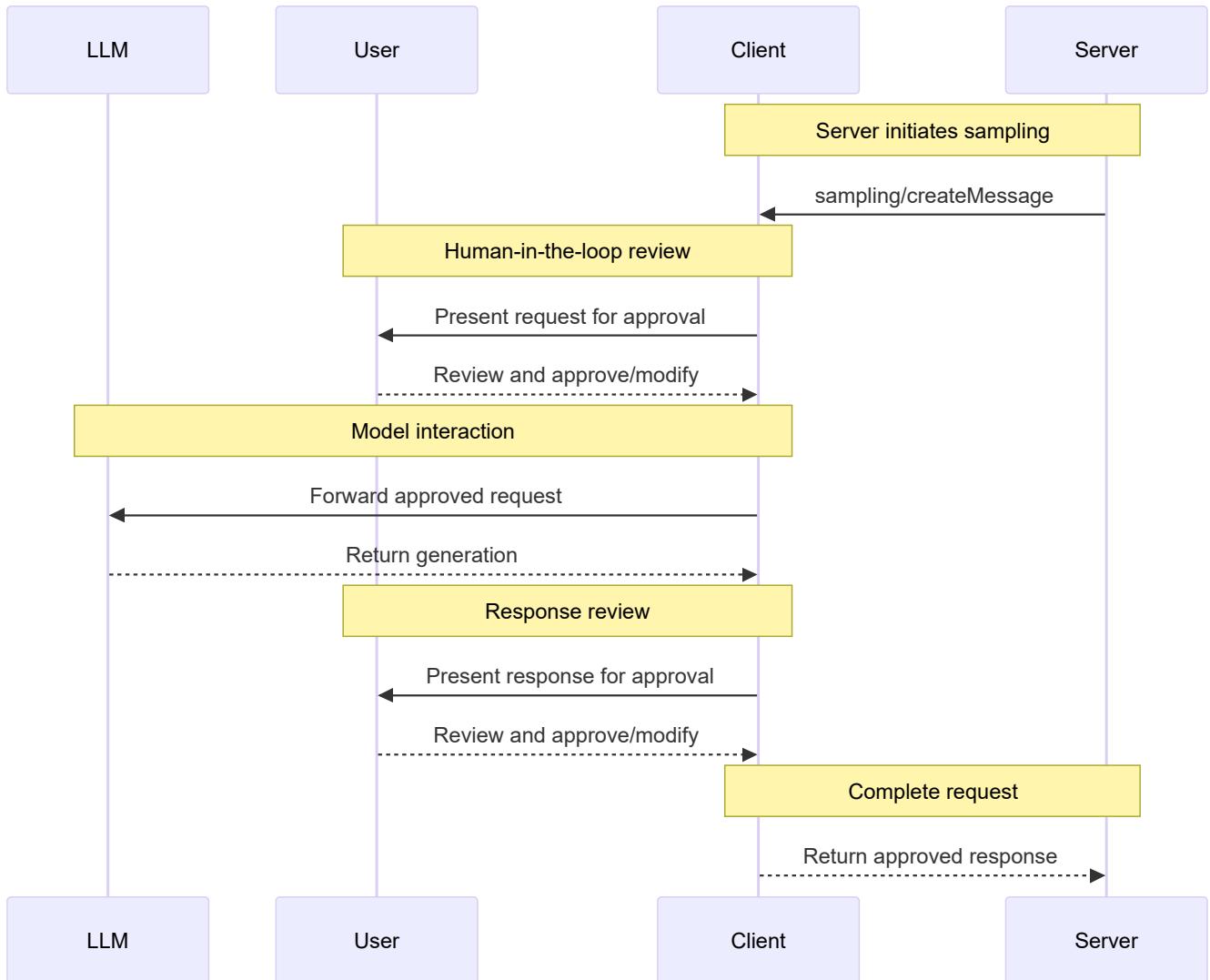
### Sampling

Sampling allows servers to request language model completions through the client, enabling agentic behaviors while maintaining security and user control.

#### Overview

Sampling enables servers to perform AI-dependent tasks without directly integrating with or paying for AI models. Instead, servers can request that the client—which already has AI model access—handle these tasks on their behalf. This approach puts the client in complete control of user permissions and security measures. Because sampling requests occur within the context of other operations—like a tool analyzing data—and are processed as separate model calls, they maintain clear boundaries between different contexts, allowing for more efficient use of the context window.

#### Sampling flow:



The flow ensures security through multiple human-in-the-loop checkpoints. Users review and can modify both the initial request and the generated response before it returns to the server.

#### Request parameters example:

```
{
  messages: [
    {
      role: "user",
      content: "Analyze these flight options and recommend the best choice:\n" +
              "[47 flights with prices, times, airlines, and layovers]\n" +
              "User preferences: morning departure, max 1 layover"
    }
  ],
  modelPreferences: {
    hints: [
      name: "claude-3-5-sonnet" // Suggested model
    ],
    costPriority: 0.3,        // Less concerned about API cost
    speedPriority: 0.2,       // Can wait for thorough analysis
    intelligencePriority: 0.9 // Need complex trade-off evaluation
  },
}
```

```
systemPrompt: "You are a travel expert helping users find the best flights based on their preferences",
maxTokens: 1500
}
```

## Example: Flight Analysis Tool

Consider a travel booking server with a tool called `findBestFlight` that uses sampling to analyze available flights and recommend the optimal choice. When a user asks "Book me the best flight to Barcelona next month," the tool needs AI assistance to evaluate complex trade-offs.

The tool queries airline APIs and gathers 47 flight options. It then requests AI assistance to analyze these options: "Analyze these flight options and recommend the best choice: [47 flights with prices, times, airlines, and layovers] User preferences: morning departure, max 1 layover."

The client asks the user: "Allow sampling request?" Upon approval, the AI evaluates trade-offs—like cheaper red-eye flights versus convenient morning departures. The tool uses this analysis to present the top three recommendations.

## User Interaction Model

Sampling is designed with human-in-the-loop control as a fundamental principle. Users maintain oversight through several mechanisms:

**Approval controls:** Every sampling request needs explicit user consent. Clients show what the server wants to analyze and why. Users can approve, deny, or modify requests.

**Transparency features:** Clients display the exact prompt, model selection, and token limits. Users review AI responses before they return to the server.

**Configuration options:** Users can set model preferences, configure auto-approval for trusted operations, or require approval for everything. Clients may provide options to redact sensitive information. Users decide how much conversation context may be included in sampling requests through the `includeContext` parameter.

**Isolation:** Sampling requests are isolated from the main conversation context by default. Servers cannot access user conversations.

**Security considerations:** Both clients and servers must handle sensitive data appropriately during sampling. Clients should implement rate limiting and validate all message content. The human-in-the-loop design ensures that server-initiated AI interactions cannot compromise security or access sensitive data without explicit user consent.

## Roots

Roots define filesystem boundaries for server operations, allowing clients to specify which directories servers should focus on.

### Overview

Roots are a mechanism for clients to communicate filesystem access boundaries to servers. They consist of file URLs that indicate directories where servers can operate, helping servers understand the scope of available files and folders. Rather than giving servers unrestricted filesystem access, roots guide them to relevant working directories while maintaining security boundaries.

#### Root structure:

```
{  
  "uri": "file:///users/agent/travel-planning",  
  "name": "Travel Planning workspace"  
}
```

Roots are exclusively filesystem paths and always use the `file://` URI scheme. They help servers understand project boundaries, workspace organization, and accessible directories. The roots list can be updated dynamically as users work with different projects or folders, with servers receiving notifications through `roots/list_changed` when boundaries change.

It's important to note that while roots provide guidance to servers about where to operate, the client is always in full control of file access. Roots simply communicate intended boundaries—actual file access is always mediated by the client's security policies.

### Example: Travel Planning Workspace

A travel agent working with multiple client trips benefits from roots to organize filesystem access. Consider a workspace with different directories for various aspects of travel planning.

The client provides filesystem roots to the travel planning server:

- `file:///users/agent/travel-planning` - Main workspace containing all travel files
- `file:///users/agent/travel-templates` - Reusable itinerary templates and resources
- `file:///users/agent/client-documents` - Client passports and travel documents

When the agent creates a Barcelona itinerary, the server works within these boundaries—accessing templates, saving the new itinerary, and referencing client documents. It cannot access files outside these roots. Servers typically access files within roots by using relative paths from the root directories or by utilizing file search tools that respect the root boundaries.

If the agent opens an archive folder like `file:///users/agent/archive/2023-trips`, the client updates the roots list via `roots/list_changed`.

### User Interaction Model

Roots are typically managed automatically by host applications based on user actions, though some applications may expose manual root management:

**Automatic root detection:** When users open folders, clients automatically expose them as roots. Opening a travel workspace gives servers access to itineraries and documents within that directory.

**Manual root configuration:** Advanced users can specify roots through configuration. For example, adding `/travel-templates` for reusable resources while excluding directories with financial records.

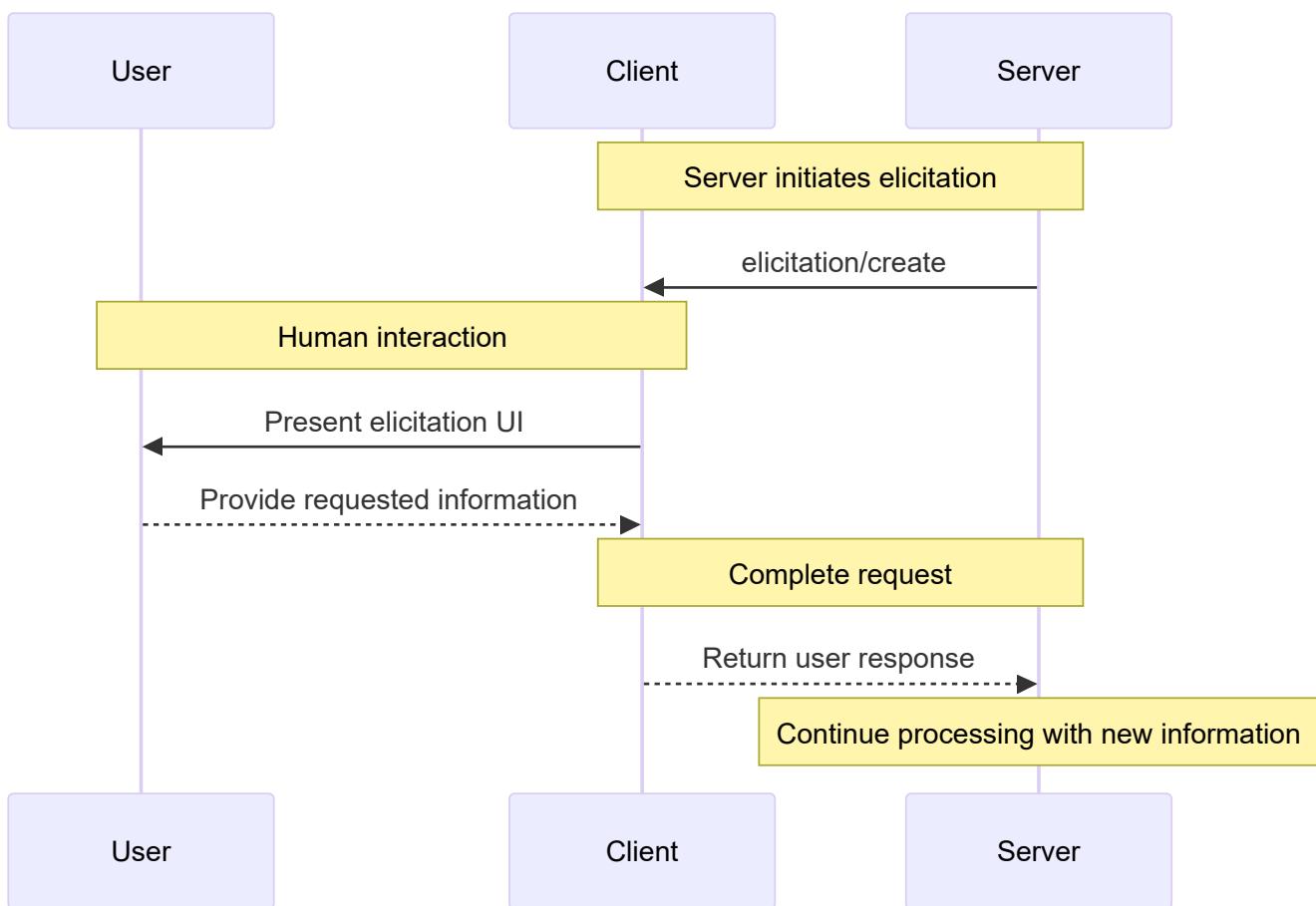
### Elicitation

Elicitation enables servers to request specific information from users during interactions, creating more dynamic and responsive workflows.

## Overview

Elicitation provides a structured way for servers to gather necessary information on demand. Instead of requiring all information up front or failing when data is missing, servers can pause their operations to request specific inputs from users. This creates more flexible interactions where servers adapt to user needs rather than following rigid patterns.

## Elicitation flow:



The flow enables dynamic information gathering. Servers can request specific data when needed, users provide information through appropriate UI, and servers continue processing with the newly acquired context.

## Elicitation components example:

```
{  
  method: "elicitation/requestInput",  
  params: {  
    message: "Please confirm your Barcelona vacation booking details:",  
    schema: {  
      type: "object",  
      properties: {  
        confirmBooking: {  
          type: "boolean",  
          description: "Confirm the booking (Flights + Hotel = $3,000)"  
        },  
        seatPreference: {  
          type: "string",  
          enum: ["window", "aisle", "no preference"]  
        }  
      }  
    }  
  }  
}
```

```

        description: "Preferred seat type for flights"
    },
    roomType: {
        type: "string",
        enum: ["sea view", "city view", "garden view"],
        description: "Preferred room type at hotel"
    },
    travelInsurance: {
        type: "boolean",
        default: false,
        description: "Add travel insurance ($150)"
    }
},
required: ["confirmBooking"]
}
}
}

```

## Example: Holiday Booking Approval

A travel booking server demonstrates elicitation's power through the final booking confirmation process. When a user has selected their ideal vacation package to Barcelona, the server needs to gather final approval and any missing details before proceeding.

The server elicits booking confirmation with a structured request that includes the trip summary (Barcelona flights June 15-22, beachfront hotel, total \$3,000) and fields for any additional preferences—such as seat selection, room type, or travel insurance options.

As the booking progresses, the server elicits contact information needed to complete the reservation. It might ask for traveler details for flight bookings, special requests for the hotel, or emergency contact information.

## User Interaction Model

Elicitation interactions are designed to be clear, contextual, and respectful of user autonomy:

**Request presentation:** Clients display elicitation requests with clear context about which server is asking, why the information is needed, and how it will be used. The request message explains the purpose while the schema provides structure and validation.

**Response options:** Users can provide the requested information through appropriate UI controls (text fields, dropdowns, checkboxes), decline to provide information with optional explanation, or cancel the entire operation. Clients validate responses against the provided schema before returning them to servers.

**Privacy considerations:** Elicitation never requests passwords or API keys. Clients warn about suspicious requests and let users review data before sending.

# Versioning

---

The Model Context Protocol uses string-based version identifiers following the format `YYYY-MM-DD`, to indicate the last date backwards incompatible changes were made.

The protocol version will *not* be incremented when the protocol is updated, as long as the changes maintain backwards compatibility. This allows for incremental improvements while preserving interoperability.

## Revisions

Revisions may be marked as:

- **Draft**: in-progress specifications, not yet ready for consumption.
- **Current**: the current protocol version, which is ready for use and may continue to receive backwards compatible changes.
- **Final**: past, complete specifications that will not be changed.

The **current** protocol version is [2025-06-18](#).

## Negotiation

Version negotiation happens during [initialization](#). Clients and servers **MAY** support multiple protocol versions simultaneously, but they **MUST** agree on a single version to use for the session.

The protocol provides appropriate error handling if version negotiation fails, allowing clients to gracefully terminate connections when they cannot find a version compatible with the server.

# Connect to Remote MCP Servers

Learn how to connect Claude to remote MCP servers and extend its capabilities with internet-hosted tools and data sources

Remote MCP servers extend AI applications' capabilities beyond your local environment, providing access to internet-hosted tools, services, and data sources. By connecting to remote MCP servers, you transform AI assistants from helpful tools into informed teammates capable of handling complex, multi-step projects with real-time access to external resources.

Many clients now support remote MCP servers, enabling a wide range of integration possibilities. This guide demonstrates how to connect to remote MCP servers using [Claude](#) as an example, one of the [many clients that support MCP](#). While we focus on Claude's implementation through Custom Connectors, the concepts apply broadly to other MCP-compatible clients.

## Understanding Remote MCP Servers

Remote MCP servers function similarly to local MCP servers but are hosted on the internet rather than your local machine. They expose tools, prompts, and resources that Claude can use to perform tasks on your behalf. These servers can integrate with various services such as project management tools, documentation systems, code repositories, and any other API-enabled service.

The key advantage of remote MCP servers is their accessibility. Unlike local servers that require installation and configuration on each device, remote servers are available from any MCP client with an internet connection. This makes them ideal for web-based AI applications, integrations that emphasize ease-of-use and services that require server-side processing or authentication.

## What are Custom Connectors?

Custom Connectors serve as the bridge between Claude and remote MCP servers. They allow you to connect Claude directly to the tools and data sources that matter most to your workflows, enabling Claude to operate within your favorite software and draw insights from the complete context of your external tools.

With Custom Connectors, you can:

- [Connect Claude to existing remote MCP servers](#) provided by third-party developers
- [Build your own remote MCP servers to connect with any tool](#)

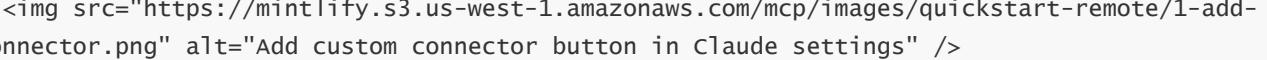
## Connecting to a Remote MCP Server

The process of connecting Claude to a remote MCP server involves adding a Custom Connector through the [Claude interface](#). This establishes a secure connection between Claude and your chosen remote server.

Open Claude in your browser and navigate to the settings page. You can access this by clicking on your profile icon and selecting "Settings" from the dropdown menu. Once in settings, locate and click on the "Connectors" section in the sidebar.

This will display your currently configured connectors and provide options to add new ones.

In the Connectors section, scroll to the bottom where you'll find the "Add custom connector" button. Click this button to begin the connection process.

```
<Frame>

</Frame>
```

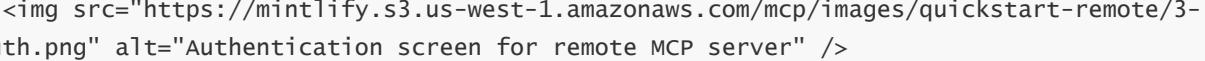
A dialog will appear prompting you to enter the remote MCP server URL. This URL should be provided by the server developer or administrator. Enter the complete URL, ensuring it includes the proper protocol (`https://`) and any necessary path components.

```
<Frame>

</Frame>
```

After entering the URL, click "Add" to proceed with the connection.

Most remote MCP servers require authentication to ensure secure access to their resources. The authentication process varies depending on the server implementation but commonly involves OAuth, API keys, or username/password combinations.

```
<Frame>

</Frame>
```

Follow the authentication prompts provided by the server. This may redirect you to a third-party authentication provider or display a form within Claude. Once authentication is complete, Claude will establish a secure connection to the remote server.

After successful connection, the remote server's resources and prompts become available in your Claude conversations. You can access these by clicking the paperclip icon in the message input area, which opens the attachment menu.

```
<Frame>
  
</Frame>
```

The menu displays all available resources and prompts from your connected servers. Select the items you want to include in your conversation. These resources provide Claude with context and information from your external tools.

```
<Frame>
  
</Frame>
```

Remote MCP servers often expose multiple tools with varying capabilities. You can control which tools Claude is allowed to use by configuring permissions in the connector settings. This ensures Claude only performs actions you've explicitly authorized.

```
<Frame>
  
</Frame>
```

Navigate back to the Connectors settings and click on your connected server. Here you can enable or disable specific tools, set usage limits, and configure other security parameters according to your needs.

## Best Practices for Using Remote MCP Servers

When working with remote MCP servers, consider these recommendations to ensure a secure and efficient experience:

**Security considerations:** Always verify the authenticity of remote MCP servers before connecting. Only connect to servers from trusted sources, and review the permissions requested during authentication. Be cautious about granting access to sensitive data or systems.

**Managing multiple connectors:** You can connect to multiple remote MCP servers simultaneously. Organize your connectors by purpose or project to maintain clarity. Regularly review and remove connectors you no longer use to keep your workspace organized and secure.

## Next Steps

Now that you've connected Claude to a remote MCP server, you can explore its capabilities in your conversations. Try using the connected tools to automate tasks, access external data, or integrate with your existing workflows.

Create custom remote MCP servers to integrate with proprietary tools and services

Browse our collection of official and community-created MCP servers

Learn how to connect Claude Desktop to local MCP servers for direct system access

Dive deeper into how MCP works and its architecture

Remote MCP servers unlock powerful possibilities for extending Claude's capabilities. As you become familiar with these integrations, you'll discover new ways to streamline your workflows and accomplish complex tasks more efficiently.

# Connect to Local MCP Servers

---

Learn how to extend Claude Desktop with local MCP servers to enable file system access and other powerful integrations

Model Context Protocol (MCP) servers extend AI applications' capabilities by providing secure, controlled access to local resources and tools. Many clients support MCP, enabling diverse integration possibilities across different platforms and applications.

This guide demonstrates how to connect to local MCP servers using Claude Desktop as an example, one of the [many clients that support MCP](#). While we focus on Claude Desktop's implementation, the concepts apply broadly to other MCP-compatible clients. By the end of this tutorial, Claude will be able to interact with files on your computer, create new documents, organize folders, and search through your file system—all with your explicit permission for each action.



## Prerequisites

Before starting this tutorial, ensure you have the following installed on your system:

## Claude Desktop

Download and install [Claude Desktop](#) for your operating system. Claude Desktop is currently available for macOS and Windows. Linux support is coming soon.

If you already have Claude Desktop installed, verify you're running the latest version by clicking the Claude menu and selecting "Check for Updates..."

## Node.js

The Filesystem Server and many other MCP servers require Node.js to run. Verify your Node.js installation by opening a terminal or command prompt and running:

```
node --version
```

If Node.js is not installed, download it from [nodejs.org](#). We recommend the LTS (Long Term Support) version for stability.

## Understanding MCP Servers

MCP servers are programs that run on your computer and provide specific capabilities to Claude Desktop through a standardized protocol. Each server exposes tools that Claude can use to perform actions, with your approval. The Filesystem Server we'll install provides tools for:

- Reading file contents and directory structures
- Creating new files and directories
- Moving and renaming files
- Searching for files by name or content

All actions require your explicit approval before execution, ensuring you maintain full control over what Claude can access and modify.

## Installing the Filesystem Server

The process involves configuring Claude Desktop to automatically start the Filesystem Server whenever you launch the application. This configuration is done through a JSON file that tells Claude Desktop which servers to run and how to connect to them.

Start by accessing the Claude Desktop settings. Click on the Claude menu in your system's menu bar (not the settings within the Claude window itself) and select "Settings..."

On macOS, this appears in the top menu bar:

```
<Frame style={{ textAlign: "center" }}>
  
</Frame>
```

This opens the Claude Desktop configuration window, which is separate from your Claude account settings.

In the Settings window, navigate to the "Developer" tab in the left sidebar. This section contains options for configuring MCP servers and other developer features.

Click the "Edit Config" button to open the configuration file:

```
<Frame>
  
</Frame>
```

This action creates a new configuration file if one doesn't exist, or opens your existing configuration. The file is located at:

```
* **macOS**: `~/Library/Application Support/Claude/clause_desktop_config.json`
* **Windows**: `%APPDATA%\Claude\clause_desktop_config.json`
```

Replace the contents of the configuration file with the following JSON structure. This configuration tells Claude Desktop to start the Filesystem Server with access to specific directories:

```
<CodeGroup>
  ````json macos
  {
    "mcpServers": {
      "filesystem": {
        "command": "npx",
        "args": [
          "-y",
          "@modelcontextprotocol/server-filesystem",
          "/Users/username/Desktop",
          "/Users/username/Downloads"
        ]
      }
    }
  }
  ````
```

```

```json windows
{
  "mcpServers": {
    "filesystem": {
      "command": "npx",
      "args": [
        "-y",
        "@modelcontextprotocol/server-filesystem",
        "C:\\\\users\\\\username\\\\Desktop",
        "C:\\\\users\\\\username\\\\Downloads"
      ]
    }
  }
}
```

```

</CodeGroup>

Replace `username` with your actual computer username. The paths listed in the `args` array specify which directories the Filesystem Server can access. You can modify these paths or add additional directories as needed.

<Tip>

\*\*Understanding the Configuration\*\*

- \* `filesystem`: A friendly name for the server that appears in Claude Desktop
- \* `command`: `npx`: Uses Node.js's npx tool to run the server
- \* `-y`: Automatically confirms the installation of the server package
- \* `@modelcontextprotocol/server-filesystem`: The package name of the Filesystem Server
- \* The remaining arguments: Directories the server is allowed to access

</Tip>

<Warning>

\*\*Security Consideration\*\*

only grant access to directories you're comfortable with Claude reading and modifying. The server runs with your user account permissions, so it can perform any file operations you can perform manually.

</Warning>

After saving the configuration file, completely quit Claude Desktop and restart it. The application needs to restart to load the new configuration and start the MCP server.

Upon successful restart, you'll see an MCP server indicator  in the bottom-right corner of the conversation input box:

```
<Frame>
  
</Frame>
```

Click on this indicator to view the available tools provided by the Filesystem Server:

```
<Frame style={{ textAlign: "center" }}>
  
</Frame>
```

If the server indicator doesn't appear, refer to the [Troubleshooting] (#troubleshooting) section for debugging steps.

## Using the Filesystem Server

With the Filesystem Server connected, Claude can now interact with your file system. Try these example requests to explore the capabilities:

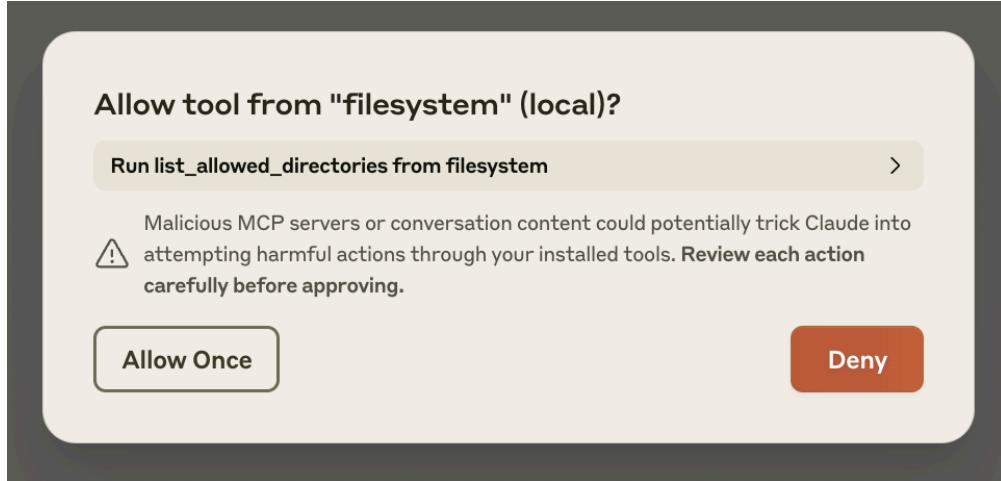
### File Management Examples

- "**Can you write a poem and save it to my desktop?**" - Claude will compose a poem and create a new text file on your desktop
- "**What work-related files are in my downloads folder?**" - Claude will scan your downloads and identify work-related documents
- "**Please organize all images on my desktop into a new folder called 'Images'**" - Claude will create a folder and move image files into it

### How Approval Works

Before executing any file system operation, Claude will request your approval. This ensures you maintain control over all actions:

```
<Frame style={{ textAlign: "center" }}>
```



Review each request carefully before approving. You can always deny a request if you're not comfortable with the proposed action.

## Troubleshooting

If you encounter issues setting up or using the Filesystem Server, these solutions address common problems:

1. Restart Claude Desktop completely
2. Check your `claude_desktop_config.json` file syntax
3. Make sure the file paths included in `claude_desktop_config.json` are valid and that they are absolute and not relative
4. Look at [logs](#) to see why the server is not connecting
5. In your command line, try manually running the server (replacing `username` as you did in `claude_desktop_config.json`) to see if you get any errors:

```
<CodeGroup>
  ````bash macos/Linux
  npx -y @modelcontextprotocol/server-filesystem /Users/username/Desktop
  /Users/username/Downloads
  ````

  ````powershell windows
  npx -y @modelcontextprotocol/server-filesystem C:\Users\username\Desktop
  C:\Users\username\Downloads
  ````

</CodeGroup>
```

Claude.app logging related to MCP is written to log files in:

```
* macos: `~/Library/Logs/Claude`
* Windows: `%APPDATA%\claude\logs`
```

```
* `mcp.log` will contain general logging about MCP connections and connection failures.  
* Files named `mcp-server-SERVERNAME.log` will contain error (stderr) logging from the named server.
```

You can run the following command to list recent logs and follow along with any new ones (on Windows, it will only show recent logs):

```
<CodeGroup>  
  ```bash macos/Linux  
  tail -n 20 -f ~/Library/Logs/Claude/mcp*.log  
  ...  
  
  ```powershell windows  
  type "%APPDATA%\Claude\logs\mcp*.log"  
  ...  
</CodeGroup>
```

If Claude attempts to use the tools but they fail:

1. Check Claude's logs for errors
2. Verify your server builds and runs without errors
3. Try restarting Claude Desktop

Please refer to our [debugging guide](#) for better debugging tools and more detailed guidance.

If your configured server fails to load, and you see within its logs an error referring to `\${APPDATA}` within a path, you may need to add the expanded value of `%APPDATA%` to your `env` key in `claude_desktop_config.json`:

```
```json  
{  
  "brave-search": {  
    "command": "npx",  
    "args": ["-y", "@modelcontextprotocol/server-brave-search"],  
    "env": {  
      "APPDATA": "C:\\\\users\\\\user\\\\AppData\\\\Roaming\\\\",  
      "BRAVE_API_KEY": "..."  
    }  
  }  
}
```

with this change in place, launch Claude Desktop once again.

<warning>

\*\*NPM should be installed globally\*\*

The `npx` command may continue to fail if you have not installed NPM globally. If NPM is already installed globally, you will find `%APPDATA%\npm` exists on your system. If not, you can install NPM globally by running the following command:

```
```bash
npm install -g npm
````
```

</warning>

## Next Steps

Now that you've successfully connected Claude Desktop to a local MCP server, explore these options to expand your setup:

Browse our collection of official and community-created MCP servers for additional capabilities

Create custom MCP servers tailored to your specific workflows and integrations

Learn how to connect Claude to remote MCP servers for cloud-based tools and services

Dive deeper into how MCP works and its architecture

# Build an MCP Server

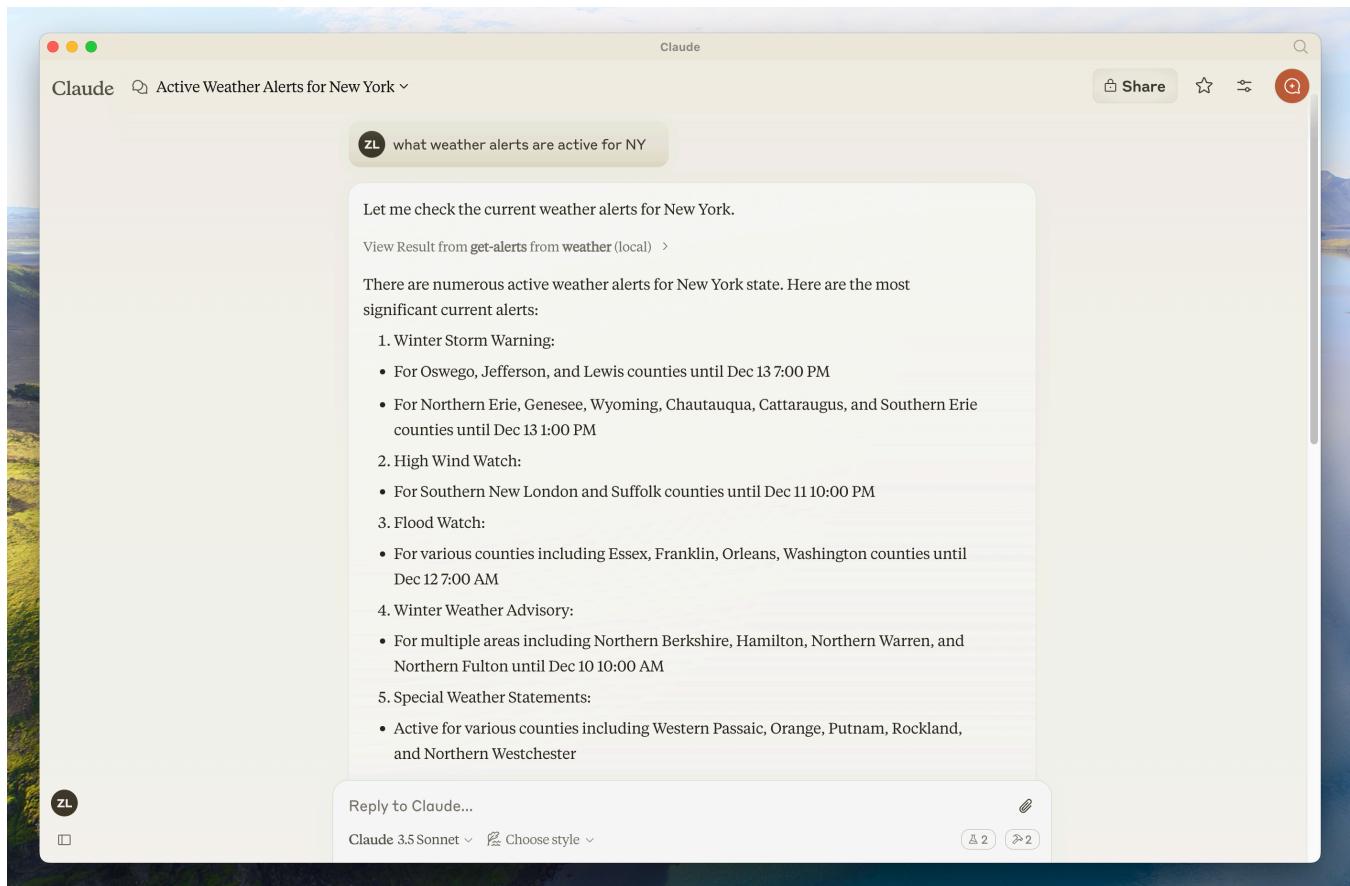
Get started building your own server to use in Claude for Desktop and other clients.

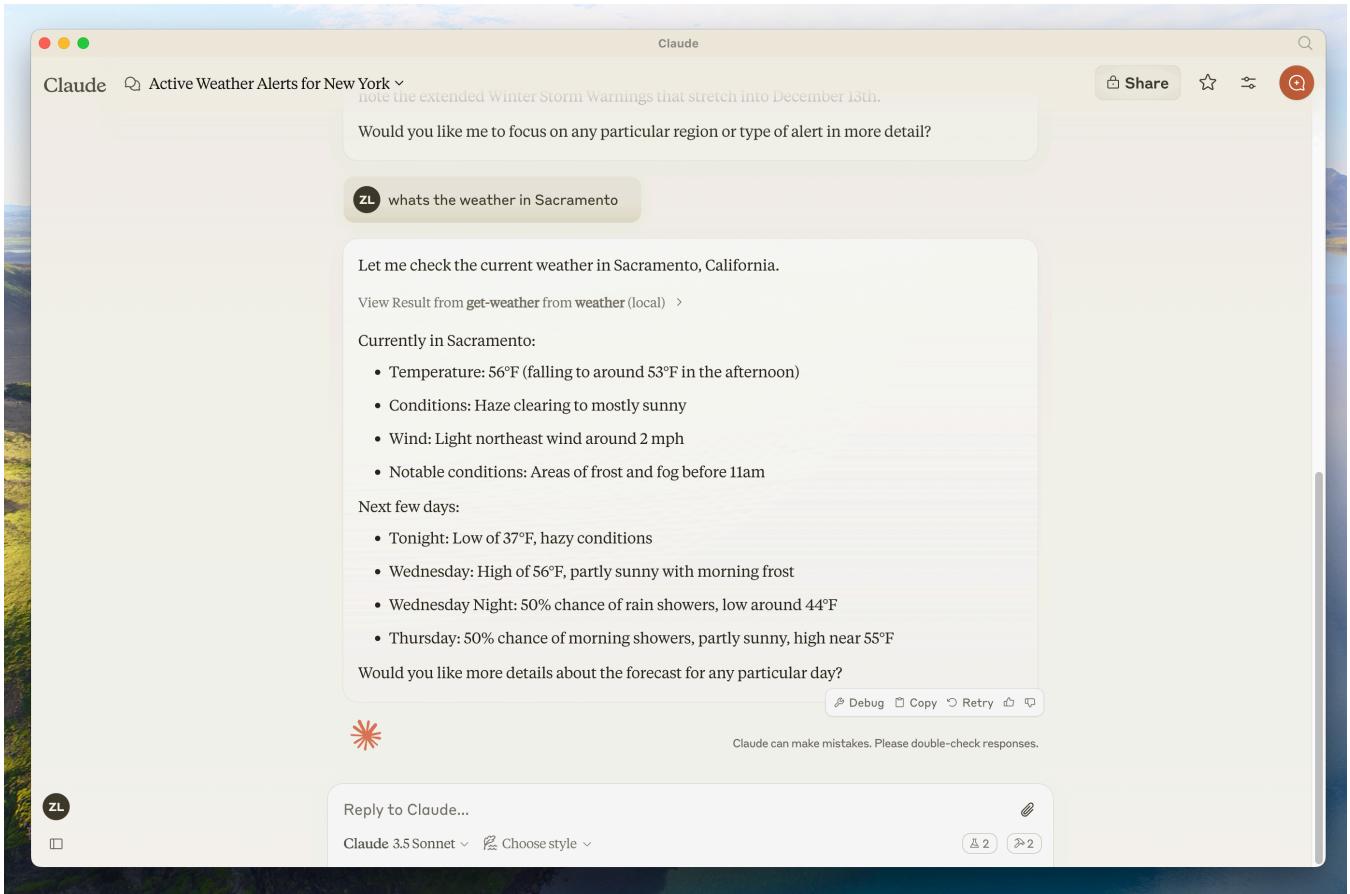
In this tutorial, we'll build a simple MCP weather server and connect it to a host, Claude for Desktop. We'll start with a basic setup, and then progress to more complex use cases.

## What we'll be building

Many LLMs do not currently have the ability to fetch the forecast and severe weather alerts. Let's use MCP to solve that!

We'll build a server that exposes two tools: `get_alerts` and `get_forecast`. Then we'll connect the server to an MCP host (in this case, Claude for Desktop):





Servers can connect to any client. We've chosen Claude for Desktop here for simplicity, but we also have guides on [building your own client](#) as well as a [list of other clients here](#).

## Core MCP Concepts

MCP servers can provide three main types of capabilities:

1. **Resources:** File-like data that can be read by clients (like API responses or file contents)
2. **Tools:** Functions that can be called by the LLM (with user approval)
3. **Prompts:** Pre-written templates that help users accomplish specific tasks

This tutorial will primarily focus on tools.

Let's get started with building our weather server! [You can find the complete code for what we'll be building here.](#)

### #### Prerequisite knowledge

This quickstart assumes you have familiarity with:

- \* Python
- \* LLMs like Claude

#### #### Logging in MCP Servers

when implementing MCP servers, be careful about how you handle logging:

\*\*For STDIO-based servers:\*\* Never write to standard output (stdout). This includes:

- \* `print()` statements in Python
- \* `console.log()` in JavaScript
- \* `fmt.Println()` in Go
- \* Similar stdout functions in other languages

writing to stdout will corrupt the JSON-RPC messages and break your server.

\*\*For HTTP-based servers:\*\* Standard output logging is fine since it doesn't interfere with HTTP responses.

#### #### Best Practices

1. Use a logging library that writes to stderr or files.

#### #### Quick Examples

```
```python
# ❌ Bad (STDIO)
print("Processing request")

# ✅ Good (STDIO)
import logging
logging.info("Processing request")
```

```

#### #### System requirements

- \* Python 3.10 or higher installed.
- \* You must use the Python MCP SDK 1.2.0 or higher.

#### #### Set up your environment

First, let's install `uv` and set up our Python project and environment:

```
<CodeGroup>
  ```bash macos/Linux
  curl -Lsf https://astral.sh/uv/install.sh | sh
  ```

  ```powershell windows
  powershell -ExecutionPolicy Bypass -c "irm https://astral.sh/uv/install.ps1 | iex"
  ```

</CodeGroup>
```

Make sure to restart your terminal afterwards to ensure that the `uv` command gets picked up.

Now, let's create and set up our project:

```
<CodeGroup>
  ```bash macos/Linux
  # Create a new directory for our project
  uv init weather
  cd weather

  # Create virtual environment and activate it
  uv venv
  source .venv/bin/activate

  # Install dependencies
  uv add "mcp[cli]" httpx

  # Create our server file
  touch weather.py
  ```

  ```powershell windows
  # Create a new directory for our project
  uv init weather
  cd weather

  # Create virtual environment and activate it
  uv venv
  .venv\Scripts\activate

  # Install dependencies
  uv add mcp[cli] httpx

  # Create our server file
  new-item weather.py
  ```

</CodeGroup>
```

Now let's dive into building your server.

### Building your server

#### Importing packages and setting up the instance

Add these to the top of your `weather.py`:

```
```python
from typing import Any
import httpx
from mcp.server.fastmcp import FastMCP

# Initialize FastMCP server
mcp = FastMCP("weather")

# Constants
```

```
NWS_API_BASE = "https://api.weather.gov"
USER_AGENT = "weather-app/1.0"
```
```

The FastMCP class uses Python type hints and docstrings to automatically generate tool definitions, making it easy to create and maintain MCP tools.

#### #### Helper functions

Next, let's add our helper functions for querying and formatting the data from the National Weather Service API:

```
```python
async def make_nws_request(url: str) -> dict[str, Any] | None:
    """Make a request to the NWS API with proper error handling."""
    headers = {
        "User-Agent": USER_AGENT,
        "Accept": "application/geo+json"
    }
    async with httpx.AsyncClient() as client:
        try:
            response = await client.get(url, headers=headers, timeout=30.0)
            response.raise_for_status()
            return response.json()
        except Exception:
            return None

def format_alert(feature: dict) -> str:
    """Format an alert feature into a readable string."""
    props = feature["properties"]
    return f"""
Event: {props.get('event', 'Unknown')}
Area: {props.get('areaDesc', 'Unknown')}
Severity: {props.get('severity', 'Unknown')}
Description: {props.get('description', 'No description available')}
Instructions: {props.get('instruction', 'No specific instructions provided')}
"""
```
```

```

#### #### Implementing tool execution

The tool execution handler is responsible for actually executing the logic of each tool. Let's add it:

```
```python
@mcp.tool()
async def get_alerts(state: str) -> str:
    """Get weather alerts for a US state.

    Args:
        state: Two-letter US state code (e.g. CA, NY)
    """
    url = f"{NWS_API_BASE}/alerts/active/area/{state}"
```

```

```

data = await make_nws_request(url)

if not data or "features" not in data:
    return "Unable to fetch alerts or no alerts found."

if not data["features"]:
    return "No active alerts for this state."

alerts = [format_alert(feature) for feature in data["features"]]
return "\n---\n".join(alerts)

@mcp.tool()
async def get_forecast(latitude: float, longitude: float) -> str:
    """Get weather forecast for a location.

    Args:
        latitude: Latitude of the location
        longitude: Longitude of the location
    """
    # First get the forecast grid endpoint
    points_url = f"{NWS_API_BASE}/points/{latitude},{longitude}"
    points_data = await make_nws_request(points_url)

    if not points_data:
        return "Unable to fetch forecast data for this location."

    # Get the forecast URL from the points response
    forecast_url = points_data["properties"]["forecast"]
    forecast_data = await make_nws_request(forecast_url)

    if not forecast_data:
        return "Unable to fetch detailed forecast."

    # Format the periods into a readable forecast
    periods = forecast_data["properties"]["periods"]
    forecasts = []
    for period in periods[:5]: # only show next 5 periods
        forecast = f"""
{period['name']}:
Temperature: {period['temperature']}°{period['temperatureUnit']}
Wind: {period['windSpeed']} {period['windDirection']}
Forecast: {period['detailedForecast']}
"""
        forecasts.append(forecast)

    return "\n---\n".join(forecasts)
```
#### Running the server

Finally, let's initialize and run the server:

```python

```

```
if __name__ == "__main__":
    # Initialize and run the server
    mcp.run(transport='stdio')
```
```

Your server is complete! Run `uv run weather.py` to start the MCP server, which will listen for messages from MCP hosts.

Let's now test your server from an existing MCP host, Claude for Desktop.

### Testing your server with Claude for Desktop

<Note>

Claude for Desktop is not yet available on Linux. Linux users can proceed to the [Building a client](/quickstart/client) tutorial to build an MCP client that connects to the server we just built.

</Note>

First, make sure you have Claude for Desktop installed. [You can install the latest version here.](<https://claude.ai/download>) If you already have Claude for Desktop, \*\*make sure it's updated to the latest version.\*\*

We'll need to configure Claude for Desktop for whichever MCP servers you want to use. To do this, open your Claude for Desktop App configuration at `~/Library/Application Support/claude/claude\_desktop\_config.json` in a text editor. Make sure to create the file if it doesn't exist.

For example, if you have [vs Code](<https://code.visualstudio.com/>) installed:

```
<CodeGroup>
```bash macos/Linux
code ~/Library/Application\ Support/claude/claude_desktop_config.json
```

```powershell windows
code $env:AppData\Claude\claude_desktop_config.json
```

</CodeGroup>
```

You'll then add your servers in the `mcpServers` key. The MCP UI elements will only show up in Claude for Desktop if at least one server is properly configured.

In this case, we'll add our single weather server like so:

```
<CodeGroup>
```json macos/Linux
{
  "mcpServers": {
    "weather": {
      "command": "uv",
      "args": [
        "--directory",
        "/ABSOLUTE/PATH/TO/PARENT/FOLDER/weather",
```
```

```

        "run",
        "weather.py"
    ]
}
}
```
```
json windows
{
  "mcpServers": {
    "weather": {
      "command": "uv",
      "args": [
        "--directory",
        "C:\\\\ABSOLUTE\\\\PATH\\\\TO\\\\PARENT\\\\FOLDER\\\\weather",
        "run",
        "weather.py"
      ]
    }
  }
}
```
```

```

</CodeGroup>

<Warning>

You may need to put the full path to the `uv` executable in the `command` field. You can get this by running `which uv` on macOS/Linux or `where uv` on Windows.

</Warning>

<Note>

Make sure you pass in the absolute path to your server. You can get this by running `pwd` on macOS/Linux or `cd` on Windows Command Prompt. On Windows, remember to use double backslashes (`\\`) or forward slashes (`/`) in the JSON path.

</Note>

This tells Claude for Desktop:

1. There's an MCP server named "weather"
2. To launch it by running `uv --directory /ABSOLUTE/PATH/TO/PARENT/FOLDER/weather run weather.py`

Save the file, and restart \*\*Claude for Desktop\*\*.

Let's get started with building our weather server! [You can find the complete code for what we'll be building here.](#)

#### Prerequisite knowledge

This quickstart assumes you have familiarity with:

- \* TypeScript
- \* LLMs like Claude

#### #### Logging in MCP Servers

When implementing MCP servers, be careful about how you handle logging:

\*\*For STDIO-based servers:\*\* Never write to standard output (stdout). This includes:

- \* `print()` statements in Python
- \* `console.log()` in JavaScript
- \* `fmt.Println()` in Go
- \* Similar stdout functions in other languages

Writing to stdout will corrupt the JSON-RPC messages and break your server.

\*\*For HTTP-based servers:\*\* Standard output logging is fine since it doesn't interfere with HTTP responses.

#### #### Best Practices

1. Use a logging library that writes to stderr or files, such as `logging` in Python.
2. For JavaScript, be especially careful - `console.log()` writes to stdout by default

#### #### Quick Examples

```
```javascript
// ❌ Bad (STDIO)
console.log("Server started");

// ✅ Good (STDIO)
console.error("Server started"); // stderr is safe
```
```

#### #### System requirements

For TypeScript, make sure you have the latest version of Node installed.

#### #### Set up your environment

First, let's install Node.js and npm if you haven't already. You can download them from [nodejs.org](https://nodejs.org/).

Verify your Node.js installation:

```
```bash
node --version
npm --version
```
```

For this tutorial, you'll need Node.js version 16 or higher.

Now, let's create and set up our project:

```

<CodeGroup>
  ```bash macos/Linux
  # Create a new directory for our project
  mkdir weather
  cd weather

  # Initialize a new npm project
  npm init -y

  # Install dependencies
  npm install @modelcontextprotocol/sdk zod
  npm install -D @types/node typescript

  # Create our files
  mkdir src
  touch src/index.ts
  ```

  ```powershell windows
  # Create a new directory for our project
  md weather
  cd weather

  # Initialize a new npm project
  npm init -y

  # Install dependencies
  npm install @modelcontextprotocol/sdk zod
  npm install -D @types/node typescript

  # Create our files
  md src
  new-item src\index.ts
  ```

</CodeGroup>

```

Update your package.json to add type: "module" and a build script:

```

```json package.json
{
  "type": "module",
  "bin": {
    "weather": "./build/index.js"
  },
  "scripts": {
    "build": "tsc && chmod 755 build/index.js"
  },
  "files": ["build"]
}
```

```

Create a `tsconfig.json` in the root of your project:

```
```json tsconfig.json
{
  "compilerOptions": {
    "target": "ES2022",
    "module": "Node16",
    "moduleResolution": "Node16",
    "outDir": "./build",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules"]
}
```

```

Now let's dive into building your server.

### ### Building your server

#### #### Importing packages and setting up the instance

Add these to the top of your `src/index.ts`:

```
```typescript
import { McpServer } from "@modelcontextprotocol/sdk/server/mcp.js";
import { StdioServerTransport } from "@modelcontextprotocol/sdk/server/stdio.js";
import { z } from "zod";

const NWS_API_BASE = "https://api.weather.gov";
const USER_AGENT = "weather-app/1.0";

// Create server instance
const server = new McpServer({
  name: "weather",
  version: "1.0.0",
  capabilities: {
    resources: {},
    tools: {}
  }
});
```

```

#### #### Helper functions

Next, let's add our helper functions for querying and formatting the data from the National Weather Service API:

```
```typescript
// Helper function for making NWS API requests

```

```
async function makeNWSRequest<T>(url: string): Promise<T | null> {
  const headers = {
    "User-Agent": USER_AGENT,
    Accept: "application/geo+json",
  };

  try {
    const response = await fetch(url, { headers });
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }
    return (await response.json()) as T;
  } catch (error) {
    console.error("Error making NWS request:", error);
    return null;
  }
}

interface AlertFeature {
  properties: {
    event?: string;
    areaDesc?: string;
    severity?: string;
    status?: string;
    headline?: string;
  };
}

// Format alert data
function formatAlert(feature: AlertFeature): string {
  const props = feature.properties;
  return [
    `Event: ${props.event || "Unknown"} `,
    `Area: ${props.areaDesc || "Unknown"} `,
    `Severity: ${props.severity || "Unknown"} `,
    `Status: ${props.status || "Unknown"} `,
    `Headline: ${props.headline || "No headline"} `,
    "---",
  ].join("\n");
}

interface ForecastPeriod {
  name?: string;
  temperature?: number;
  temperatureUnit?: string;
  windSpeed?: string;
  windDirection?: string;
  shortForecast?: string;
}

interface AlertsResponse {
  features: AlertFeature[];
}
```

```

interface PointsResponse {
  properties: {
    forecast?: string;
  };
}

interface ForecastResponse {
  properties: {
    periods: ForecastPeriod[];
  };
}
```

```

#### #### Implementing tool execution

The tool execution handler is responsible for actually executing the logic of each tool. Let's add it:

```

```typescript
// Register weather tools
server.tool(
  "get_alerts",
  "Get weather alerts for a state",
  {
    state: z.string().length(2).describe("Two-letter state code (e.g. CA, NY)"),
  },
  async ({ state }) => {
    const stateCode = state.toUpperCase();
    const alertsUrl = `${NWS_API_BASE}/alerts?area=${stateCode}`;
    const alertsData = await makeNWSRequest<AlertsResponse>(alertsUrl);

    if (!alertsData) {
      return {
        content: [
          {
            type: "text",
            text: "Failed to retrieve alerts data",
          },
        ],
      };
    }

    const features = alertsData.features || [];
    if (features.length === 0) {
      return {
        content: [
          {
            type: "text",
            text: `No active alerts for ${stateCode}`,
          },
        ],
      };
    }
  }
)
```

```

```
}

const formattedAlerts = features.map(formatAlert);
const alertsText = `Active alerts for ${stateCode}:\n\n${formattedAlerts.join("\n")}`;

return {
  content: [
    {
      type: "text",
      text: alertsText,
    },
  ],
};

server.tool(
  "get_forecast",
  "Get weather forecast for a location",
  {
    latitude: z.number().min(-90).max(90).describe("Latitude of the location"),
    longitude: z
      .number()
      .min(-180)
      .max(180)
      .describe("Longitude of the location"),
  },
  async ({ latitude, longitude }) => {
    // Get grid point data
    const pointsurl =
`${NWS_API_BASE}/points/${latitude.toFixed(4)},${longitude.toFixed(4)}`;
    const pointsData = await makeNWSRequest<PointsResponse>(pointsurl);

    if (!pointsData) {
      return {
        content: [
          {
            type: "text",
            text: `Failed to retrieve grid point data for coordinates: ${latitude}, ${longitude}. This location may not be supported by the NWS API (only US locations are supported).`,
          },
        ],
      };
    }

    const forecasturl = pointsData.properties?.forecast;
    if (!forecasturl) {
      return {
        content: [
          {
            type: "text",
            text: "Failed to get forecast URL from grid point data",
          },
        ],
      };
    }
  },
);
```

```
        },
      ],
    };
}

// Get forecast data
const forecastData = await makeNWSRequest<ForecastResponse>(forecastUrl);
if (!forecastData) {
  return {
    content: [
      {
        type: "text",
        text: "Failed to retrieve forecast data",
      },
    ],
  };
}

const periods = forecastData.properties?.periods || [];
if (periods.length === 0) {
  return {
    content: [
      {
        type: "text",
        text: "No forecast periods available",
      },
    ],
  };
}

// Format forecast periods
const formattedForecast = periods.map((period: ForecastPeriod) =>
  [
    `${period.name || "Unknown":}`,
    `Temperature: ${period.temperature || "Unknown"}°${period.temperatureUnit || "F"} `,
    `Wind: ${period.windSpeed || "Unknown"} ${period.windDirection || ""}`,
    `${period.shortForecast || "No forecast available"} `,
    "---",
  ].join("\n"),
);

const forecastText = `Forecast for ${latitude},
${longitude}:\n\n${formattedForecast.join("\n")}`;

return {
  content: [
    {
      type: "text",
      text: forecastText,
    },
  ],
};
},
```

```
);  
...  
  
#### Running the server
```

Finally, implement the main function to run the server:

```
```typescript  
async function main() {  
    const transport = new StdioServerTransport();  
    await server.connect(transport);  
    console.error("Weather MCP Server running on stdio");  
}  
  
main().catch((error) => {  
    console.error("Fatal error in main()", error);  
    process.exit(1);  
});  
...  
  
Make sure to run `npm run build` to build your server! This is a very important step in getting your server to connect.
```

Let's now test your server from an existing MCP host, Claude for Desktop.

### Testing your server with Claude for Desktop

<Note>

Claude for Desktop is not yet available on Linux. Linux users can proceed to the [Building a client](/quickstart/client) tutorial to build an MCP client that connects to the server we just built.

</Note>

First, make sure you have Claude for Desktop installed. [You can install the latest version here.](https://claude.ai/download) If you already have Claude for Desktop, \*\*make sure it's updated to the latest version.\*\*

We'll need to configure Claude for Desktop for whichever MCP servers you want to use. To do this, open your Claude for Desktop App configuration at `~/Library/Application Support/claude/claude\_desktop\_config.json` in a text editor. Make sure to create the file if it doesn't exist.

For example, if you have [vs Code](https://code.visualstudio.com/) installed:

```
<CodeGroup>  
  ```bash macos/Linux  
  code ~/Library/Application\ Support/claude/claude_desktop_config.json  
  ...  
  
  ```powershell windows  
  code $env:AppData\Claude\claude_desktop_config.json  
  ...  
</CodeGroup>
```

You'll then add your servers in the `mcpServers` key. The MCP UI elements will only show up in Claude for Desktop if at least one server is properly configured.

In this case, we'll add our single weather server like so:

```
<CodeGroup>
  ```json macos/Linux
  {
    "mcpServers": {
      "weather": {
        "command": "node",
        "args": ["/ABSOLUTE/PATH/TO/PARENT/FOLDER/weather/build/index.js"]
      }
    }
  }
  ```

  ```json windows
  {
    "mcpServers": {
      "weather": {
        "command": "node",
        "args": ["C:\\PATH\\TO\\PARENT\\FOLDER\\weather\\build\\index.js"]
      }
    }
  }
  ```

</CodeGroup>
```

This tells Claude for Desktop:

1. There's an MCP server named "weather"
2. Launch it by running `node /ABSOLUTE/PATH/TO/PARENT/FOLDER/weather/build/index.js`

Save the file, and restart \*\*Claude for Desktop\*\*.

This is a quickstart demo based on Spring AI MCP auto-configuration and boot starters.

To learn how to create sync and async MCP Servers, manually, consult the [Java SDK Server](#) documentation.

Let's get started with building our weather server!

[You can find the complete code for what we'll be building here.](<https://github.com/spring-projects/spring-ai-examples/tree/main/model-context-protocol/weather/starter-stdio-server>)

For more information, see the [MCP Server Boot Starter] (<https://docs.spring.io/spring-ai/reference/api/mcp-server-boot-starter-docs.html>) reference documentation.

For manual MCP Server implementation, refer to the [MCP Server Java SDK documentation](/sdk/java/mcp-server).

#### #### Logging in MCP Servers

When implementing MCP servers, be careful about how you handle logging:

\*\*For STDIO-based servers:\*\* Never write to standard output (stdout). This includes:

- \* `print()` statements in Python
- \* `console.log()` in JavaScript
- \* `fmt.Println()` in Go
- \* Similar stdout functions in other languages

Writing to stdout will corrupt the JSON-RPC messages and break your server.

\*\*For HTTP-based servers:\*\* Standard output logging is fine since it doesn't interfere with HTTP responses.

#### #### Best Practices

1. Use a logging library that writes to stderr or files.
2. Ensure any configured logging library will not write to STDOUT

#### #### System requirements

- \* Java 17 or higher installed.
- \* [Spring Boot 3.3.x](https://docs.spring.io/spring-boot/installing.html) or higher

#### #### Set up your environment

Use the [Spring Initializer](https://start.spring.io/) to bootstrap the project.

You will need to add the following dependencies:

```
<CodeGroup>
  ````xml Maven
  <dependencies>
    <dependency>
      <groupId>org.springframework.ai</groupId>
      <artifactId>spring-ai-starter-mcp-server</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-web</artifactId>
    </dependency>
  </dependencies>
  ````

  ````groovy Gradle
dependencies {
  implementation platform("org.springframework.ai:spring-ai-starter-mcp-server")
```

```
        implementation platform("org.springframework:spring-web")
    }
    ...
</CodeGroup>
```

Then configure your application by setting the application properties:

```
<CodeGroup>
    ```bash application.properties
    spring.main.bannerMode=off
    logging.pattern.console=
    ```

    ```yaml application.yml
    logging:
        pattern:
            console:
    spring:
        main:
            banner-mode: off
    ```

</CodeGroup>
```

The [Server Configuration Properties]([https://docs.spring.io/spring-ai/reference/api/mcp/mcp-server-boot-starter-docs.html#\\_configuration\\_properties](https://docs.spring.io/spring-ai/reference/api/mcp/mcp-server-boot-starter-docs.html#_configuration_properties)) documents all available properties.

Now let's dive into building your server.

### Building your server

#### Weather Service

Let's implement a [WeatherService.java](<https://github.com/spring-projects/spring-ai-examples/blob/main/model-context-protocol/weather/starter-stdio-server/src/main/java/org/springframework/ai/mcp/sample/server/WeatherService.java>) that uses a REST client to query the data from the National Weather Service API:

```
```java
@Service
public class WeatherService {

    private final RestClient restClient;

    public WeatherService() {
        this.restClient = RestClient.builder()
            .baseUrl("https://api.weather.gov")
            .defaultHeader("Accept", "application/geo+json")
            .defaultHeader("User-Agent", "WeatherApiClient/1.0 (your@email.com)")
            .build();
    }

    @Tool(description = "Get weather forecast for a specific latitude/longitude")
}
```

```

public String getWeatherForecastByLocation(
    double latitude, // Latitude coordinate
    double longitude // Longitude coordinate
) {
    // Returns detailed forecast including:
    // - Temperature and unit
    // - Wind speed and direction
    // - Detailed forecast description
}

@Tool(description = "Get weather alerts for a US state")
public String getAlerts(
    @ToolParam(description = "Two-letter US state code (e.g. CA, NY)") String state
) {
    // Returns active alerts including:
    // - Event type
    // - Affected area
    // - Severity
    // - Description
    // - Safety instructions
}

// .....
}
```

```

The `@Service` annotation will auto-register the service in your application context. The Spring AI `@Tool` annotation, making it easy to create and maintain MCP tools.

The auto-configuration will automatically register these tools with the MCP server.

#### #### Create your Boot Application

```

```java
@SpringBootApplication
public class McpServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(McpServerApplication.class, args);
    }

    @Bean
    public ToolCallbackProvider weatherTools(WeatherService weatherService) {
        return MethodToolCallbackProvider.builder().toolObjects(weatherService).build();
    }
}
```

```

Uses the the `MethodToolCallbackProvider` utils to convert the `@Tools` into actionable callbacks used by the MCP server.

#### #### Running the server

```
Finally, let's build the server:
```

```
```bash
./mvnw clean install
```

```

This will generate a `mcp-weather-stdio-server-0.0.1-SNAPSHOT.jar` file within the `target` folder.

Let's now test your server from an existing MCP host, Claude for Desktop.

### Testing your server with Claude for Desktop

<Note>

Claude for Desktop is not yet available on Linux.  
</Note>

First, make sure you have Claude for Desktop installed.

[You can install the latest version here.](<https://claude.ai/download>) If you already have Claude for Desktop, \*\*make sure it's updated to the latest version.\*\*

We'll need to configure Claude for Desktop for whichever MCP servers you want to use. To do this, open your Claude for Desktop App configuration at `~/Library/Application Support/Claude/clade\_desktop\_config.json` in a text editor. Make sure to create the file if it doesn't exist.

For example, if you have [VS Code] (<https://code.visualstudio.com/>) installed:

```
<CodeGroup>
  ```bash macos/Linux
  code ~/Library/Application\ Support/Claude/clade_desktop_config.json
  ```

  ```powershell windows
  code $env:AppData\Claude\clade_desktop_config.json
  ```

</CodeGroup>
```

You'll then add your servers in the `mcpServers` key.

The MCP UI elements will only show up in Claude for Desktop if at least one server is properly configured.

In this case, we'll add our single weather server like so:

```
<CodeGroup>
  ```json macos/Linux
  {
    "mcpServers": {
      "spring-ai-mcp-weather": {
        "command": "java",
        "args": [
          "-Dspring.ai.mcp.server.stdio=true",
          "-jar",

```

```

        "/ABSOLUTE/PATH/TO/PARENT/FOLDER/mcp-weather-stdio-server-0.0.1-SNAPSHOT.jar"
    ]
}
}
```
json Windows
{
  "mcpServers": {
    "spring-ai-mcp-weather": {
      "command": "java",
      "args": [
        "-Dspring.ai.mcp.server.transport=STDIO",
        "-jar",
        "C:\\\\ABSOLUTE\\\\PATH\\\\TO\\\\PARENT\\\\FOLDER\\\\weather\\\\mcp-weather-stdio-server-0.0.1-SNAPSHOT.jar"
      ]
    }
  }
}
```
</CodeGroup>

```

<Note>  
 Make sure you pass in the absolute path to your server.  
</Note>

This tells Claude for Desktop:

1. There's an MCP server named "my-weather-server"
2. To launch it by running `java -jar /ABSOLUTE/PATH/TO/PARENT/FOLDER/mcp-weather-stdio-server-0.0.1-SNAPSHOT.jar`

Save the file, and restart \*\*Claude for Desktop\*\*.

### Testing your server with Java client

#### Create a MCP Client manually

Use the `McpClient` to connect to the server:

```

```java
var stdioParams = ServerParameters.builder("java")
  .args("-jar", "/ABSOLUTE/PATH/TO/PARENT/FOLDER/mcp-weather-stdio-server-0.0.1-SNAPSHOT.jar")
  .build();

var stdioTransport = new StdioClientTransport(stdioParams);

var mcpClient = McpClient.sync(stdioTransport).build();

mcpClient.initialize();

```

```
ListToolsResult toolsList = mcpClient.listTools();

CallToolResult weather = mcpClient.callTool(
    new CallToolRequest("getWeatherForecastByLocation",
        Map.of("latitude", "47.6062", "longitude", "-122.3321")));

CallToolResult alert = mcpClient.callTool(
    new CallToolRequest("getAlerts", Map.of("state", "NY")));

mcpClient.closeGracefully();
```

#### Use MCP Client Boot Starter
```

Create a new boot starter application using the `spring-ai-starter-mcp-client` dependency:

```
```xml
<dependency>
    <groupId>org.springframework.ai</groupId>
    <artifactId>spring-ai-starter-mcp-client</artifactId>
</dependency>
```

```

and set the `spring.ai.mcp.client.stdio.servers-configuration` property to point to your `claude\_desktop\_config.json`.

You can reuse the existing Anthropic Desktop configuration:

```
```properties
spring.ai.mcp.client.stdio.servers-configuration=file:PATH/TO/claude_desktop_config.json
```

```

when you start your client application, the auto-configuration will create, automatically MCP clients from the claude\desktop\\_config.json.

For more information, see the [MCP Client Boot Starters](<https://docs.spring.io/spring-ai/reference/api/mcp/mcp-server-boot-client-docs.html>) reference documentation.

### ### More Java MCP Server examples

The [starter-webflux-server](<https://github.com/spring-projects/spring-ai-examples/tree/main/model-context-protocol/weather/starter-webflux-server>) demonstrates how to create a MCP server using SSE transport.

It showcases how to define and register MCP Tools, Resources, and Prompts, using the Spring Boot's auto-configuration capabilities.

Let's get started with building our weather server! [You can find the complete code for what we'll be building here.](#)

### #### Prerequisite knowledge

This quickstart assumes you have familiarity with:

- \* Kotlin
- \* LLMs like Claude

#### #### System requirements

- \* Java 17 or higher installed.

#### #### Set up your environment

First, let's install `java` and `gradle` if you haven't already.

You can download `java` from [official Oracle JDK website]

(<https://www.oracle.com/java/technologies/downloads/>).

Verify your `java` installation:

```
```bash
java --version
```

```

Now, let's create and set up your project:

```
<CodeGroup>
  ```bash macos/Linux
  # Create a new directory for our project
  mkdir weather
  cd weather

  # Initialize a new kotlin project
  gradle init
  ```

  ```powershell windows
  # Create a new directory for our project
  md weather
  cd weather

  # Initialize a new kotlin project
  gradle init
  ```

</CodeGroup>
```

After running `gradle init`, you will be presented with options for creating your project. Select \*\*Application\*\* as the project type, \*\*Kotlin\*\* as the programming language, and \*\*Java 17\*\* as the Java version.

Alternatively, you can create a Kotlin application using the [IntelliJ IDEA project wizard] (<https://kotlinlang.org/docs/jvm-get-started.html>).

After creating the project, add the following dependencies:

```
<CodeGroup>
```

```

```kotlin build.gradle.kts
val mcpVersion = "0.4.0"
val slf4jVersion = "2.0.9"
val ktorVersion = "3.1.1"

dependencies {
    implementation("io.modelcontextprotocol:kotlin-sdk:$mcpVersion")
    implementation("org.slf4j:slf4j-nop:$slf4jVersion")
    implementation("io.ktor:ktor-client-content-negotiation:$ktorVersion")
    implementation("io.ktor:ktor-serialization-kotlinx-json:$ktorVersion")
}
```

```groovy build.gradle
def mcpVersion = '0.3.0'
def slf4jVersion = '2.0.9'
def ktorVersion = '3.1.1'

dependencies {
    implementation "io.modelcontextprotocol:kotlin-sdk:$mcpVersion"
    implementation "org.slf4j:slf4j-nop:$slf4jVersion"
    implementation "io.ktor:ktor-client-content-negotiation:$ktorVersion"
    implementation "io.ktor:ktor-serialization-kotlinx-json:$ktorVersion"
}
```

</CodeGroup>

```

Also, add the following plugins to your build script:

```

<CodeGroup>
```kotlin build.gradle.kts
plugins {
    kotlin("plugin.serialization") version "your_version_of_kotlin"
    id("com.github.johnrengelman.shadow") version "8.1.1"
}
```

```groovy build.gradle
plugins {
    id 'org.jetbrains.kotlin.plugin.serialization' version 'your_version_of_kotlin'
    id 'com.github.johnrengelman.shadow' version '8.1.1'
}
```

</CodeGroup>

```

Now let's dive into building your server.

### Building your server

#### Setting up the instance

Add a server initialization function:

```

```kotlin
// Main function to run the MCP server
fun `run mcp server`() {
    // Create the MCP Server instance with a basic implementation
    val server = Server(
        implementation(
            name = "weather", // Tool name is "weather"
            version = "1.0.0" // Version of the implementation
        ),
        serverOptions(
            capabilities = ServerCapabilities(tools = ServerCapabilities.Tools(listChanged =
true))
        )
    )

    // Create a transport using standard IO for server communication
    val transport = StdioServerTransport(
        System.`in`.asInput(),
        System.out.assink().buffered()
    )

    runBlocking {
        server.connect(transport)
        val done = Job()
        server.onClose {
            done.complete()
        }
        done.join()
    }
}
```
```

```

#### #### Weather API helper functions

Next, let's add functions and data classes for querying and converting responses from the National Weather Service API:

```

```kotlin
// Extension function to fetch forecast information for given latitude and longitude
suspend fun HttpClient.getForecast(latitude: Double, longitude: Double): List<String> {
    val points = this.get("/points/$latitude,$longitude").body<Points>()
    val forecast = this.get(points.properties.forecast).body<Forecast>()
    return forecast.properties.periods.map { period ->
        """
            ${period.name}:
            Temperature: ${period.temperature} ${period.temperatureUnit}
            Wind: ${period.windSpeed} ${period.windDirection}
            Forecast: ${period.detailedForecast}
        """.trimIndent()
    }
}

// Extension function to fetch weather alerts for a given state
```

```

```

suspend fun HttpClient.getAlerts(state: String): List<String> {
    val alerts = this.get("/alerts/active/area/$state").body<Alert>()
    return alerts.features.map { feature ->
        """
            Event: ${feature.properties.event}
            Area: ${feature.properties.areaDesc}
            Severity: ${feature.properties.severity}
            Description: ${feature.properties.description}
            Instruction: ${feature.properties.instruction}
        """.trimIndent()
    }
}

@Serializable
data class Points(
    val properties: Properties
) {
    @Serializable
    data class Properties(val forecast: String)
}

@Serializable
data class Forecast(
    val properties: Properties
) {
    @Serializable
    data class Properties(val periods: List<Period>)

    @Serializable
    data class Period(
        val number: Int, val name: String, val startTime: String, val endTime: String,
        val isDaytime: Boolean, val temperature: Int, val temperatureUnit: String,
        val temperatureTrend: String, val probabilityOfPrecipitation: JsonObject,
        val windSpeed: String, val windDirection: String,
        val shortForecast: String, val detailedForecast: String,
    )
}

@Serializable
data class Alert(
    val features: List<Feature>
) {
    @Serializable
    data class Feature(
        val properties: Properties
    )

    @Serializable
    data class Properties(
        val event: String, val areaDesc: String, val severity: String,
        val description: String, val instruction: String?,
    )
}

```

```
```
```

```
#### Implementing tool execution
```

The tool execution handler is responsible for actually executing the logic of each tool.  
Let's add it:

```
```kotlin
// Create an HTTP client with a default request configuration and JSON content negotiation
val httpClient = HttpClient {
    defaultRequest {
        url("https://api.weather.gov")
        headers {
            append("Accept", "application/geo+json")
            append("User-Agent", "weatherApiClient/1.0")
        }
        contentType(ContentType.Application.Json)
    }
    // Install content negotiation plugin for JSON serialization/deserialization
    install(ContentNegotiation) { json(Json { ignoreUnknownKeys = true }) }
}

// Register a tool to fetch weather alerts by state
server.addTool(
    name = "get_alerts",
    description = """
        Get weather alerts for a US state. Input is Two-letter US state code (e.g. CA, NY)
        """.trimIndent(),
    inputSchema = Tool.Input(
        properties = buildJsonObject {
            putJsonObject("state") {
                put("type", "string")
                put("description", "Two-letter US state code (e.g. CA, NY)")
            }
        },
        required = listOf("state")
    )
) { request ->
    val state = request.arguments["state"]?.jsonPrimitive?.content
    if (state == null) {
        return@addTool CallToolResult(
            content = listOf(TextContent("The 'state' parameter is required."))
        )
    }
}

val alerts = httpClient.getAlerts(state)

CallToolResult(content = alerts.map { TextContent(it) })
}

// Register a tool to fetch weather forecast by latitude and longitude
server.addTool(
    name = "get_forecast",
```

```

description = """
    Get weather forecast for a specific latitude/longitude
""".trimIndent(),
inputSchema = Tool.Input(
    properties = buildJsonObject {
        putJsonObject("latitude") { put("type", "number") }
        putJsonObject("longitude") { put("type", "number") }
    },
    required = listOf("latitude", "longitude")
)
) { request ->
    val latitude = request.arguments["latitude"]?.jsonPrimitive?.doubleOrNull
    val longitude = request.arguments["longitude"]?.jsonPrimitive?.doubleOrNull
    if (latitude == null || longitude == null) {
        return@addTool CallToolResult(
            content = listOf(TextContent("The 'latitude' and 'longitude' parameters are
required."))
        )
    }
}

val forecast = httpClient.getForecast(latitude, longitude)

CallToolResult(content = forecast.map { TextContent(it) })
}
```

```

#### #### Running the server

Finally, implement the main function to run the server:

```

```kotlin
fun main() = `run mcp server`()
```

```

Make sure to run `./gradlew build` to build your server. This is a very important step in getting your server to connect.

Let's now test your server from an existing MCP host, Claude for Desktop.

#### ### Testing your server with Claude for Desktop

<Note>

Claude for Desktop is not yet available on Linux. Linux users can proceed to the [Building a client](/quickstart/client) tutorial to build an MCP client that connects to the server we just built.

</Note>

First, make sure you have Claude for Desktop installed. [You can install the latest version here.](<https://claude.ai/download>) If you already have Claude for Desktop, \*\*make sure it's updated to the latest version.\*\*

We'll need to configure Claude for Desktop for whichever MCP servers you want to use.

To do this, open your Claude for Desktop App configuration at `~/Library/Application Support/Claude/clause\_desktop\_config.json` in a text editor.  
Make sure to create the file if it doesn't exist.

For example, if you have [vs Code](https://code.visualstudio.com/) installed:

```
<CodeGroup>
  ```bash macos/Linux
  code ~/Library/Application\ Support/Claude/clause_desktop_config.json
  ```

  ```powershell windows
  code $env:AppData\Claude\clause_desktop_config.json
  ```

</CodeGroup>
```

You'll then add your servers in the `mcpServers` key.

The MCP UI elements will only show up in Claude for Desktop if at least one server is properly configured.

In this case, we'll add our single weather server like so:

```
<CodeGroup>
  ```json macos/Linux
  {
    "mcpServers": {
      "weather": {
        "command": "java",
        "args": [
          "-jar",
          "/ABSOLUTE/PATH/TO/PARENT/FOLDER/weather/build/libs/weather-0.1.0-all.jar"
        ]
      }
    }
  ```

  ```json windows
  {
    "mcpServers": {
      "weather": {
        "command": "java",
        "args": [
          "-jar",
          "C:\\\\PATH\\\\TO\\\\PARENT\\\\FOLDER\\\\weather\\\\build\\\\libs\\\\weather-0.1.0-all.jar"
        ]
      }
    }
  ```

</CodeGroup>
```

This tells claude for Desktop:

1. There's an MCP server named "weather"
2. Launch it by running `java -jar /ABSOLUTE/PATH/TO/PARENT/FOLDER/weather/build/libs/weather-0.1.0-all.jar`

Save the file, and restart \*\*Claude for Desktop\*\*.

Let's get started with building our weather server! [You can find the complete code for what we'll be building here.](#)

#### #### Prerequisite knowledge

This quickstart assumes you have familiarity with:

- \* C#
- \* LLMs like Claude
- \* .NET 8 or higher

#### #### Logging in MCP Servers

When implementing MCP servers, be careful about how you handle logging:

\*\*For STDIO-based servers:\*\* Never write to standard output (stdout). This includes:

- \* `print()` statements in Python
- \* `console.log()` in JavaScript
- \* `fmt.Println()` in Go
- \* Similar stdout functions in other languages

Writing to stdout will corrupt the JSON-RPC messages and break your server.

\*\*For HTTP-based servers:\*\* Standard output logging is fine since it doesn't interfere with HTTP responses.

#### #### Best Practices

1. Use a logging library that writes to stderr or files

#### #### System requirements

- \* [.NET 8 SDK] (<https://dotnet.microsoft.com/download/dotnet/8.0>) or higher installed.

#### #### Set up your environment

First, let's install `dotnet` if you haven't already. You can download `dotnet` from [official Microsoft .NET website] (<https://dotnet.microsoft.com/download/>). Verify your `dotnet` installation:

```
```bash
dotnet --version
```

```
```
```

Now, let's create and set up your project:

```
<CodeGroup>
  ```bash macos/Linux
  # Create a new directory for our project
  mkdir weather
  cd weather
  # Initialize a new C# project
  dotnet new console
  ```

  ```powershell windows
  # Create a new directory for our project
  mkdir weather
  cd weather
  # Initialize a new C# project
  dotnet new console
  ```

</CodeGroup>
```

After running `dotnet new console`, you will be presented with a new C# project.

You can open the project in your favorite IDE, such as [Visual Studio] (<https://visualstudio.microsoft.com/>) or [Rider] (<https://www.jetbrains.com/rider/>).

Alternatively, you can create a C# application using the [Visual Studio project wizard] (<https://learn.microsoft.com/en-us/visualstudio/get-started/csharp/tutorial-console?view=vs-2022>).

After creating the project, add NuGet package for the Model Context Protocol SDK and hosting:

```
```bash
# Add the Model Context Protocol SDK NuGet package
dotnet add package ModelContextProtocol --prerelease
# Add the .NET Hosting NuGet package
dotnet add package Microsoft.Extensions.Hosting
````
```

Now let's dive into building your server.

```
## Building your server
```

Open the `Program.cs` file in your project and replace its contents with the following code:

```
```csharp
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using ModelContextProtocol;
using System.Net.Http.Headers;

var builder = Host.CreateApplicationBuilder(settings: null);

builder.Services.AddMcpServer()
```

```

    .WithStdioServerTransport()
    .WithToolsFromAssembly();

builder.Services.AddSingleton(_ =>
{
    var client = new HttpClient() { BaseAddress = new Uri("https://api.weather.gov") };
    client.DefaultRequestHeaders.UserAgent.Add(new ProductInfoHeaderValue("weather-tool",
    "1.0"));
    return client;
});

var app = builder.Build();

await app.RunAsync();
```

```

**<Note>**  
when creating the `ApplicationHostBuilder`, ensure you use `CreateEmptyApplicationBuilder` instead of `CreateDefaultBuilder`. This ensures that the server does not write any additional messages to the console. This is only necessary for servers using STDIO transport.  
**</Note>**

This code sets up a basic console application that uses the Model Context Protocol SDK to create an MCP server with standard I/O transport.

### ### Weather API helper functions

Create an extension class for `HttpClient` which helps simplify JSON request handling:

```

```csharp
using System.Text.Json;

internal static class HttpClientExt
{
    public static async Task<JsonDocument> ReadJsonDocumentAsync(this HttpClient client,
    string requestUri)
    {
        using var response = await client.GetAsync(requestUri);
        response.EnsureSuccessStatusCode();
        return await JsonDocument.ParseAsync(await response.Content.ReadAsStreamAsync());
    }
}
```

```

Next, define a class with the tool execution handlers for querying and converting responses from the National Weather Service API:

```

```csharp
using ModelContextProtocol.Server;
using System.ComponentModel;
using System.Globalization;
using System.Text.Json;
```

```

```

namespace QuickstartWeatherServer.Tools;

[McpServerToolType]
public static class WeatherTools
{
    [McpServerTool, Description("Get weather alerts for a US state.")]
    public static async Task<string> GetAlerts(
        HttpClient client,
        [Description("The US state to get alerts for.")] string state)
    {
        using var jsonDocument = await
client.ReadJsonDocumentAsync($""/alerts/active/area/{state}");
        var jsonElement = jsonDocument.RootElement;
        var alerts = jsonElement.GetProperty("features").EnumerateArray();

        if (!alerts.Any())
        {
            return "No active alerts for this state.";
        }

        return string.Join("\n--\n", alerts.Select(alert =>
{
    jsonElement properties = alert.GetProperty("properties");
    return $"""
        Event: {properties.GetProperty("event").GetString()}
        Area: {properties.GetProperty("areaDesc").GetString()}
        Severity: {properties.GetProperty("severity").GetString()}
        Description: {properties.GetProperty("description").GetString()}
        Instruction: {properties.GetProperty("instruction").GetString()}
        """;
}));
    }

    [McpServerTool, Description("Get weather forecast for a location.")]
    public static async Task<string> GetForecast(
        HttpClient client,
        [Description("Latitude of the location.")] double latitude,
        [Description("Longitude of the location.")] double longitude)
    {
        var pointUrl = string.Create(CultureInfo.InvariantCulture, $"{"/points/{latitude}, {longitude}"}");
        using var jsonDocument = await client.ReadJsonDocumentAsync(pointUrl);
        var forecastUrl =
jsonDocument.RootElement.GetProperty("properties").GetProperty("forecast").GetString()
            ?? throw new Exception($"No forecast URL provided by
{client.BaseAddress}points/{latitude},{longitude}");

        using var forecastDocument = await client.ReadJsonDocumentAsync(forecastUrl);
        var periods =
forecastDocument.RootElement.GetProperty("properties").GetProperty("periods").EnumerateArray
();
    }
}

```

```
        return string.Join("\n---\n", periods.Select(period => $"""
            {period.GetProperty("name").GetString()}
            Temperature: {period.GetProperty("temperature").GetInt32()}°F
            Wind: {period.GetProperty("windspeed").GetString()}
{period.GetProperty("windDirection").GetString()}
            Forecast: {period.GetProperty("detailedForecast").GetString()}
        """"));
    }
}
```

```

### Running the server

Finally, run the server using the following command:

```
```bash
dotnet run
```

```

This will start the server and listen for incoming requests on standard input/output.

### Testing your server with Claude for Desktop

<Note>

Claude for Desktop is not yet available on Linux. Linux users can proceed to the [Building a client](/quickstart/client) tutorial to build an MCP client that connects to the server we just built.

</Note>

First, make sure you have Claude for Desktop installed. [You can install the latest version here.](https://claude.ai/download) If you already have Claude for Desktop, \*\*make sure it's updated to the latest version.\*\*

We'll need to configure Claude for Desktop for whichever MCP servers you want to use. To do this, open your Claude for Desktop App configuration at `~/Library/Application Support/claude/claude\_desktop\_config.json` in a text editor. Make sure to create the file if it doesn't exist.

For example, if you have [VS Code](https://code.visualstudio.com/) installed:

```
<CodeGroup>
```bash macos/Linux
code ~/Library/Application\ Support/claude/claude_desktop_config.json
```

```powershell windows
code $env:AppData\Claude\claude_desktop_config.json
```
</CodeGroup>
```

You'll then add your servers in the `mcpServers` key. The MCP UI elements will only show up in Claude for Desktop if at least one server is properly configured.

In this case, we'll add our single weather server like so:

```
<CodeGroup>
```

```

```json macos/Linux
{
  "mcpServers": {
    "weather": {
      "command": "dotnet",
      "args": ["run", "--project", "/ABSOLUTE/PATH/TO/PROJECT", "--no-build"]
    }
  }
}
```
```
```json windows
{
  "mcpServers": {
    "weather": {
      "command": "dotnet",
      "args": [
        "run",
        "--project",
        "C:\\\\ABSOLUTE\\\\PATH\\\\TO\\\\PROJECT",
        "--no-build"
      ]
    }
  }
}
```
```
</CodeGroup>

```

This tells Claude for Desktop:

1. There's an MCP server named "weather"
  2. Launch it by running `dotnet run /ABSOLUTE/PATH/TO/PROJECT`
- Save the file, and restart \*\*Claude for Desktop\*\*.

## Test with commands

Let's make sure Claude for Desktop is picking up the two tools we've exposed in our `weather` server. You can do this by looking for the "Search and tools"  icon:



After clicking on the slider icon, you should see two tools listed:



## Disable all tools



## get\_alerts



## get\_forecast

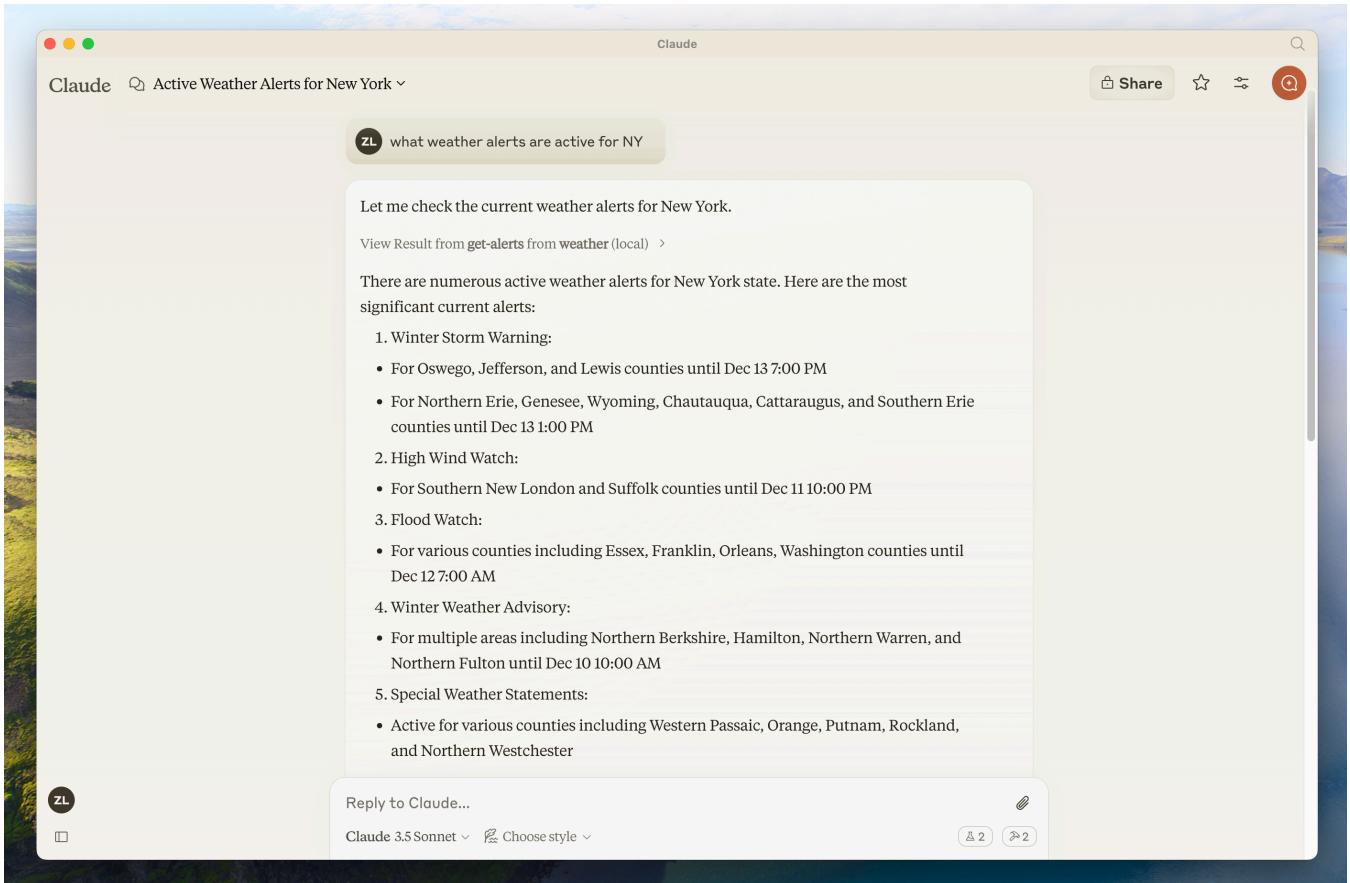


If your server isn't being picked up by Claude for Desktop, proceed to the [Troubleshooting](#) section for debugging tips.

If the tool settings icon has shown up, you can now test your server by running the following commands in Claude for Desktop:

- What's the weather in Sacramento?
- What are the active weather alerts in Texas?

The screenshot shows the Claude for Desktop interface. The window title is "Claude" and the subtitle is "Active Weather Alerts for New York". A note at the top says "Note the extended Winter Storm Warnings that stretch into December 13th." Below this is a message bubble: "Would you like me to focus on any particular region or type of alert in more detail?". A user input field contains "zl what's the weather in Sacramento". The response below says "Let me check the current weather in Sacramento, California." and includes a link "View Result from get-weather from weather (local) >". It then details the current weather in Sacramento: "Currently in Sacramento:" followed by a bulleted list: "Temperature: 56°F (falling to around 53°F in the afternoon)", "Conditions: Haze clearing to mostly sunny", "Wind: Light northeast wind around 2 mph", and "Notable conditions: Areas of frost and fog before 11am". It also provides a forecast for the next few days: "Next few days:" followed by a bulleted list: "Tonight: Low of 37°F, hazy conditions", "Wednesday: High of 56°F, partly sunny with morning frost", "Wednesday Night: 50% chance of rain showers, low around 44°F", and "Thursday: 50% chance of morning showers, partly sunny, high near 55°F". At the bottom, it asks "Would you like more details about the forecast for any particular day?" and includes a footer note "Claude can make mistakes. Please double-check responses." and a reply input field "Reply to Claude...".



Since this is the US National Weather service, the queries will only work for US locations.

## What's happening under the hood

When you ask a question:

1. The client sends your question to Claude
2. Claude analyzes the available tools and decides which one(s) to use
3. The client executes the chosen tool(s) through the MCP server
4. The results are sent back to Claude
5. Claude formulates a natural language response
6. The response is displayed to you!

## Troubleshooting

### Getting logs from Claude for Desktop

Claude.app logging related to MCP is written to log files in `~/Library/Logs/Claude`:

\* `mcp.log` will contain general logging about MCP connections and connection failures.

```
* Files named `mcp-server-SERVERNAME.log` will contain error (stderr) logging from the named server.
```

You can run the following command to list recent logs and follow along with any new ones:

```
```bash
# Check Claude's logs for errors
tail -n 20 -f ~/Library/Logs/Claude/mcp*.log
```

```

**\*\*Server not showing up in Claude\*\***

1. Check your `claude\_desktop\_config.json` file syntax
2. Make sure the path to your project is absolute and not relative
3. Restart Claude for Desktop completely

**\*\*Tool calls failing silently\*\***

If Claude attempts to use the tools but they fail:

1. Check Claude's logs for errors
2. Verify your server builds and runs without errors
3. Try restarting Claude for Desktop

**\*\*None of this is working. What do I do?\*\***

Please refer to our [debugging guide](/legacy/tools/debugging) for better debugging tools and more detailed guidance.

## Error: Failed to retrieve grid point data

This usually means either:

1. The coordinates are outside the US
2. The NWS API is having issues
3. You're being rate limited

Fix:

- \* Verify you're using US coordinates
- \* Add a small delay between requests
- \* Check the NWS API status page

**\*\*Error: No active alerts for \[STATE]\*\***

This isn't an error - it just means there are no current weather alerts for that state. Try a different state or check during severe weather.

For more advanced troubleshooting, check out our guide on [Debugging MCP](#)

## Next steps

Learn how to build your own MCP client that can connect to your server

Check out our gallery of official MCP servers and implementations

Learn how to effectively debug MCP servers and integrations

Learn how to use LLMs like Claude to speed up your MCP development

# Inspector

In-depth guide to using the MCP Inspector for testing and debugging Model Context Protocol servers

The [MCP Inspector](#) is an interactive developer tool for testing and debugging MCP servers. While the [Debugging Guide](#) covers the Inspector as part of the overall debugging toolkit, this document provides a detailed exploration of the Inspector's features and capabilities.

## Getting started

### Installation and basic usage

The Inspector runs directly through `npx` without requiring installation:

```
npx @modelcontextprotocol/inspector <command>
```

```
npx @modelcontextprotocol/inspector <command> <arg1> <arg2>
```

### Inspecting servers from NPM or PyPi

A common way to start server packages from [NPM](#) or [PyPi](#).

```
bash
npx -y @modelcontextprotocol/inspector npx <package-name> <args>
# For example
npx -y @modelcontextprotocol/inspector npx @modelcontextprotocol/server-filesystem
/Users/username/Desktop
```

```
bash
npx @modelcontextprotocol/inspector uvx <package-name> <args>
# For example
npx @modelcontextprotocol/inspector uvx mcp-server-git --repository ~code/mcp/servers.git
```

### Inspecting locally developed servers

To inspect servers locally developed or downloaded as a repository, the most common way is:

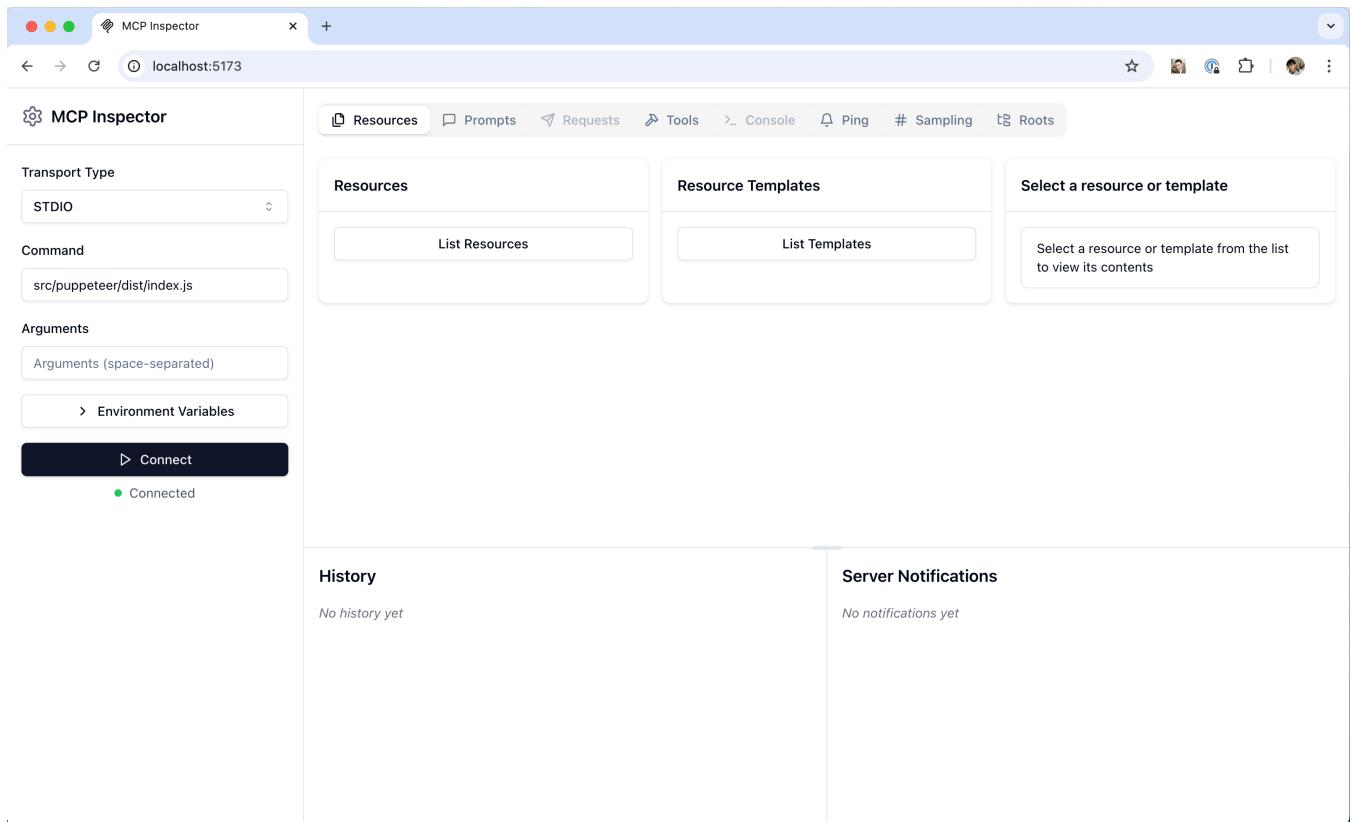
```
bash
npx @modelcontextprotocol/inspector node path/to/server/index.js args...
```

```
bash
```

```
npx @modelcontextprotocol/inspector \
  uv \
  --directory path/to/server \
  run \
  package-name \
  args...
```

Please carefully read any attached README for the most accurate instructions.

## Feature overview



The Inspector provides several features for interacting with your MCP server:

### Server connection pane

- Allows selecting the [transport](#) for connecting to the server
- For local servers, supports customizing the command-line arguments and environment

### Resources tab

- Lists all available resources
- Shows resource metadata (MIME types, descriptions)
- Allows resource content inspection

- Supports subscription testing

## Prompts tab

- Displays available prompt templates
- Shows prompt arguments and descriptions
- Enables prompt testing with custom arguments
- Previews generated messages

## Tools tab

- Lists available tools
- Shows tool schemas and descriptions
- Enables tool testing with custom inputs
- Displays tool execution results

## Notifications pane

- Presents all logs recorded from the server
- Shows notifications received from the server

## Best practices

### Development workflow

1. Start Development
  - Launch Inspector with your server
  - Verify basic connectivity
  - Check capability negotiation
2. Iterative testing
  - Make server changes
  - Rebuild the server
  - Reconnect the Inspector
  - Test affected features
  - Monitor messages
3. Test edge cases
  - Invalid inputs
  - Missing prompt arguments
  - Concurrent operations
  - Verify error handling and error responses

## Next steps

Check out the MCP Inspector source code

Learn about broader debugging strategies

# Build an MCP Client

Get started building your own client that can integrate with all MCP servers.

In this tutorial, you'll learn how to build an LLM-powered chatbot client that connects to MCP servers. It helps to have gone through the [Server quickstart](#) that guides you through the basics of building your first server.

[You can find the complete code for this tutorial here.](#)

### ### System Requirements

Before starting, ensure your system meets these requirements:

- \* Mac or Windows computer
- \* Latest Python version installed
- \* Latest version of `uv` installed

### ### Setting Up Your Environment

First, create a new Python project with `uv`:

```
```bash
# Create project directory
uv init mcp-client
cd mcp-client

# Create virtual environment
uv venv

# Activate virtual environment
# On Windows:
.venv\Scripts\activate
# On Unix or macOS:
source .venv/bin/activate

# Install required packages
uv add mcp anthropic python-dotenv

# Remove boilerplate files
# On Windows:
del main.py
# On Unix or macOS:
rm main.py

# Create our main file
touch client.py
```

### Setting Up Your API Key
```

```
You'll need an Anthropic API key from the [Anthropic Console]  
(https://console.anthropic.com/settings/keys).
```

Create a ` `.env` file to store it:

```
```bash  
# Create .env file  
touch .env  
```
```

Add your key to the ` `.env` file:

```
```bash  
ANTHROPIC_API_KEY=<your key here>  
```
```

Add ` `.env` to your ` `.gitignore` :

```
```bash  
echo ".env" >> .gitignore  
```
```

```
<Warning>  
  Make sure you keep your `ANTHROPIC_API_KEY` secure!  
</Warning>
```

### Creating the Client

#### Basic Client Structure

First, let's set up our imports and create the basic client class:

```
```python  
import asyncio  
from typing import Optional  
from contextlib import AsyncExitStack  
  
from mcp import ClientSession, StdioServerParameters  
from mcp.client.stdio import stdio_client  
  
from anthropic import Anthropic  
from dotenv import load_dotenv  
  
load_dotenv() # Load environment variables from .env  
  
class MCPClient:  
    def __init__(self):  
        # Initialize session and client objects  
        self.session: Optional[ClientSession] = None  
        self.exit_stack = AsyncExitStack()  
        self.anthropic = Anthropic()  
        # methods will go here  
```
```

#### #### Server Connection Management

Next, we'll implement the method to connect to an MCP server:

```
```python
async def connect_to_server(self, server_script_path: str):
    """Connect to an MCP server

Args:
    server_script_path: Path to the server script (.py or .js)
"""

    is_python = server_script_path.endswith('.py')
    is_js = server_script_path.endswith('.js')
    if not (is_python or is_js):
        raise ValueError("Server script must be a .py or .js file")

    command = "python" if is_python else "node"
    server_params = StdioServerParameters(
        command=command,
        args=[server_script_path],
        env=None
    )

    stdio_transport = await self.exit_stack.enter_async_context(stdio_client(server_params))
    self.stdio, self.write = stdio_transport
    self.session = await self.exit_stack.enter_async_context(clientSession(self.stdio,
self.write))

    await self.session.initialize()

    # List available tools
    response = await self.session.list_tools()
    tools = response.tools
    print("\nConnected to server with tools:", [tool.name for tool in tools])
```

#### Query Processing Logic
```

Now let's add the core functionality for processing queries and handling tool calls:

```
```python
async def process_query(self, query: str) -> str:
    """Process a query using Claude and available tools"""
    messages = [
        {
            "role": "user",
            "content": query
        }
    ]

    response = await self.session.list_tools()
    available_tools = [{
```

```
"name": tool.name,
"description": tool.description,
"input_schema": tool.inputSchema
} for tool in response.tools]

# Initial Claude API call
response = self.anthropic.messages.create(
    model="claude-3-5-sonnet-20241022",
    max_tokens=1000,
    messages=messages,
    tools=available_tools
)

# Process response and handle tool calls
final_text = []

assistant_message_content = []
for content in response.content:
    if content.type == 'text':
        final_text.append(content.text)
        assistant_message_content.append(content)
    elif content.type == 'tool_use':
        tool_name = content.name
        tool_args = content.input

        # Execute tool call
        result = await self.session.call_tool(tool_name, tool_args)
        final_text.append(f"[Calling tool {tool_name} with args {tool_args}]")

        assistant_message_content.append(content)
        messages.append({
            "role": "assistant",
            "content": assistant_message_content
        })
        messages.append({
            "role": "user",
            "content": [
                {
                    "type": "tool_result",
                    "tool_use_id": content.id,
                    "content": result.content
                }
            ]
        })

    # Get next response from Claude
    response = self.anthropic.messages.create(
        model="claude-3-5-sonnet-20241022",
        max_tokens=1000,
        messages=messages,
        tools=available_tools
    )
```

```
    final_text.append(response.content[0].text)

    return "\n".join(final_text)
```

```

#### #### Interactive Chat Interface

Now we'll add the chat loop and cleanup functionality:

```
```python
async def chat_loop(self):
    """Run an interactive chat loop"""
    print("\nMCP Client Started!")
    print("Type your queries or 'quit' to exit.")

    while True:
        try:
            query = input("\nQuery: ").strip()

            if query.lower() == 'quit':
                break

            response = await self.process_query(query)
            print("\n" + response)

        except Exception as e:
            print(f"\nError: {str(e)}")

async def cleanup(self):
    """Clean up resources"""
    await self.exit_stack.aclose()
```

```

#### #### Main Entry Point

Finally, we'll add the main execution logic:

```
```python
async def main():
    if len(sys.argv) < 2:
        print("Usage: python client.py <path_to_server_script>")
        sys.exit(1)

    client = MCPClient()
    try:
        await client.connect_to_server(sys.argv[1])
        await client.chat_loop()
    finally:
        await client.cleanup()

if __name__ == "__main__":
    import sys
    asyncio.run(main())
```

```

```

You can find the complete `client.py` file [here.]  
(<https://gist.github.com/zckly/f3f28ea731e096e53b39b47bf0a2d4b1>)

### ### Key Components Explained

#### #### 1. Client Initialization

- \* The `MCPClient` class initializes with session management and API clients
- \* Uses `AsyncExitStack` for proper resource management
- \* Configures the Anthropic client for Claude interactions

#### #### 2. Server Connection

- \* Supports both Python and Node.js servers
- \* Validates server script type
- \* Sets up proper communication channels
- \* Initializes the session and lists available tools

#### #### 3. Query Processing

- \* Maintains conversation context
- \* Handles Claude's responses and tool calls
- \* Manages the message flow between Claude and tools
- \* Combines results into a coherent response

#### #### 4. Interactive Interface

- \* Provides a simple command-line interface
- \* Handles user input and displays responses
- \* Includes basic error handling
- \* Allows graceful exit

#### #### 5. Resource Management

- \* Proper cleanup of resources
- \* Error handling for connection issues
- \* Graceful shutdown procedures

### ## Common Customization Points

#### 1. \*\*Tool Handling\*\*

- \* Modify `process\_query()` to handle specific tool types
- \* Add custom error handling for tool calls
- \* Implement tool-specific response formatting

#### 2. \*\*Response Processing\*\*

- \* Customize how tool results are formatted
- \* Add response filtering or transformation
- \* Implement custom logging

3. \*\*User Interface\*\*
  - \* Add a GUI or web interface
  - \* Implement rich console output
  - \* Add command history or auto-completion

### ### Running the Client

To run your client with any MCP server:

```
```bash
uv run client.py path/to/server.py # python server
uv run client.py path/to/build/index.js # node server
````
```

<Note>

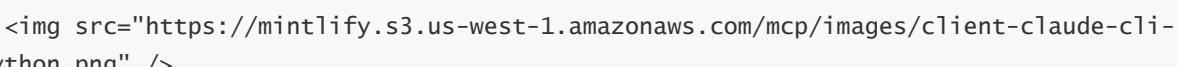
If you're continuing the weather tutorial from the server quickstart, your command might look something like this: `python client.py .../quickstart-resources/weather-server-python/weather.py`

</Note>

The client will:

1. Connect to the specified server
2. List available tools
3. Start an interactive chat session where you can:
  - \* Enter queries
  - \* See tool executions
  - \* Get responses from Claude

Here's an example of what it should look like if connected to the weather server from the server quickstart:

```
<Frame>

</Frame>
```

### ### How It Works

When you submit a query:

1. The client gets the list of available tools from the server
2. Your query is sent to Claude along with tool descriptions
3. Claude decides which tools (if any) to use
4. The client executes any requested tool calls through the server
5. Results are sent back to Claude
6. Claude provides a natural language response
7. The response is displayed to you

### ### Best practices

1. \*\*Error Handling\*\*

- \* Always wrap tool calls in try-catch blocks
- \* Provide meaningful error messages
- \* Gracefully handle connection issues

## 2. \*\*Resource Management\*\*

- \* Use `AsyncExitStack` for proper cleanup
- \* Close connections when done
- \* Handle server disconnections

## 3. \*\*Security\*\*

- \* Store API keys securely in ` `.env`
- \* Validate server responses
- \* Be cautious with tool permissions

### ### Troubleshooting

#### #### Server Path Issues

- \* Double-check the path to your server script is correct
- \* Use the absolute path if the relative path isn't working
- \* For Windows users, make sure to use forward slashes (/) or escaped backslashes (\\) in the path
- \* Verify the server file has the correct extension (.py for Python or .js for Node.js)

Example of correct path usage:

```
```bash
# Relative path
uv run client.py ./server/weather.py

# Absolute path
uv run client.py /Users/username/projects/mcp-server/weather.py

# Windows path (either format works)
uv run client.py C:/projects/mcp-server/weather.py
uv run client.py C:\\projects\\\\mcp-server\\\\weather.py
```

```

#### #### Response Timing

- \* The first response might take up to 30 seconds to return
- \* This is normal and happens while:
  - \* The server initializes
  - \* Claude processes the query
  - \* Tools are being executed
- \* Subsequent responses are typically faster
- \* Don't interrupt the process during this initial waiting period

#### #### Common Error Messages

If you see:

- \* `FileNotFoundException`: Check your server path
- \* `Connection refused`: Ensure the server is running and the path is correct
- \* `Tool execution failed`: Verify the tool's required environment variables are set
- \* `Timeout error`: Consider increasing the timeout in your client configuration

[You can find the complete code for this tutorial here.](#)

### ### System Requirements

Before starting, ensure your system meets these requirements:

- \* Mac or Windows computer
- \* Node.js 17 or higher installed
- \* Latest version of `npm` installed
- \* Anthropic API key (claude)

### ### Setting Up Your Environment

First, let's create and set up our project:

```
<CodeGroup>
  ```bash macos/Linux
  # Create project directory
  mkdir mcp-client-typescript
  cd mcp-client-typescript

  # Initialize npm project
  npm init -y

  # Install dependencies
  npm install @anthropic-ai/sdk @modelcontextprotocol/sdk dotenv

  # Install dev dependencies
  npm install -D @types/node typescript

  # Create source file
  touch index.ts
  ```

  ```powershell windows
  # Create project directory
  md mcp-client-typescript
  cd mcp-client-typescript

  # Initialize npm project
  npm init -y

  # Install dependencies
  npm install @anthropic-ai/sdk @modelcontextprotocol/sdk dotenv
```

```
# Install dev dependencies
npm install -D @types/node typescript

# Create source file
new-item index.ts
```
</CodeGroup>
```

Update your `package.json` to set `type: "module"` and a build script:

```
```json package.json
{
  "type": "module",
  "scripts": {
    "build": "tsc && chmod 755 build/index.js"
  }
}
```

```

Create a `tsconfig.json` in the root of your project:

```
```json tsconfig.json
{
  "compilerOptions": {
    "target": "ES2022",
    "module": "Node16",
    "moduleResolution": "Node16",
    "outDir": "./build",
    "rootDir": "./",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true
  },
  "include": ["index.ts"],
  "exclude": ["node_modules"]
}
```

```

### ### Setting Up Your API Key

You'll need an Anthropic API key from the [Anthropic Console](<https://console.anthropic.com/settings/keys>).

Create a `\*.env` file to store it:

```
```bash
echo "ANTHROPIC_API_KEY=<your key here>" > .env
```

```

Add `\*.env` to your `\*.gitignore`:

```

```bash
echo ".env" >> .gitignore
```

<Warning>
  Make sure you keep your `ANTHROPIC_API_KEY` secure!
</Warning>

### Creating the Client

#### Basic Client Structure

```

First, let's set up our imports and create the basic client class in `index.ts`:

```

```typescript
import { Anthropic } from "@anthropic-ai/sdk";
import {
  MessageParam,
  Tool,
} from "@anthropic-ai/sdk/resources/messages/messages.mjs";
import { Client } from "@modelcontextprotocol/sdk/client/index.js";
import { StdioClientTransport } from "@modelcontextprotocol/sdk/client/stdio.js";
import readline from "readline/promises";
import dotenv from "dotenv";

dotenv.config();

const ANTHROPIC_API_KEY = process.env.ANTHROPIC_API_KEY;
if (!ANTHROPIC_API_KEY) {
  throw new Error("ANTHROPIC_API_KEY is not set");
}

class MCPClient {
  private mcp: Client;
  private anthropic: Anthropic;
  private transport: StdioClientTransport | null = null;
  private tools: Tool[] = [];

  constructor() {
    this.anthropic = new Anthropic({
      apiKey: ANTHROPIC_API_KEY,
    });
    this.mcp = new Client({ name: "mcp-client-cli", version: "1.0.0" });
  }
  // methods will go here
}

#### Server Connection Management

```

Next, we'll implement the method to connect to an MCP server:

```

```typescript

```

```

async connectToServer(serverScriptPath: string) {
  try {
    const isJs = serverScriptPath.endsWith(".js");
    const isPy = serverScriptPath.endsWith(".py");
    if (!isJs && !isPy) {
      throw new Error("Server script must be a .js or .py file");
    }
    const command = isPy
      ? process.platform === "win32"
        ? "python"
        : "python3"
      : process.execPath;

    this.transport = new StdioClientTransport({
      command,
      args: [serverScriptPath],
    });
    await this.mcp.connect(this.transport);

    const toolsResult = await this.mcp.listTools();
    this.tools = toolsResult.tools.map((tool) => {
      return {
        name: tool.name,
        description: tool.description,
        input_schema: tool.inputsSchema,
      };
    });
    console.log(
      "Connected to server with tools:",
      this.tools.map(({ name }) => name)
    );
  } catch (e) {
    console.log("Failed to connect to MCP server: ", e);
    throw e;
  }
}
```

```

#### #### Query Processing Logic

Now let's add the core functionality for processing queries and handling tool calls:

```

```typescript
async processQuery(query: string) {
  const messages: MessageParam[] = [
    {
      role: "user",
      content: query,
    },
  ];

  const response = await this.anthropic.messages.create({
    model: "claude-3-5-sonnet-20241022",
  });
}
```

```

```

    max_tokens: 1000,
    messages,
    tools: this.tools,
  });

const finalText = [];

for (const content of response.content) {
  if (content.type === "text") {
    finalText.push(content.text);
  } else if (content.type === "tool_use") {
    const toolName = content.name;
    const toolArgs = content.input as { [x: string]: unknown } | undefined;

    const result = await this.mcp.callTool({
      name: toolName,
      arguments: toolArgs,
    });
    finalText.push(
      `[Calling tool ${toolName} with args ${JSON.stringify(toolArgs)}]`
    );
  }

  messages.push({
    role: "user",
    content: result.content as string,
  });
}

const response = await this.anthropic.messages.create({
  model: "claude-3-5-sonnet-20241022",
  max_tokens: 1000,
  messages,
});

finalText.push(
  response.content[0].type === "text" ? response.content[0].text : ""
);
}

}

return finalText.join("\n");
}
```

```

#### #### Interactive Chat Interface

Now we'll add the chat loop and cleanup functionality:

```

```typescript
async chatLoop() {
  const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout,
  });

```

```

try {
    console.log("\nMCP Client Started!");
    console.log("Type your queries or 'quit' to exit.");

    while (true) {
        const message = await rl.question("\nQuery: ");
        if (message.toLowerCase() === "quit") {
            break;
        }
        const response = await this.processQuery(message);
        console.log("\n" + response);
    }
} finally {
    rl.close();
}
}

async cleanup() {
    await this.mcp.close();
}
```

```

#### #### Main Entry Point

Finally, we'll add the main execution logic:

```

```typescript
async function main() {
    if (process.argv.length < 3) {
        console.log("Usage: node index.ts <path_to_server_script>");
        return;
    }
    const mcpClient = new MCPClient();
    try {
        await mcpClient.connectToServer(process.argv[2]);
        await mcpClient.chatLoop();
    } finally {
        await mcpClient.cleanup();
        process.exit(0);
    }
}

main();
```

```

#### ### Running the Client

To run your client with any MCP server:

```

```bash
# Build TypeScript
npm run build

```

```
# Run the client
node build/index.js path/to/server.py # python server
node build/index.js path/to/build/index.js # node server
```
```

<Note>

If you're continuing the weather tutorial from the server quickstart, your command might look something like this: `node build/index.js .../quickstart-resources/weather-server-typescript/build/index.js`

</Note>

\*\*The client will:\*\*

1. Connect to the specified server
2. List available tools
3. Start an interactive chat session where you can:
  - \* Enter queries
  - \* See tool executions
  - \* Get responses from Claude

### How It Works

when you submit a query:

1. The client gets the list of available tools from the server
2. Your query is sent to Claude along with tool descriptions
3. Claude decides which tools (if any) to use
4. The client executes any requested tool calls through the server
5. Results are sent back to Claude
6. Claude provides a natural language response
7. The response is displayed to you

### Best practices

## 1. \*\*Error Handling\*\*

- \* Use TypeScript's type system for better error detection
- \* wrap tool calls in try-catch blocks
- \* Provide meaningful error messages
- \* Gracefully handle connection issues

## 2. \*\*Security\*\*

- \* Store API keys securely in ` `.env`
- \* validate server responses
- \* Be cautious with tool permissions

### Troubleshooting

#### Server Path Issues

- \* Double-check the path to your server script is correct
- \* Use the absolute path if the relative path isn't working

- \* For Windows users, make sure to use forward slashes (/) or escaped backslashes (\\\) in the path
- \* Verify the server file has the correct extension (.js for Node.js or .py for Python)

Example of correct path usage:

```
```bash
# Relative path
node build/index.js ./server/build/index.js

# Absolute path
node build/index.js /users/username/projects/mcp-server/build/index.js

# Windows path (either format works)
node build/index.js C:/projects/mcp-server/build/index.js
node build/index.js C:\\projects\\mcp-server\\build\\index.js
```

```

#### #### Response Timing

- \* The first response might take up to 30 seconds to return
- \* This is normal and happens while:
  - \* The server initializes
  - \* Claude processes the query
  - \* Tools are being executed
- \* Subsequent responses are typically faster
- \* Don't interrupt the process during this initial waiting period

#### #### Common Error Messages

If you see:

- \* `Error: Cannot find module`: Check your build folder and ensure TypeScript compilation succeeded
- \* `Connection refused`: Ensure the server is running and the path is correct
- \* `Tool execution failed`: Verify the tool's required environment variables are set
- \* `ANTHROPIC\_API\_KEY is not set`: Check your .env file and environment variables
- \* `TypeError`: Ensure you're using the correct types for tool arguments

This is a quickstart demo based on Spring AI MCP auto-configuration and boot starters.

To learn how to create sync and async MCP Clients manually, consult the [Java SDK Client](#) documentation

This example demonstrates how to build an interactive chatbot that combines Spring AI's Model Context Protocol (MCP) with the [Brave Search MCP Server] (<https://github.com/modelcontextprotocol/servers-archived/tree/main/src/brave-search>). The application creates a conversational interface powered by Anthropic's Claude AI model that can perform internet searches through Brave Search, enabling natural language interactions with real-time web data.  
[You can find the complete code for this tutorial here.](<https://github.com/spring-projects/spring-ai-examples/tree/main/model-context-protocol/web-search/brave-chatbot>)

### ### System Requirements

Before starting, ensure your system meets these requirements:

- \* Java 17 or higher
- \* Maven 3.6+
- \* npx package manager
- \* Anthropic API key (Claude)
- \* Brave Search API key

### ### Setting Up Your Environment

#### 1. Install npx (Node Package execute):

First, make sure to install [npm] (<https://docs.npmjs.com/downloading-and-installing-nodejs-and-npm>)  
and then run:

```
```bash
npm install -g npx
````
```

#### 2. Clone the repository:

```
```bash
git clone https://github.com/spring-projects/spring-ai-examples.git
cd model-context-protocol/brave-chatbot
````
```

#### 3. Set up your API keys:

```
```bash
export ANTHROPIC_API_KEY='your-anthropic-api-key-here'
export BRAVE_API_KEY='your-brave-api-key-here'
````
```

#### 4. Build the application:

```
```bash
./mvnw clean install
````
```

#### 5. Run the application using Maven:

```
```bash
./mvnw spring-boot:run
````
```

```
```
<warning>
  Make sure you keep your `ANTHROPIC_API_KEY` and `BRAVE_API_KEY` keys secure!
</warning>
```

### ### How it works

The application integrates Spring AI with the Brave Search MCP server through several components:

#### #### MCP Client Configuration

##### 1. Required dependencies in pom.xml:

```
```xml
<dependency>
  <groupId>org.springframework.ai</groupId>
  <artifactId>spring-ai-starter-mcp-client</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.ai</groupId>
  <artifactId>spring-ai-starter-model-anthropic</artifactId>
</dependency>
````
```

##### 2. Application properties (application.yml):

```
```yaml
spring:
  ai:
    mcp:
      client:
        enabled: true
        name: brave-search-client
        version: 1.0.0
        type: SYNC
        request-timeout: 20s
        stdio:
          root-change-notification: true
          servers-configuration: classpath:/mcp-servers-config.json
        toolcallback:
          enabled: true
      anthropic:
        api-key: ${ANTHROPIC_API_KEY}
````
```

This activates the `spring-ai-starter-mcp-client` to create one or more `McpClient`s based on the provided server configuration.

The `spring.ai.mcp.client.toolcallback.enabled=true` property enables the tool callback mechanism, that automatically registers all MCP tool as spring ai tools.  
It is disabled by default.

```
3. MCP Server Configuration (`mcp-servers-config.json`):
```

```
```json
{
  "mcpServers": {
    "brave-search": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-brave-search"],
      "env": {
        "BRAVE_API_KEY": "<PUT YOUR BRAVE API KEY>"
      }
    }
  }
}
```

```

#### #### Chat Implementation

The chatbot is implemented using Spring AI's ChatClient with MCP tool integration:

```
```java
var chatClient = chatClientBuilder
  .defaultSystem("You are useful assistant, expert in AI and Java.")
  .defaultToolCallbacks((Object[]) mcpToolAdapter.toolCallbacks())
  .defaultAdvisors(new MessageChatMemoryAdvisor(new InMemoryChatMemory()))
  .build();
```

<warning>
  Breaking change: From SpringAI 1.0.0-M8 onwards, use `.` instead of `.` to register MCP tools.
</warning>
```

Key features:

- \* Uses Claude AI model for natural language understanding
- \* Integrates Brave Search through MCP for real-time web search capabilities
- \* Maintains conversation memory using InMemoryChatMemory
- \* Runs as an interactive command-line application

#### #### Build and run

```
```bash
./mvnw clean install
java -jar ./target/ai-mcp-brave-chatbot-0.0.1-SNAPSHOT.jar
```

```

or

```
```bash
./mvnw spring-boot:run
```

```

The application will start an interactive chat session where you can ask questions. The chatbot will use Brave Search when it needs to find information from the internet to answer your queries.

The chatbot can:

- \* Answer questions using its built-in knowledge
- \* Perform web searches when needed using Brave Search
- \* Remember context from previous messages in the conversation
- \* Combine information from multiple sources to provide comprehensive answers

#### #### Advanced Configuration

The MCP client supports additional configuration options:

- \* Client customization through `McpSyncClientCustomizer` or `McpAsyncClientCustomizer`
- \* Multiple clients with multiple transport types: `STDIO` and `SSE` (Server-Sent Events)
- \* Integration with Spring AI's tool execution framework
- \* Automatic client initialization and lifecycle management

For WebFlux-based applications, you can use the WebFlux starter instead:

```
```xml
<dependency>
    <groupId>org.springframework.ai</groupId>
    <artifactId>spring-ai-mcp-client-webflux-spring-boot-starter</artifactId>
</dependency>
````
```

This provides similar functionality but uses a WebFlux-based SSE transport implementation, recommended for production deployments.

[You can find the complete code for this tutorial here.](#)

#### ### System Requirements

Before starting, ensure your system meets these requirements:

- \* Java 17 or higher
- \* Anthropic API key (claude)

#### ### Setting up your environment

First, let's install `java` and `gradle` if you haven't already. You can download `java` from [official oracle JDK website] (<https://www.oracle.com/java/technologies/downloads/>). Verify your `java` installation:

```
```bash
java --version
````
```

```
```
```

Now, let's create and set up your project:

```
<CodeGroup>
  ```bash macos/Linux
  # Create a new directory for our project
  mkdir kotlin-mcp-client
  cd kotlin-mcp-client

  # Initialize a new kotlin project
  gradle init
  ```

  ```powershell windows
  # Create a new directory for our project
  md kotlin-mcp-client
  cd kotlin-mcp-client
  # Initialize a new kotlin project
  gradle init
  ```

</CodeGroup>
```

After running `gradle init`, you will be presented with options for creating your project. Select **Application** as the project type, **Kotlin** as the programming language, and **Java 17** as the Java version.

Alternatively, you can create a Kotlin application using the [IntelliJ IDEA project wizard] (<https://kotlinlang.org/docs/jvm-get-started.html>).

After creating the project, add the following dependencies:

```
<CodeGroup>
  ```kotlin build.gradle.kts
  val mcpVersion = "0.4.0"
  val slf4jVersion = "2.0.9"
  val anthropicVersion = "0.8.0"

  dependencies {
    implementation("io.modelcontextprotocol:kotlin-sdk:$mcpVersion")
    implementation("org.slf4j:slf4j-nop:$slf4jVersion")
    implementation("com.anthropic:anthropic-java:$anthropicVersion")
  }
  ```

  ```groovy build.gradle
  def mcpVersion = '0.3.0'
  def slf4jVersion = '2.0.9'
  def anthropicVersion = '0.8.0'
  dependencies {
    implementation "io.modelcontextprotocol:kotlin-sdk:$mcpVersion"
    implementation "org.slf4j:slf4j-nop:$slf4jVersion"
    implementation "com.anthropic:anthropic-java:$anthropicVersion"
  }
  ```

</CodeGroup>
```

```
}
```

```
```
```

```
</CodeGroup>
```

Also, add the following plugins to your build script:

```
<CodeGroup>
```

```
```kotlin build.gradle.kts
```

```
plugins {
```

```
    id("com.github.johnrengelman.shadow") version "8.1.1"
```

```
}
```

```
```
```

```
```groovy build.gradle
```

```
plugins {
```

```
    id 'com.github.johnrengelman.shadow' version '8.1.1'
```

```
}
```

```
```
```

```
</CodeGroup>
```

### ### Setting up your API key

You'll need an Anthropic API key from the [Anthropic Console](<https://console.anthropic.com/settings/keys>).

Set up your API key:

```
```bash
```

```
export ANTHROPIC_API_KEY='your-anthropic-api-key-here'
```

```
```
```

```
<warning>
```

```
Make sure you keep your `ANTHROPIC_API_KEY` secure!
```

```
</warning>
```

### ### Creating the Client

#### #### Basic Client Structure

First, let's create the basic client class:

```
```kotlin
```

```
class MCPClient : AutoCloseable {
```

```
    private val anthropic = AnthropicOkHttpClient.fromEnv()
```

```
    private val mcp: Client = Client(clientInfo = Implementation(name = "mcp-client-cli",
```

```
version = "1.0.0"))
```

```
    private lateinit var tools: List<ToolUnion>
```

```
    // methods will go here
```

```
    override fun close() {
```

```
        runBlocking {
```

```
            mcp.close()
```

```
        anthropic.close()
    }
}
```

```

#### #### Server connection management

Next, we'll implement the method to connect to an MCP server:

```
```kotlin
suspend fun connectToServer(serverScriptPath: String) {
    try {
        val command = buildList {
            when (serverScriptPath.substringAfterLast(".")) {
                "js" -> add("node")
                "py" -> add(if (System.getProperty("os.name").toLowerCase().contains("win"))
"python" else "python3")
                "jar" -> addAll(listOf("java", "-jar"))
                else -> throw IllegalArgumentException("Server script must be a .js, .py or
.jar file")
            }
            add(serverScriptPath)
        }

        val process = ProcessBuilder(command).start()
        val transport = StdioClientTransport(
            input = process.getInputStream.asSource().buffered(),
            output = process.getOutputStream.asSink().buffered()
        )

        mcp.connect(transport)

        val toolsResult = mcp.listTools()
        tools = toolsResult?.tools?.map { tool ->
            ToolUnion.ofTool(
                Tool.builder()
                    .name(tool.name)
                    .description(tool.description ?: "")
                    .inputSchema(
                        Tool.InputSchema.builder()
                            .type(JsonValue.from(tool.inputSchema.type))
                            .properties(tool.inputSchema.properties.toJsonValue())
                            .putAdditionalProperty("required",
JsonValue.from(tool.inputSchema.required))
                            .build()
                    )
                    .build()
            )
        }
    } ?: emptyList()
    println("Connected to server with tools: ${tools.joinToString(", ")} {
it.tool().get().name() }}")
} catch (e: Exception) {
    println("Failed to connect to MCP server: $e")
}
```

```
        throw e
    }
}
```

```

Also create a helper function to convert from `JsonObject` to `JsonValue` for Anthropic:

```
```kotlin
private fun JsonObject.toJsonValue(): JsonValue {
    val mapper = ObjectMapper()
    val node = mapper.readTree(this.toString())
    return JsonValue.fromJsonNode(node)
}
```

```

#### Query processing logic

Now let's add the core functionality for processing queries and handling tool calls:

```
```kotlin
private val messageParamsBuilder: MessageCreateParams.Builder =
MessageCreateParams.builder()
    .model(Model.CLAUDE_3_5 SONNET_20241022)
    .maxTokens(1024)

suspend fun processQuery(query: String): String {
    val messages = mutableListOf(
        MessageParam.builder()
            .role(MessageParam.Role.USER)
            .content(query)
            .build()
    )

    val response = anthropic.messages().create(
        messageParamsBuilder
            .messages(messages)
            .tools(tools)
            .build()
    )

    val finalText = mutableListOf<String>()
    response.content().forEach { content ->
        when {
            content.isText() -> finalText.add(content.text().getOrNull()?.text() ?: "")

            content.isTooluse() -> {
                val toolName = content.tooluse().get().name()
                val toolArgs =
                    content.tooluse().get()._input().convert(object :
TypeReference<Map<String, JsonValue>>() {})

                val result = mcp.callTool(
                    name = toolName,

```

```

        arguments = toolArgs ?: emptyMap()
    )
    finalText.add("[Calling tool $toolName with args $toolArgs]")

    messages.add(
        MessageParam.builder()
            .role(MessageParam.Role.USER)
            .content(
                """
                    "type": "tool_result",
                    "tool_name": $toolName,
                    "result": ${result?.content?.joinToString("\n")} { (it as
TextContent).text ?: "" }
                """
                """.trimIndent()
            )
            .build()
    )

    val aiResponse = anthropic.messages().create(
        messageParamsBuilder
            .messages(messages)
            .build()
    )

    finalText.add(aiResponse.content().first().text().getOrNull()?.text() ?: "")
}
}

return finalText.toString("\n", prefix = "", postfix = "")
}
```

```

#### #### Interactive chat

we'll add the chat loop:

```

```kotlin
suspend fun chatLoop() {
    println("\nMCP Client Started!")
    println("Type your queries or 'quit' to exit.")

    while (true) {
        print("\nQuery: ")
        val message = readLine() ?: break
        if (message.lowercase() == "quit") break
        val response = processQuery(message)
        println("\n$response")
    }
}
```

```

#### #### Main entry point

Finally, we'll add the main execution function:

```
```kotlin
fun main(args: Array<String>) = runBlocking {
    if (args.isEmpty()) throw IllegalArgumentException("Usage: java -jar
<your_path>/build/libs/kotlin-mcp-client-0.1.0-all.jar <path_to_server_script>")
    val serverPath = args.first()
    val client = MCPClient()
    client.use {
        client.connectToServer(serverPath)
        client.chatLoop()
    }
}
```

```

### Running the client

To run your client with any MCP server:

```
```bash
./gradlew build

# Run the client
java -jar build/libs/<your-jar-name>.jar path/to/server.jar # jvm server
java -jar build/libs/<your-jar-name>.jar path/to/server.py # python server
java -jar build/libs/<your-jar-name>.jar path/to/build/index.js # node server
```

```

<Note>

If you're continuing the weather tutorial from the server quickstart, your command might look something like this: `java -jar build/libs/kotlin-mcp-client-0.1.0-all.jar .../samples/weather-stdio-server/build/libs/weather-stdio-server-0.1.0-all.jar`

</Note>

\*\*The client will:\*\*

1. Connect to the specified server
2. List available tools
3. Start an interactive chat session where you can:
  - \* Enter queries
  - \* See tool executions
  - \* Get responses from Claude

### How it works

Here's a high-level workflow schema:

```
```mermaid
---
config:
    theme: neutral
---
```

```

sequenceDiagram
    actor User
    participant Client
    participant Claude
    participant MCP_Server as MCP Server
    participant Tools

    User->>Client: Send query
    Client<<->>MCP_Server: Get available tools
    Client->>Claude: Send query with tool descriptions
    Claude-->>Client: Decide tool execution
    Client->>MCP_Server: Request tool execution
    MCP_Server-->>Tools: Execute chosen tools
    Tools-->>MCP_Server: Return results
    MCP_Server-->>Client: Send results
    Client->>Claude: Send tool results
    Claude-->>Client: Provide final response
    Client-->>User: Display response
```

```

when you submit a query:

1. The client gets the list of available tools from the server
2. Your query is sent to Claude along with tool descriptions
3. Claude decides which tools (if any) to use
4. The client executes any requested tool calls through the server
5. Results are sent back to Claude
6. Claude provides a natural language response
7. The response is displayed to you

### ### Best practices

#### 1. \*\*Error Handling\*\*

- \* Leverage Kotlin's type system to model errors explicitly
- \* wrap external tool and API calls in `try-catch` blocks when exceptions are possible
- \* Provide clear and meaningful error messages
- \* Handle network timeouts and connection issues gracefully

#### 2. \*\*Security\*\*

- \* Store API keys and secrets securely in `local.properties`, environment variables, or secret managers
- \* validate all external responses to avoid unexpected or unsafe data usage
- \* Be cautious with permissions and trust boundaries when using tools

### ### Troubleshooting

#### #### Server Path Issues

- \* Double-check the path to your server script is correct
- \* Use the absolute path if the relative path isn't working
- \* For Windows users, make sure to use forward slashes (/) or escaped backslashes (\\) in the path

- \* Make sure that the required runtime is installed (java for Java, npm for Node.js, or uv for Python)
- \* Verify the server file has the correct extension (.jar for Java, .js for Node.js or .py for Python)

Example of correct path usage:

```
```bash
# Relative path
java -jar build/libs/client.jar ./server/build/libs/server.jar

# Absolute path
java -jar build/libs/client.jar /Users/username/projects/mcp-server/build/libs/server.jar

# Windows path (either format works)
java -jar build/libs/client.jar C:/projects/mcp-server/build/libs/server.jar
java -jar build/libs/client.jar C:\\projects\\mcp-server\\build\\libs\\server.jar
```

```

#### #### Response Timing

- \* The first response might take up to 30 seconds to return
- \* This is normal and happens while:
  - \* The server initializes
  - \* Claude processes the query
  - \* Tools are being executed
- \* Subsequent responses are typically faster
- \* Don't interrupt the process during this initial waiting period

#### #### Common Error Messages

If you see:

- \* `Connection refused`: Ensure the server is running and the path is correct
- \* `Tool execution failed`: Verify the tool's required environment variables are set
- \* `ANTHROPIC\_API\_KEY is not set`: Check your environment variables

[You can find the complete code for this tutorial here.](#)

#### ### System Requirements

Before starting, ensure your system meets these requirements:

- \* .NET 8.0 or higher
- \* Anthropic API key (claude)
- \* Windows, Linux, or macOS

#### ### Setting up your environment

First, create a new .NET project:

```
```bash
dotnet new console -n QuickstartClient
cd QuickstartClient
````
```

Then, add the required dependencies to your project:

```
```bash
dotnet add package ModelContextProtocol --prerelease
dotnet add package Anthropic.SDK
dotnet add package Microsoft.Extensions.Hosting
dotnet add package Microsoft.Extensions.AI
````
```

### Setting up your API key

You'll need an Anthropic API key from the [Anthropic Console](<https://console.anthropic.com/settings/keys>).

```
```bash
dotnet user-secrets init
dotnet user-secrets set "ANTHROPIC_API_KEY" "<your key here>"
````
```

### Creating the Client

#### Basic Client Structure

First, let's setup the basic client class in the file `Program.cs`:

```
```csharp
using Anthropic.SDK;
using Microsoft.Extensions.AI;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;
using ModelContextProtocol.Client;
using ModelContextProtocol.Protocol.Transport;

var builder = Host.CreateApplicationBuilder(args);

builder.Configuration
    .AddEnvironmentVariables()
    .AddUserSecrets<Program>();
````
```

This creates the beginnings of a .NET console application that can read the API key from user secrets.

Next, we'll setup the MCP Client:

```
```csharp
var (command, arguments) = GetCommandAndArguments(args);
````
```

```

var clientTransport = new StdioClientTransport(new()
{
    Name = "Demo Server",
    Command = command,
    Arguments = arguments,
});

await using var mcpClient = await McpClientFactory.CreateAsync(clientTransport);

var tools = await mcpClient.ListToolsAsync();
foreach (var tool in tools)
{
    Console.WriteLine($"Connected to server with tools: {tool.Name}");
}
```

```

Add this function at the end of the `Program.cs` file:

```

```csharp
static (string command, string[] arguments) GetCommandAndArguments(string[] args)
{
    return args switch
    {
        [var script] when script.EndsWith(".py") => ("python", args),
        [var script] when script.EndsWith(".js") => ("node", args),
        [var script] when Directory.Exists(script) || (File.Exists(script) &&
script.EndsWith(".csproj")) => ("dotnet", ["run", "--project", script, "--no-build"]),
        _ => throw new NotSupportedException("An unsupported server script was provided.
Supported scripts are .py, .js, or .csproj")
    };
}
```

```

This creates a MCP client that will connect to a server that is provided as a command line argument. It then lists the available tools from the connected server.

#### #### Query processing logic

Now let's add the core functionality for processing queries and handling tool calls:

```

```csharp
using var anthropicClient = new AnthropicClient(new
APIAuthentication(builder.Configuration["ANTHROPIC_API_KEY"]))
    .Messages
    .AsBuilder()
    .UseFunctionInvocation()
    .Build();

var options = new ChatOptions
{
    MaxOutputTokens = 1000,
    ModelId = "claude-3-5-sonnet-20241022",
}
```

```

```

        Tools = [... tools]
    };

Console.ForegroundColor = ConsoleColor.Green;
Console.WriteLine("MCP Client Started!");
Console.ResetColor();

PromptForInput();
while(Console.ReadLine() is string query && !"exit".Equals(query,
StringComparison.OrdinalIgnoreCase))
{
    if (string.IsNullOrEmpty(query))
    {
        PromptForInput();
        continue;
    }

    await foreach (var message in anthropicClient.GetStreamingResponseAsync(query, options))
    {
        Console.Write(message);
    }
    Console.WriteLine();

    PromptForInput();
}

static void PromptForInput()
{
    Console.WriteLine("Enter a command (or 'exit' to quit):");
    Console.ForegroundColor = ConsoleColor.Cyan;
    Console.Write("> ");
    Console.ResetColor();
}
```

```

### ### Key Components Explained

#### #### 1. Client Initialization

- \* The client is initialized using `McpClientFactory.CreateAsync()`, which sets up the transport type and command to run the server.

#### #### 2. Server Connection

- \* Supports Python, Node.js, and .NET servers.
- \* The server is started using the command specified in the arguments.
- \* Configures to use stdio for communication with the server.
- \* Initializes the session and available tools.

#### #### 3. Query Processing

- \* Leverages [Microsoft.Extensions.AI](<https://learn.microsoft.com/dotnet/ai/ai-extensions>) for the chat client.

- \* Configures the `IChatClient` to use automatic tool (function) invocation.
- \* The client reads user input and sends it to the server.
- \* The server processes the query and returns a response.
- \* The response is displayed to the user.

### ### Running the client

To run your client with any MCP server:

```
```bash
dotnet run -- path/to/server.csproj # dotnet server
dotnet run -- path/to/server.py # python server
dotnet run -- path/to/server.js # node server
```

```

## <Note>

If you're continuing the weather tutorial from the server quickstart, your command might look something like this: `dotnet run -- path/to/QuickstartWeatherServer`.

</Note>

The client will:

1. Connect to the specified server
  2. List available tools
  3. Start an interactive chat session where you can:
    - \* Enter queries
    - \* See tool executions
    - \* Get responses from Claude
  4. Exit the session when done

Here's an example of what it should look like it connected to a weather server quickstart:

```
<Frame>
  
</Frame>
```

## Next steps

Check out our gallery of official MCP servers and implementations

View the list of clients that support MCP integrations

Learn how to use LLMs like Claude to speed up your MCP development

Understand how MCP connects clients, servers, and LLMs

# FAQs

---

Explaining MCP and why it matters in simple terms

## What is MCP?

MCP (Model Context Protocol) is a standard way for AI applications and agents to connect to and work with your data sources (e.g. local files, databases, or content repositories) and tools (e.g. GitHub, Google Maps, or Puppeteer).

Think of MCP as a universal adapter for AI applications, similar to what USB-C is for physical devices. USB-C acts as a universal adapter to connect devices to various peripherals and accessories. Similarly, MCP provides a standardized way to connect AI applications to different data and tools.

Before USB-C, you needed different cables for different connections. Similarly, before MCP, developers had to build custom connections to each data source or tool they wanted their AI application to work with—a time-consuming process that often resulted in limited functionality. Now, with MCP, developers can easily add connections to their AI applications, making their applications much more powerful from day one.

## Why does MCP matter?

### For AI application users

MCP means your AI applications can access the information and tools you work with every day, making them much more helpful. Rather than AI being limited to what it already knows about, it can now understand your specific documents, data, and work context.

For example, by using MCP servers, applications can access your personal documents from Google Drive or data about your codebase from GitHub, providing more personalized and contextually relevant assistance.

Imagine asking an AI assistant: "Summarize last week's team meeting notes and schedule follow-ups with everyone."

By using connections to data sources powered by MCP, the AI assistant can:

- Connect to your Google Drive through an MCP server to read meeting notes
- Understand who needs follow-ups based on the notes
- Connect to your calendar through another MCP server to schedule the meetings automatically

### For developers

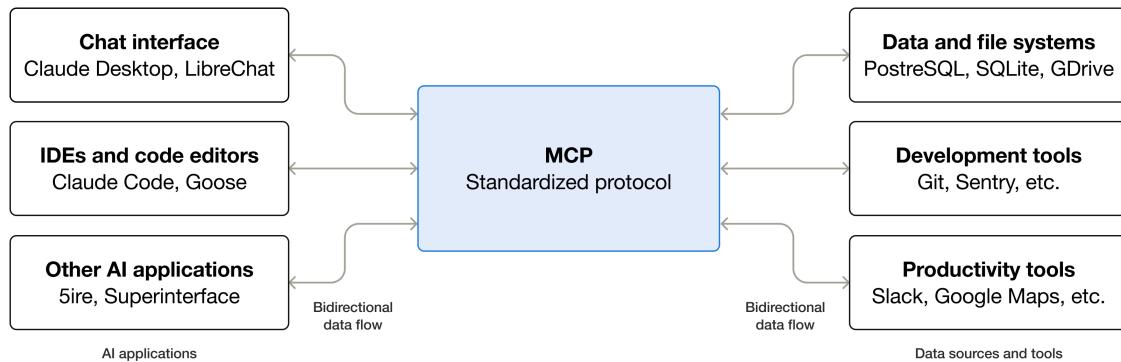
MCP reduces development time and complexity when building AI applications that need to access various data sources. With MCP, developers can focus on building great AI experiences rather than repeatedly creating custom connectors.

Traditionally, connecting applications with data sources required building custom, one-off connections for each data source and each application. This created significant duplicative work—every developer wanting to connect their AI application to Google Drive or Slack needed to build their own connection.

MCP simplifies this by enabling developers to build MCP servers for data sources that are then reusable by various applications. For example, using the open source Google Drive MCP server, many different applications can access data from Google Drive without each developer needing to build a custom connection.

This open source ecosystem of MCP servers means developers can leverage existing work rather than starting from scratch, making it easier to build powerful AI applications that seamlessly integrate with the tools and data sources their users already rely on.

## How does MCP work?



MCP creates a bridge between your AI applications and your data through a straightforward system:

- **MCP servers** connect to your data sources and tools (like Google Drive or Slack)
- **MCP clients** are run by AI applications (like Claude Desktop) to connect them to these servers
- When you give permission, your AI application discovers available MCP servers
- The AI model can then use these connections to read information and take actions

This modular system means new capabilities can be added without changing AI applications themselves—just like adding new accessories to your computer without upgrading your entire system.

## Who creates and maintains MCP servers?

MCP servers are developed and maintained by:

- Developers at Anthropic who build servers for common tools and data sources
- Open source contributors who create servers for tools they use
- Enterprise development teams building servers for their internal systems
- Software providers making their applications AI-ready

Once an open source MCP server is created for a data source, it can be used by any MCP-compatible AI application, creating a growing ecosystem of connections. See our [list of example servers](#), or [get started building your own server](#).

# 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 [modelcontextprotocol.io](#).

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

# Overview

---

MCP provides a standardized way for applications to:

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

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

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

MCP takes some inspiration from the

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

# Key Details

---

## Base Protocol

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

## Features

Servers offer any of the following features to clients:

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

Clients may offer the following features to servers:

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

## Additional Utilities

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

# Security and Trust & Safety

---

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

## Key Principles

### 1. User Consent and Control

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

### 2. Data Privacy

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

### 3. Tool Safety

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

### 4. LLM Sampling Controls

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

## Implementation Guidelines

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

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

## Learn More

---

Explore the detailed specification for each protocol component:

### # Key Changes

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

# Major changes

---

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

## Other schema changes

---

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

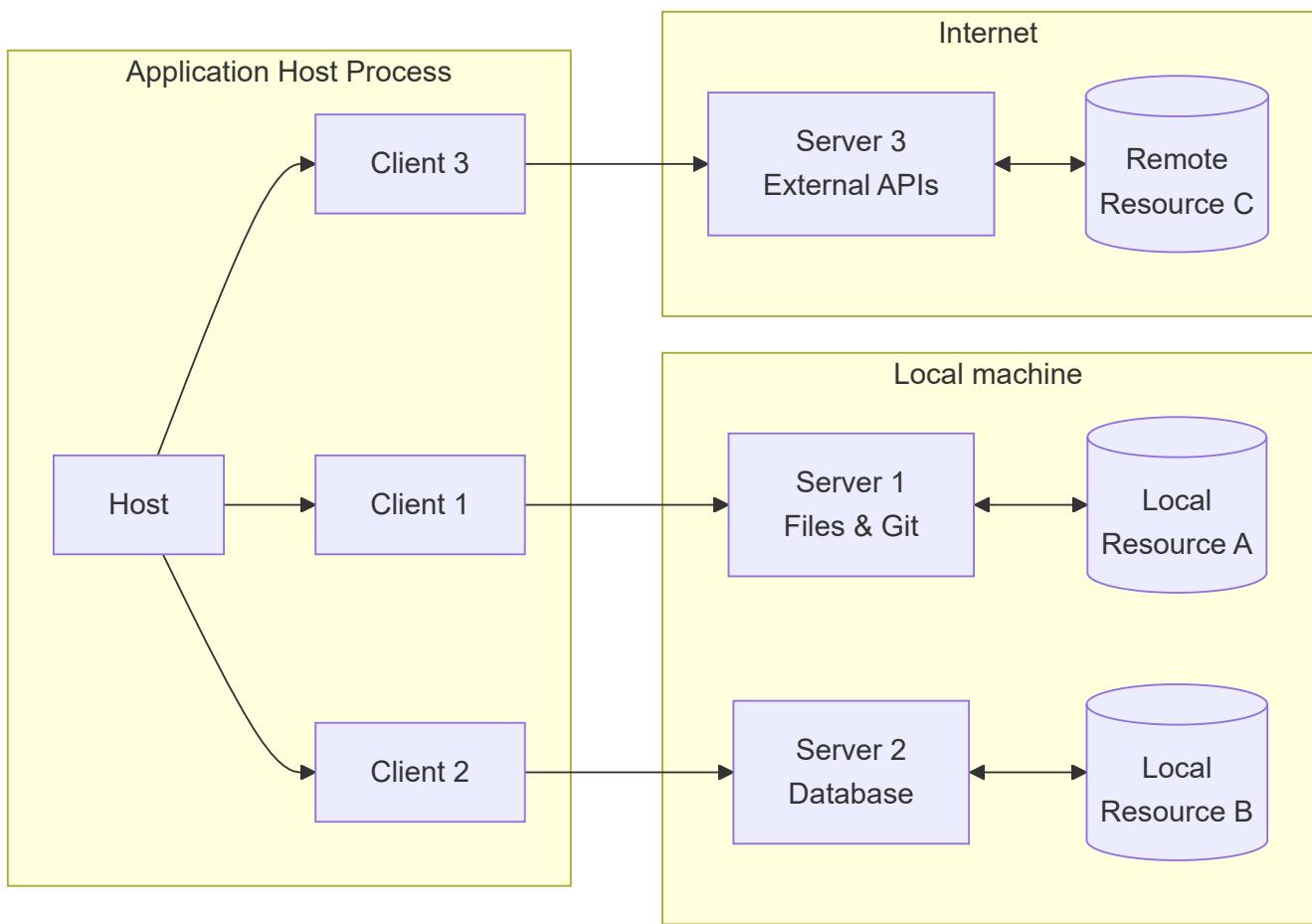
## Full changelog

---

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

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

# Core Components



## Host

The host process acts as the container and coordinator:

- Creates and manages multiple client instances
- Controls client connection permissions and lifecycle
- Enforces security policies and consent requirements
- Handles user authorization decisions
- Coordinates AI/LLM integration and sampling
- Manages context aggregation across clients

## Clients

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

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

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

## Servers

Servers provide specialized context and capabilities:

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

# Design Principles

---

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

## 1. Servers should be extremely easy to build

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

## 2. Servers should be highly composable

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

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

- Servers receive only necessary contextual information
- Full conversation history stays with the host
- Each server connection maintains isolation
- Cross-server interactions are controlled by the host
- Host process enforces security boundaries

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

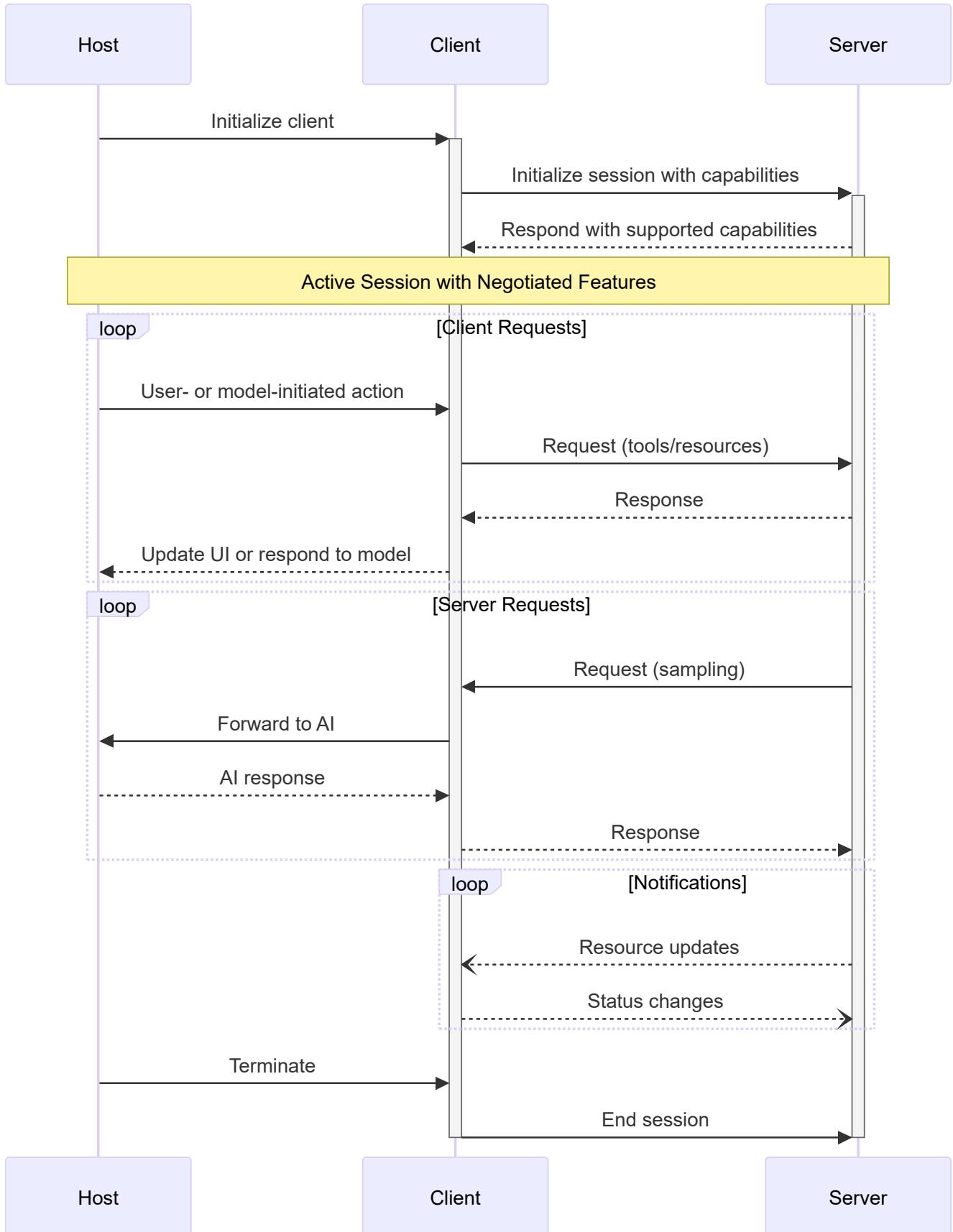
- Core protocol provides minimal required functionality
- Additional capabilities can be negotiated as needed
- Servers and clients evolve independently
- Protocol designed for future extensibility
- Backwards compatibility is maintained

# Capability Negotiation

---

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

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



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

- Implemented [server features](#) must be advertised in the server's capabilities

- Emitting resource subscription notifications requires the server to declare subscription support
- Tool invocation requires the server to declare tool capabilities
- [Sampling](#) requires the client to declare support in its capabilities

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

**Protocol Revision:** 2025-06-18

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

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

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

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

# Messages

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

## Requests

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

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

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

## Responses

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

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

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

## Notifications

Notifications are sent from the client to the server or vice versa, as a one-way message.

The receiver **MUST NOT** send a response.

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

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

# Auth

---

MCP provides an [Authorization](#) framework for use with HTTP.

Implementations using an HTTP-based transport **SHOULD** conform to this specification, whereas implementations using STDIO transport **SHOULD NOT** follow this specification, and instead retrieve credentials from the environment.

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

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

# Schema

---

The full specification of the protocol is defined as a

[TypeScript schema](#).

This is the source of truth for all protocol messages and structures.

There is also a

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

## General fields

### `_meta`

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

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

Additionally, definitions in the [schema](#)

may reserve particular names for purpose-specific metadata, as declared in those definitions.

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

#### Prefix:

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

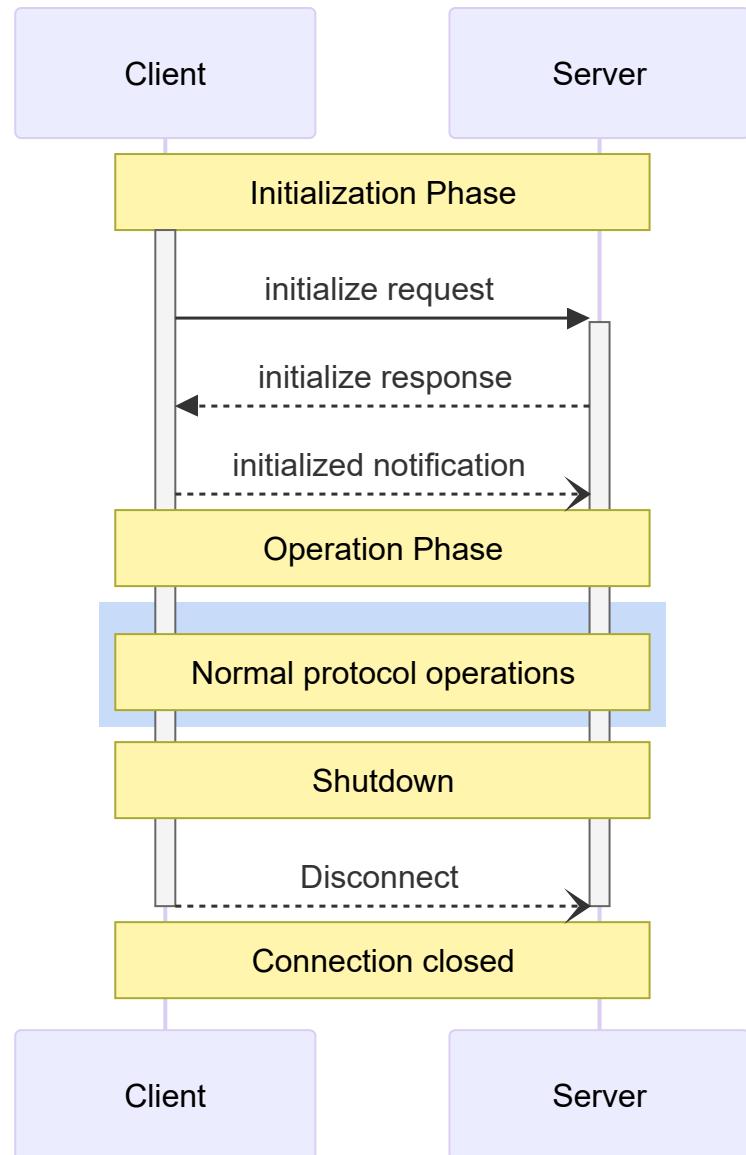
#### Name:

- Unless empty, MUST begin and end with an alphanumeric character (`[a-zA-Z0-9]`).
- MAY contain hyphens (`-`), underscores (`_`), dots (`.`), and alphanumerics in between.

**Protocol Revision:** 2025-06-18

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

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



# Lifecycle Phases

## Initialization

The initialization phase **MUST** be the first interaction between client and server.

During this phase, the client and server:

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

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

- Protocol version supported
- Client capabilities
- Client implementation information

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "initialize",
  "params": {
    "protocolversion": "2024-11-05",
    "capabilities": {
      "roots": {
        "listChanged": true
      },
      "sampling": {},
      "elicitation": {}
    },
    "clientInfo": {
      "name": "ExampleClient",
      "title": "Example Client Display Name",
      "version": "1.0.0"
    }
  }
}
```

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

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "protocolversion": "2024-11-05",
    "capabilities": {
      "logging": {},
      "prompts": {
        "listChanged": true
      },
      "completion": {}
    }
  }
}
```

```

    "resources": {
        "subscribe": true,
        "listChanged": true
    },
    "tools": {
        "listChanged": true
    }
},
"serverInfo": {
    "name": "ExampleServer",
    "title": "Example Server Display Name",
    "version": "1.0.0"
},
"instructions": "Optional instructions for the client"
}
}

```

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

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

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

## Version Negotiation

In the `initialize` request, the client **MUST** send a protocol version it supports.

This **SHOULD** be the *latest* version supported by the client.

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

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

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

For details, see [the Protocol Version Header section in Transports](#).

## Capability Negotiation

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

Key capabilities include:

Category	Capability	Description
Client	<code>roots</code>	Ability to provide filesystem <a href="#">roots</a>
Client	<code>sampling</code>	Support for LLM <a href="#">sampling</a> requests
Client	<code>elicitation</code>	Support for server <a href="#">elicitation</a> requests
Client	<code>experimental</code>	Describes support for non-standard experimental features
Server	<code>prompts</code>	Offers <a href="#">prompt templates</a>
Server	<code>resources</code>	Provides readable <a href="#">resources</a>
Server	<code>tools</code>	Exposes callable <a href="#">tools</a>
Server	<code>logging</code>	Emits structured <a href="#">log messages</a>
Server	<code>completions</code>	Supports argument <a href="#">autocomplete</a>
Server	<code>experimental</code>	Describes support for non-standard experimental features

Capability objects can describe sub-capabilities like:

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

## Operation

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

Both parties **MUST**:

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

## Shutdown

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

## stdio

For the stdio [transport](#), the client **SHOULD** initiate shutdown by:

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

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

## HTTP

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

# Timeouts

---

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

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

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

# Error Handling

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

- Protocol version mismatch
- Failure to negotiate required capabilities
- Request [timeouts](#)

Example initialization error:

```
{  
  "jsonrpc": "2.0",  
  "id": 1,  
  "error": {  
    "code": -32602,  
    "message": "Unsupported protocol version",  
    "data": {  
      "supported": ["2024-11-05"],  
      "requested": "1.0.0"  
    }  
  }  
}  
```# Transports  
  
<div id="enable-section-numbers" />  
  
<Info>**Protocol Revision**: 2025-06-18</Info>
```

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

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

1. [stdio](#stdio), communication over standard in and standard out
2. [Streamable HTTP](#streamable-http)

Clients **\*\*SHOULD\*\*** support stdio whenever possible.

It is also possible for clients and servers to implement [custom transports](#custom-transports) in a pluggable fashion.

## stdio

In the **\*\*stdio\*\*** transport:

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

```

* The server **MUST NOT** write anything to its `stdout` that is not a valid MCP message.
* The client **MUST NOT** write anything to the server's `stdin` that is not a valid MCP
message.

```mermaid
sequenceDiagram
    participant Client
    participant Server Process

    Client->>+Server Process: Launch subprocess
    loop Message Exchange
        Client->>Server Process: Write to stdin
        Server Process->>Client: Write to stdout
        Server Process-->Client: Optional logs on stderr
    end
    Client->>Server Process: Close stdin, terminate subprocess
    deactivate Server Process
```

```

# Streamable HTTP

---

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

In the **Streamable HTTP** transport, the server operates as an independent process that can handle multiple client connections. This transport uses HTTP POST and GET requests.

Server can optionally make use of

[Server-Sent Events](#) (SSE) to stream multiple server messages. This permits basic MCP servers, as well as more feature-rich servers supporting streaming and server-to-client notifications and requests.

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

## Security Warning

When implementing Streamable HTTP transport:

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

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

## Sending Messages to the Server

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

1. The client **MUST** use HTTP POST to send JSON-RPC messages to the MCP endpoint.
2. The client **MUST** include an `Accept` header, listing both `application/json` and `text/event-stream` as supported content types.
3. The body of the POST request **MUST** be a single JSON-RPC *request*, *notification*, or *response*.
4. If the input is a JSON-RPC *response* or *notification*:
  - o If the server accepts the input, the server **MUST** return HTTP status code 202 Accepted with no body.
  - o If the server cannot accept the input, it **MUST** return an HTTP error status code (e.g., 400 Bad Request). The HTTP response body **MAY** comprise a JSON-RPC *error response* that has no `id`.
5. If the input is a JSON-RPC *request*, the server **MUST** either return `Content-Type: text/event-stream`, to initiate an SSE stream, or `Content-Type: application/json`, to return one JSON object. The client **MUST** support both these cases.
6. If the server initiates an SSE stream:

- The SSE stream **SHOULD** eventually include JSON-RPC *response* for the JSON-RPC *request* sent in the POST body.
- The server **MAY** send JSON-RPC *requests* and *notifications* before sending the JSON-RPC *response*. These messages **SHOULD** relate to the originating client *request*.
- The server **SHOULD NOT** close the SSE stream before sending the JSON-RPC *response* for the received JSON-RPC *request*, unless the [session](#) expires.
- After the JSON-RPC *response* has been sent, the server **SHOULD** close the SSE stream.
- Disconnection **MAY** occur at any time (e.g., due to network conditions). Therefore:
  - Disconnection **SHOULD NOT** be interpreted as the client cancelling its request.
  - To cancel, the client **SHOULD** explicitly send an MCP [CancelledNotification](#).
  - To avoid message loss due to disconnection, the server **MAY** make the stream [resumable](#).

## Listening for Messages from the Server

1. The client **MAY** issue an HTTP GET to the MCP endpoint. This can be used to open an SSE stream, allowing the server to communicate to the client, without the client first sending data via HTTP POST.
2. The client **MUST** include an `Accept` header, listing `text/event-stream` as a supported content type.
3. The server **MUST** either return `Content-Type: text/event-stream` in response to this HTTP GET, or else return HTTP 405 Method Not Allowed, indicating that the server does not offer an SSE stream at this endpoint.
4. If the server initiates an SSE stream:
  - The server **MAY** send JSON-RPC *requests* and *notifications* on the stream.
  - These messages **SHOULD** be unrelated to any concurrently-running JSON-RPC *request* from the client.
  - The server **MUST NOT** send a JSON-RPC *response* on the stream **unless resuming** a stream associated with a previous client *request*.
  - The server **MAY** close the SSE stream at any time.
  - The client **MAY** close the SSE stream at any time.

## Multiple Connections

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

- The risk of message loss **MAY** be mitigated by making the stream [resumable](#).

## Resumability and Redelivery

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

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

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

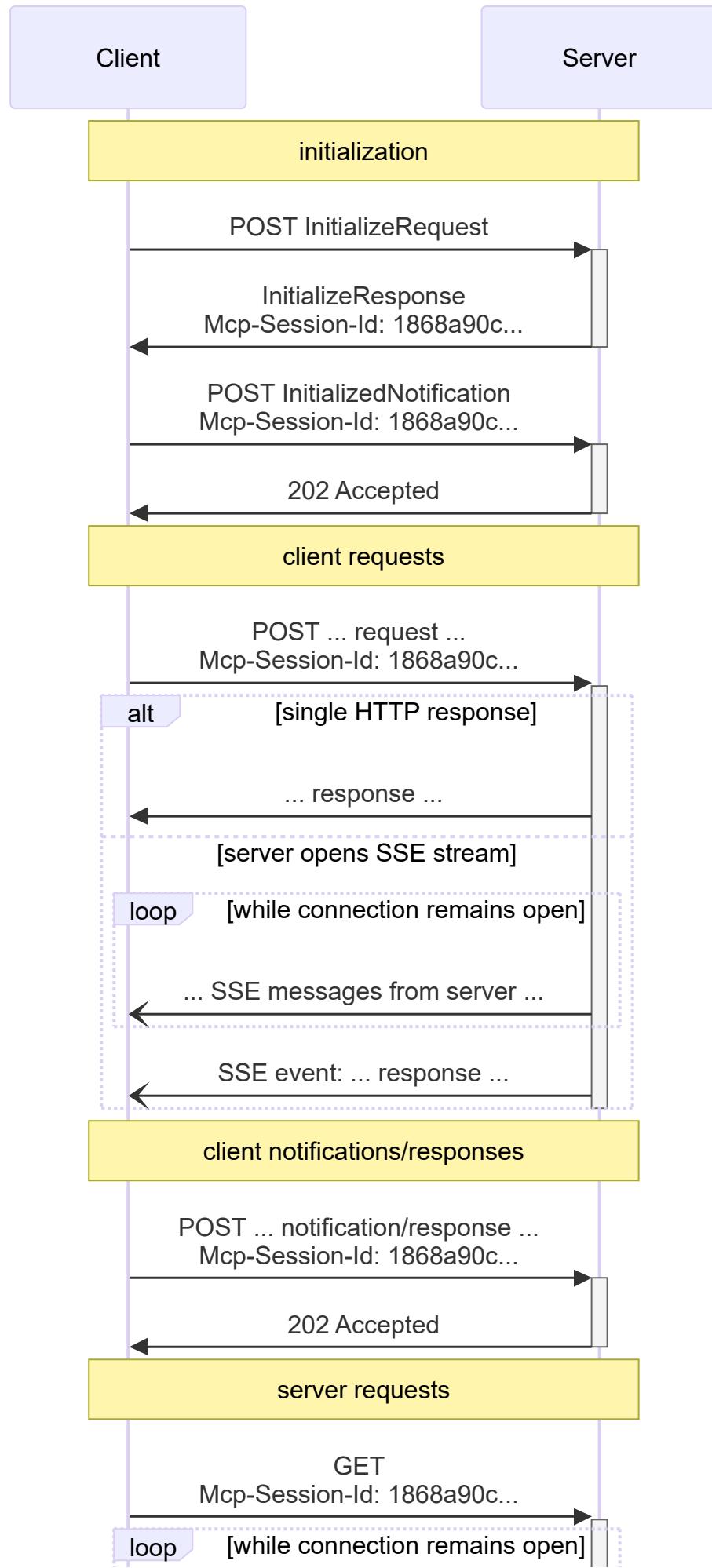
## Session Management

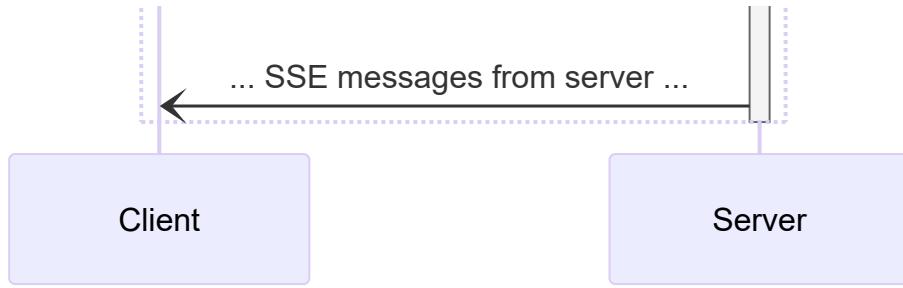
An MCP "session" consists of logically related interactions between a client and a server, beginning with the [initialization phase](#). To support servers which want to establish stateful sessions:

1. A server using the Streamable HTTP transport **MAY** assign a session ID at initialization time, by including it in an `Mcp-session-Id` header on the HTTP response containing the `InitializeResult`.
  - The session ID **SHOULD** be globally unique and cryptographically secure (e.g., a securely generated UUID, a JWT, or a cryptographic hash).
  - The session ID **MUST** only contain visible ASCII characters (ranging from 0x21 to 0x7E).
2. If an `Mcp-session-Id` is returned by the server during initialization, clients using the Streamable HTTP transport **MUST** include it in the `Mcp-session-Id` header on all of their subsequent HTTP requests.
  - Servers that require a session ID **SHOULD** respond to requests without an `Mcp-session-Id` header (other than initialization) with HTTP 400 Bad Request.
3. The server **MAY** terminate the session at any time, after which it **MUST** respond to requests containing that session ID with HTTP 404 Not Found.
4. When a client receives HTTP 404 in response to a request containing an `Mcp-session-Id`, it **MUST** start a new session by sending a new `InitializeRequest` without a session ID attached.

5. Clients that no longer need a particular session (e.g., because the user is leaving the client application) **SHOULD** send an HTTP DELETE to the MCP endpoint with the `Mcp-Session-Id` header, to explicitly terminate the session.
  - o The server **MAY** respond to this request with HTTP 405 Method Not Allowed, indicating that the server does not allow clients to terminate sessions.

## Sequence Diagram





## Protocol Version Header

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

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

The protocol version sent by the client **SHOULD** be the one [negotiated during initialization](#).

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

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

## Backwards Compatibility

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

**Servers** wanting to support older clients should:

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

**Clients** wanting to support older servers should:

1. Accept an MCP server URL from the user, which may point to either a server using the old transport or the new transport.
2. Attempt to POST an `InitializeRequest` to the server URL, with an `Accept` header as defined above:
  - If it succeeds, the client can assume this is a server supporting the new Streamable HTTP transport.
  - If it fails with an HTTP 4xx status code (e.g., 405 Method Not Allowed or 404 Not Found):
    - Issue a GET request to the server URL, expecting that this will open an SSE stream and return an `endpoint` event as the first event.

- When the `endpoint` event arrives, the client can assume this is a server running the old HTTP+SSE transport, and should use that transport for all subsequent communication.

# Custom Transports

---

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

Implementers who choose to support custom transports **MUST** ensure they preserve the JSON-RPC message format and lifecycle requirements defined by MCP. Custom transports

**SHOULD** document their specific connection establishment and message exchange patterns to aid interoperability.  
# Authorization

**Protocol Revision:** 2025-06-18

# Introduction

---

## Purpose and Scope

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

## Protocol Requirements

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

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

## Standards Compliance

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

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

# Authorization Flow

---

## Roles

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

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

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

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

## Overview

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

## Authorization Server Discovery

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

### Authorization Server Location

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

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

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

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

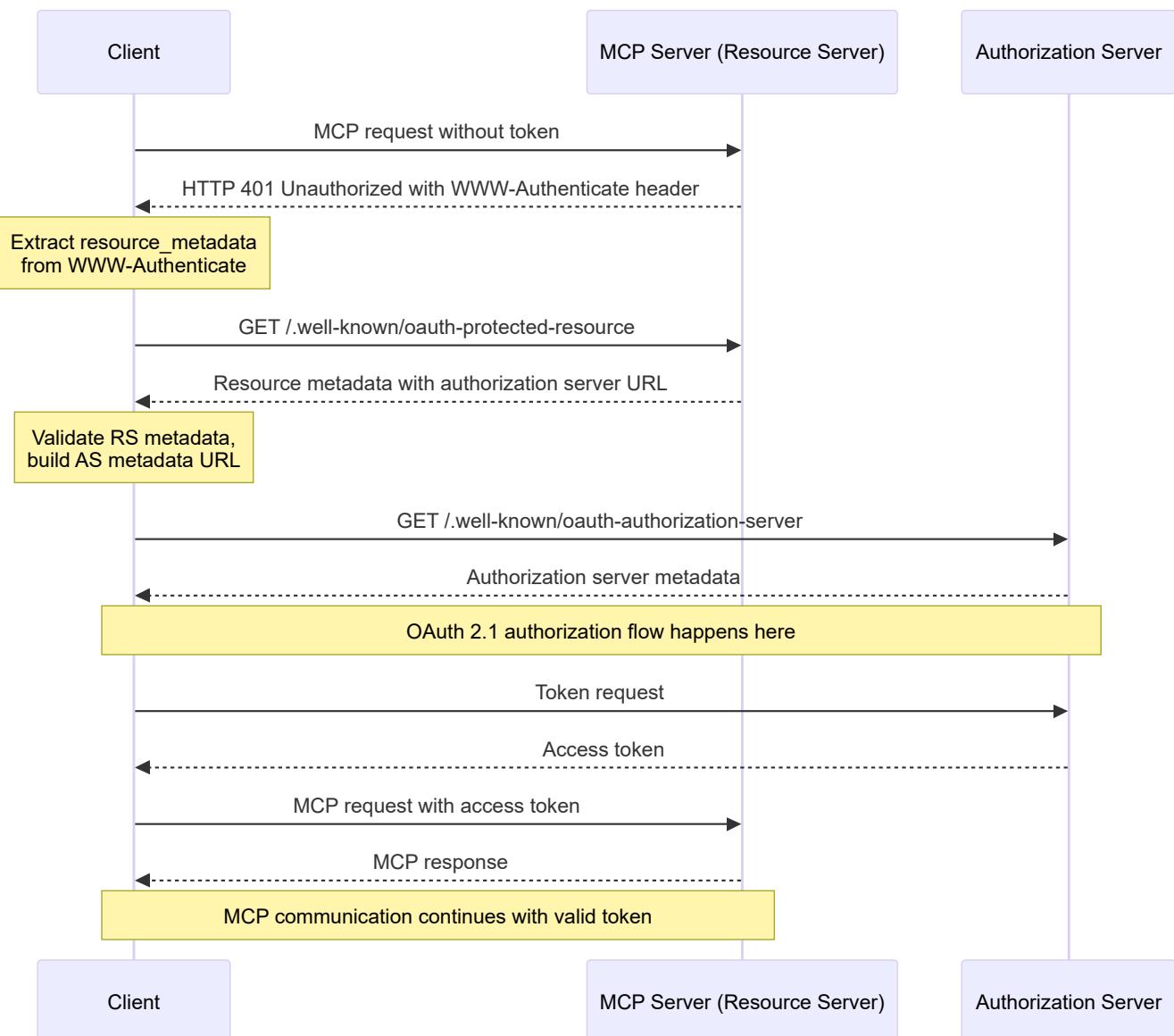
MCP clients **MUST** be able to parse `WWW-Authenticate` headers and respond appropriately to `HTTP 401 Unauthorized` responses from the MCP server.

## Server Metadata Discovery

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

## Sequence Diagram

The following diagram outlines an example flow:



## Dynamic Client Registration

MCP clients and authorization servers **SHOULD** support the OAuth 2.0 Dynamic Client Registration Protocol [RFC7591](#)

to allow MCP clients to obtain OAuth client IDs without user interaction. This provides a standardized way for clients to automatically register with new authorization servers, which is crucial for MCP because:

- Clients may not know all possible MCP servers and their authorization servers in advance.

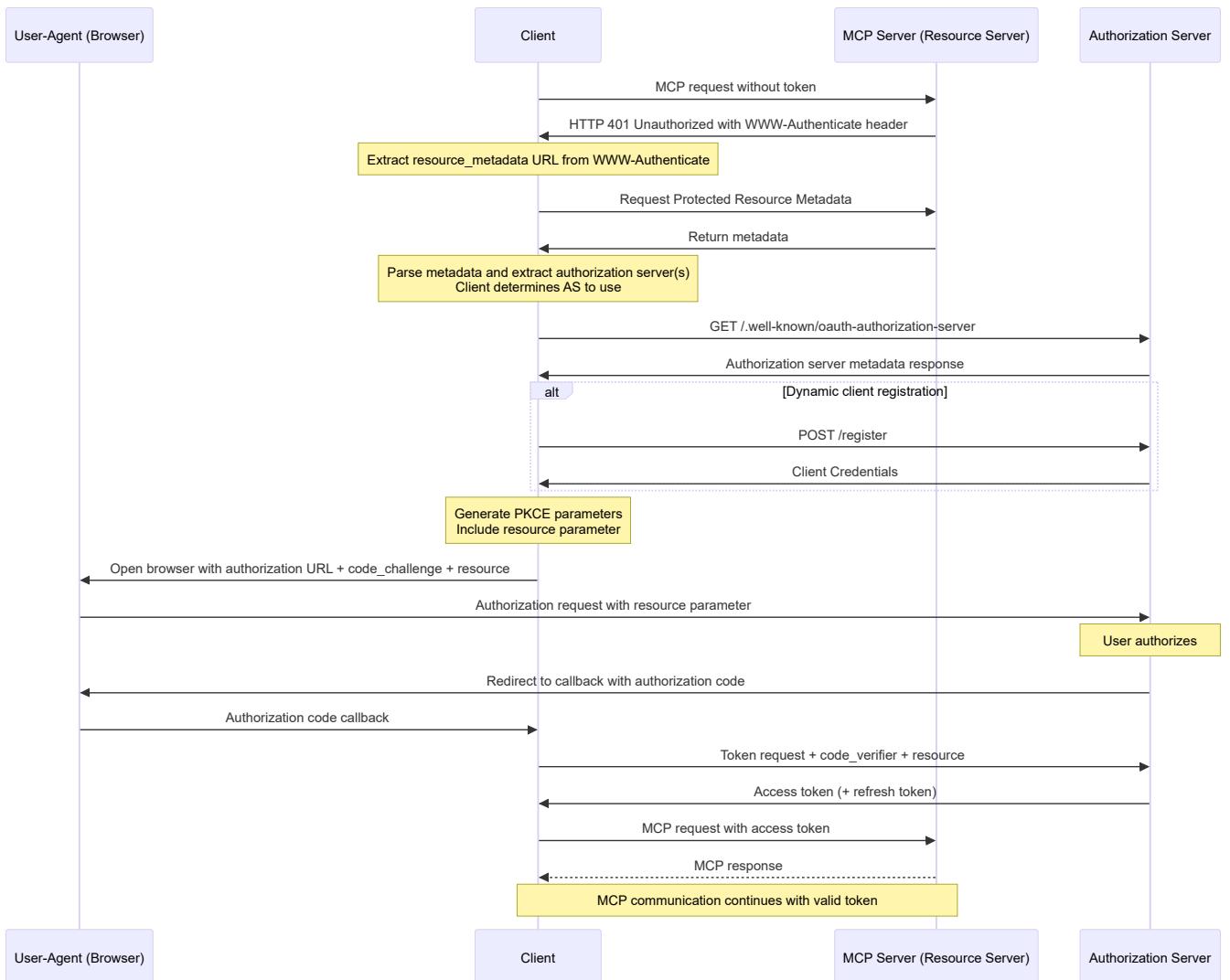
- Manual registration would create friction for users.
- It enables seamless connection to new MCP servers and their authorization servers.
- Authorization servers can implement their own registration policies.

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

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

## Authorization Flow Steps

The complete Authorization flow proceeds as follows:



## Resource Parameter Implementation

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

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

### Canonical Server URI

For the purposes of this specification, the canonical URI of an MCP server is defined as the resource identifier as specified in

[RFC 8707 Section 2](#) and aligns with the `resource` parameter in [RFC 9728](#).

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

Examples of valid canonical URIs:

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

Examples of invalid canonical URIs:

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

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

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

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

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

# Access Token Usage

## Token Requirements

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

Specifically:

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

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

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

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

Example request:

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

## Token Handling

MCP servers, acting in their role as an OAuth 2.1 resource server, **MUST** validate access tokens as described in [OAuth 2.1 Section 5.2](#).

MCP servers **MUST** validate that access tokens were issued specifically for them as the intended audience, according to [RFC 8707 Section 2](#).

If validation fails, servers **MUST** respond according to

[OAuth 2.1 Section 5.3](#)

error handling requirements. Invalid or expired tokens **MUST** receive a HTTP 401 response.

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

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

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

## Error Handling

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

| Status Code | Description  | Usage                                      |
|-------------|--------------|--------------------------------------------|
| 401         | Unauthorized | Authorization required or token invalid    |
| 403         | Forbidden    | Invalid scopes or insufficient permissions |

| Status Code | Description | Usage                           |
|-------------|-------------|---------------------------------|
| 400         | Bad Request | Malformed authorization request |

# Security Considerations

---

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

## Token Audience Binding and Validation

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

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

The [Security Best Practices document](#)

outlines why token audience validation is crucial and why token passthrough is explicitly forbidden.

## Token Theft

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

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

Authorization servers **SHOULD** issue short-lived access tokens to reduce the impact of leaked tokens.

For public clients, authorization servers **MUST** rotate refresh tokens as described in [OAuth 2.1 Section 4.3.1 "Token Endpoint Extension"](#).

## Communication Security

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

Specifically:

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

## Authorization Code Protection

An attacker who has gained access to an authorization code contained in an authorization response can try to redeem the authorization code for an access token or otherwise make use of the authorization code.

(Further described in [OAuth 2.1 Section 7.5](#))

To mitigate this, MCP clients **MUST** implement PKCE according to [OAuth 2.1 Section 7.5.2](#).

PKCE helps prevent authorization code interception and injection attacks by requiring clients to create a secret verifier-challenge pair, ensuring that only the original requestor can exchange an authorization code for tokens.

## Open Redirection

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

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

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

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

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

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

## Confused Deputy Problem

Attackers can exploit MCP servers acting as intermediaries to third-party APIs, leading to [confused deputy vulnerabilities](#).

By using stolen authorization codes, they can obtain access tokens without user consent.

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

## Access Token Privilege Restriction

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

This vulnerability has two critical dimensions:

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

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

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

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

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

MCP clients **MUST** implement and use the `resource` parameter as defined in [RFC 8707 - Resource Indicators for OAuth 2.0](#)

to explicitly specify the target resource for which the token is being requested. This requirement aligns with the recommendation in

[RFC 9728 Section 7.4](#). This ensures that access tokens are bound to their intended resources and cannot be misused across different services.

## # Security Best Practices

# Introduction

---

## Purpose and Scope

This document provides security considerations for the Model Context Protocol (MCP), complementing the MCP Authorization specification. This document identifies security risks, attack vectors, and best practices specific to MCP implementations.

The primary audience for this document includes developers implementing MCP authorization flows, MCP server operators, and security professionals evaluating MCP-based systems. This document should be read alongside the MCP Authorization specification and [OAuth 2.0 security best practices](#).

# Attacks and Mitigations

---

This section gives a detailed description of attacks on MCP implementations, along with potential countermeasures.

## Confused Deputy Problem

Attackers can exploit MCP servers proxying other resource servers, creating "[confused deputy](#)" vulnerabilities.

### Terminology

#### MCP Proxy Server

: An MCP server that connects MCP clients to third-party APIs, offering MCP features while delegating operations and acting as a single OAuth client to the third-party API server.

#### Third-Party Authorization Server

: Authorization server that protects the third-party API. It may lack dynamic client registration support, requiring MCP proxy to use a static client ID for all requests.

#### Third-Party API

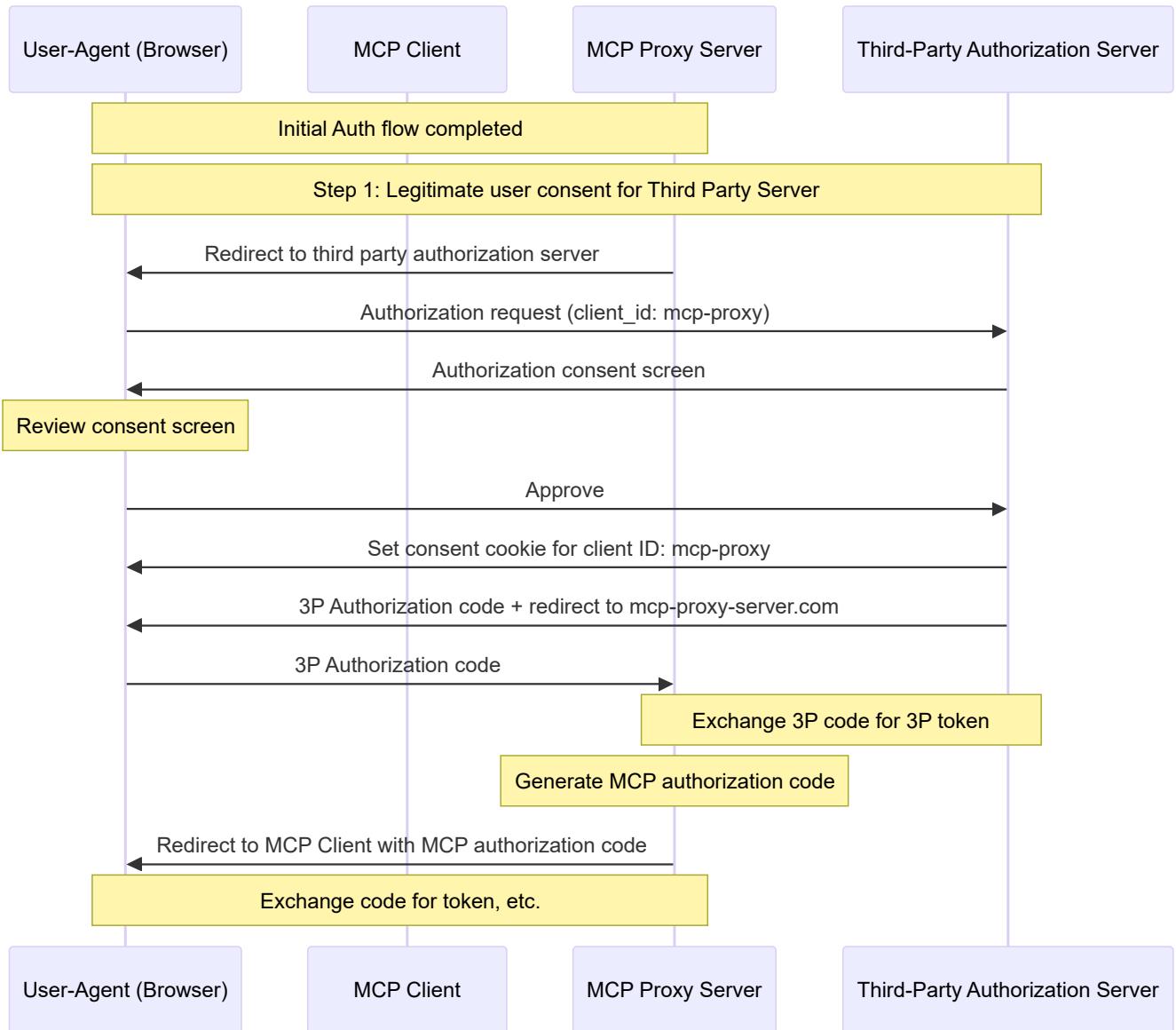
: The protected resource server that provides the actual API functionality. Access to this API requires tokens issued by the third-party authorization server.

#### Static Client ID

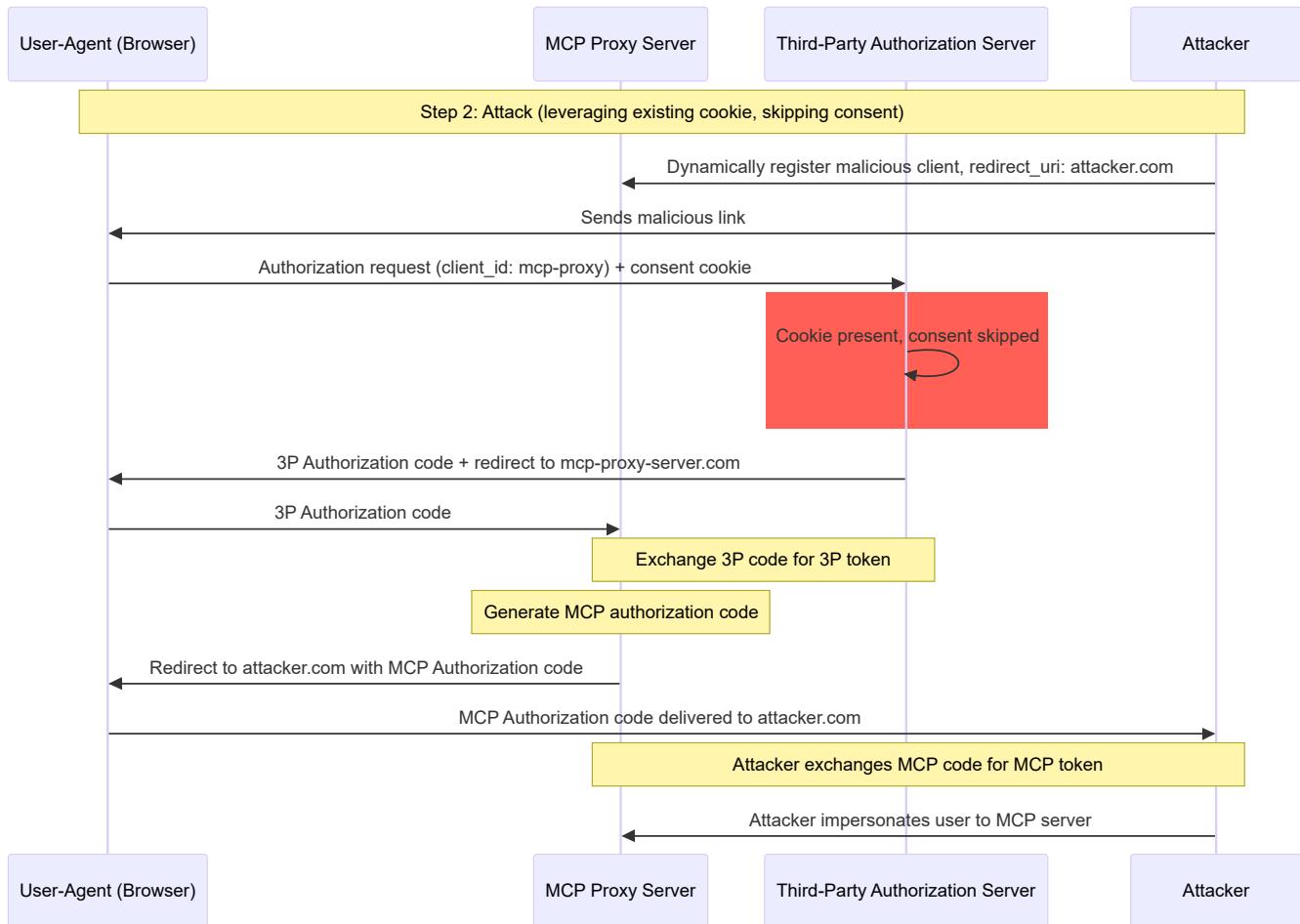
: A fixed OAuth 2.0 client identifier used by the MCP proxy server when communicating with the third-party authorization server. This Client ID refers to the MCP server acting as a client to the Third-Party API. It is the same value for all MCP server to Third-Party API interactions regardless of which MCP client initiated the request.

## Architecture and Attack Flows

### Normal OAuth proxy usage (preserves user consent)



## Malicious OAuth proxy usage (skips user consent)



## Attack Description

When an MCP proxy server uses a static client ID to authenticate with a third-party authorization server that does not support dynamic client registration, the following attack becomes possible:

1. A user authenticates normally through the MCP proxy server to access the third-party API
2. During this flow, the third-party authorization server sets a cookie on the user agent indicating consent for the static client ID
3. An attacker later sends the user a malicious link containing a crafted authorization request which contains a malicious redirect URI along with a new dynamically registered client ID
4. When the user clicks the link, their browser still has the consent cookie from the previous legitimate request
5. The third-party authorization server detects the cookie and skips the consent screen
6. The MCP authorization code is redirected to the attacker's server (specified in the crafted redirect\_uri during dynamic client registration)
7. The attacker exchanges the stolen authorization code for access tokens for the MCP server without the user's explicit approval
8. Attacker now has access to the third-party API as the compromised user

## Mitigation

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).

## Token Passthrough

"Token passthrough" is an anti-pattern where an MCP server accepts tokens from an MCP client without validating that the tokens were properly issued *to the MCP server* and "passing them through" to the downstream API.

## Risks

Token passthrough is explicitly forbidden in the [authorization specification](#) as it introduces a number of security risks, that include:

- **Security Control Circumvention**

- The MCP Server or downstream APIs might implement important security controls like rate limiting, request validation, or traffic monitoring, that depend on the token audience or other credential constraints. If clients can obtain and use tokens directly with the downstream APIs without the MCP server validating them properly or ensuring that the tokens are issued for the right service, they bypass these controls.

- **Accountability and Audit Trail Issues**

- The MCP Server will be unable to identify or distinguish between MCP Clients when clients are calling with an upstream-issued access token which may be opaque to the MCP Server.
- The downstream Resource Server's logs may show requests that appear to come from a different source with a different identity, rather than the MCP server that is actually forwarding the tokens.
- Both factors make incident investigation, controls, and auditing more difficult.
- If the MCP Server passes tokens without validating their claims (e.g., roles, privileges, or audience) or other metadata, a malicious actor in possession of a stolen token can use the server as a proxy for data exfiltration.

- **Trust Boundary Issues**

- The downstream Resource Server grants trust to specific entities. This trust might include assumptions about origin or client behavior patterns. Breaking this trust boundary could lead to unexpected issues.
- If the token is accepted by multiple services without proper validation, an attacker compromising one service can use the token to access other connected services.

- **Future Compatibility Risk**

- Even if an MCP Server starts as a "pure proxy" today, it might need to add security controls later. Starting with proper token audience separation makes it easier to evolve the security model.

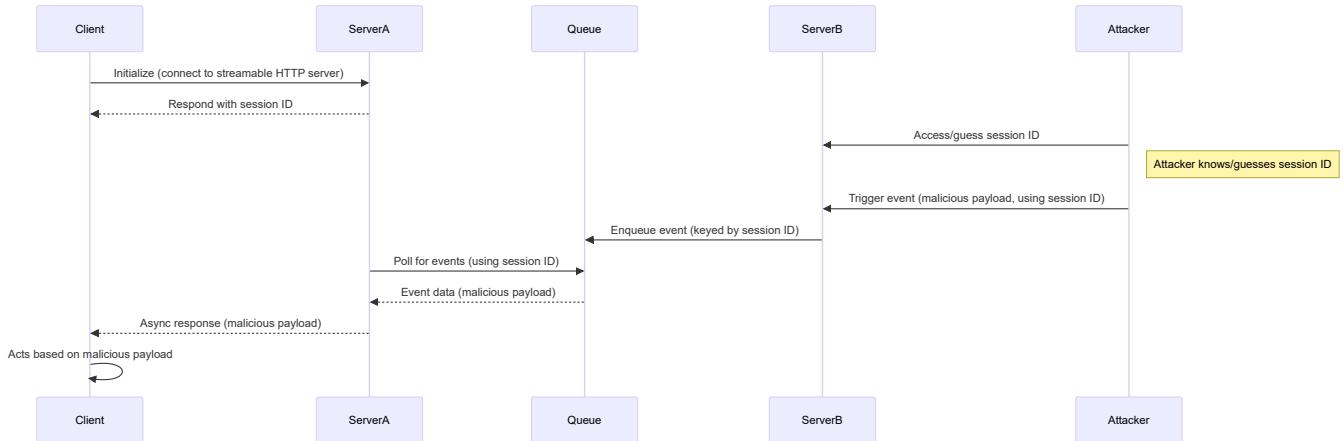
## Mitigation

MCP servers **MUST NOT** accept any tokens that were not explicitly issued for the MCP server.

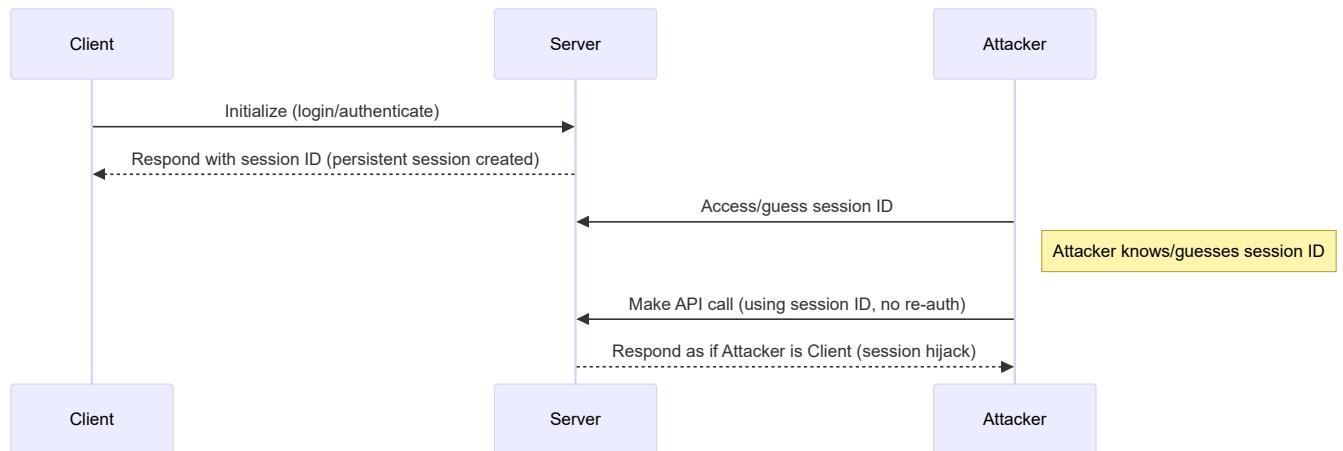
# Session Hijacking

Session hijacking is an attack vector where a client is provided a session ID by the server, and an unauthorized party is able to obtain and use that same session ID to impersonate the original client and perform unauthorized actions on their behalf.

## Session Hijack Prompt Injection



## Session Hijack Impersonation



## Attack Description

When you have multiple stateful HTTP servers that handle MCP requests, the following attack vectors are possible:

### Session Hijack Prompt Injection

1. The client connects to **Server A** and receives a session ID.
2. The attacker obtains an existing session ID and sends a malicious event to **Server B** with said session ID.
  - o When a server supports [redelivery/resumable streams](#), deliberately terminating the request before receiving the response could lead to it being resumed by the original client via the GET request for server sent events.
  - o If a particular server initiates server sent events as a consequence of a tool call such as a [notifications/tools/list\\_changed](#), where it is possible to affect the tools that are offered by the server, a client could end up with tools that they were not aware were enabled.

3. **Server B** enqueues the event (associated with session ID) into a shared queue.
4. **Server A** polls the queue for events using the session ID and retrieves the malicious payload.
5. **Server A** sends the malicious payload to the client as an asynchronous or resumed response.
6. The client receives and acts on the malicious payload, leading to potential compromise.

## Session Hijack Impersonation

1. The MCP client authenticates with the MCP server, creating a persistent session ID.
2. The attacker obtains the session ID.
3. The attacker makes calls to the MCP server using the session ID.
4. MCP server does not check for additional authorization and treats the attacker as a legitimate user, allowing unauthorized access or actions.

## Mitigation

To prevent session hijacking and event injection attacks, the following mitigations should be implemented:

MCP servers that implement authorization **MUST** verify all inbound requests.

MCP Servers **MUST NOT** use sessions for authentication.

MCP servers **MUST** use secure, non-deterministic session IDs.

Generated session IDs (e.g., UUIDs) **SHOULD** use secure random number generators. Avoid predictable or sequential session identifiers that could be guessed by an attacker. Rotating or expiring session IDs can also reduce the risk.

MCP servers **SHOULD** bind session IDs to user-specific information.

When storing or transmitting session-related data (e.g., in a queue), combine the session ID with information unique to the authorized user, such as their internal user ID. Use a key format like `<user_id>:<session_id>`.

This ensures that even if an attacker guesses a session ID, they cannot impersonate another user as the user ID is derived from the user token and not provided by the client.

MCP servers can optionally leverage additional unique identifiers.

**Protocol Revision:** 2025-06-18

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

# Cancellation Flow

---

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

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

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

# Behavior Requirements

---

1. Cancellation notifications **MUST** only reference requests that:

- Were previously issued in the same direction
- Are believed to still be in-progress

2. The `initialize` request **MUST NOT** be cancelled by clients

3. Receivers of cancellation notifications **SHOULD**:

- Stop processing the cancelled request
- Free associated resources
- Not send a response for the cancelled request

4. Receivers **MAY** ignore cancellation notifications if:

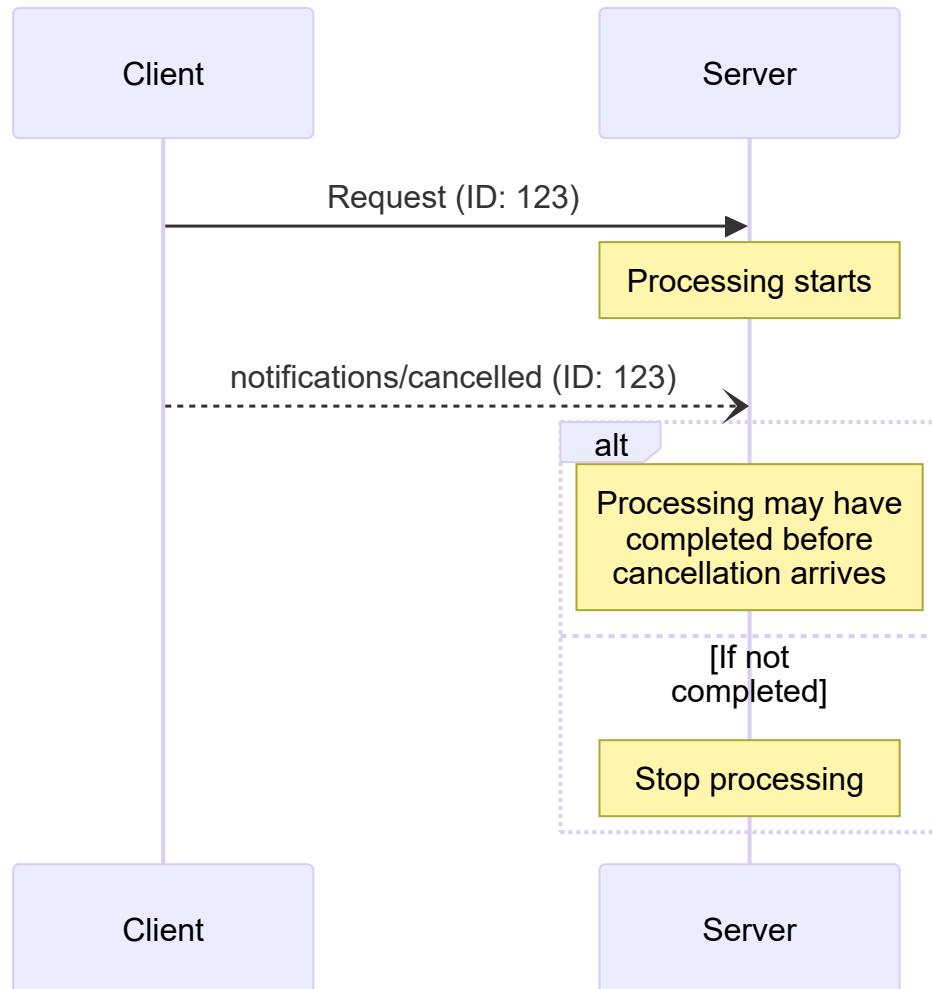
- The referenced request is unknown
- Processing has already completed
- The request cannot be cancelled

5. The sender of the cancellation notification **SHOULD** ignore any response to the request that arrives afterward

# Timing Considerations

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

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



## Implementation Notes

---

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

# Error Handling

---

Invalid cancellation notifications **SHOULD** be ignored:

- Unknown request IDs
- Already completed requests
- Malformed notifications

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

**Ping**  
**Protocol Revision:** 2025-06-18

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

# Overview

---

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

## Message Format

---

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

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

# Behavior Requirements

---

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

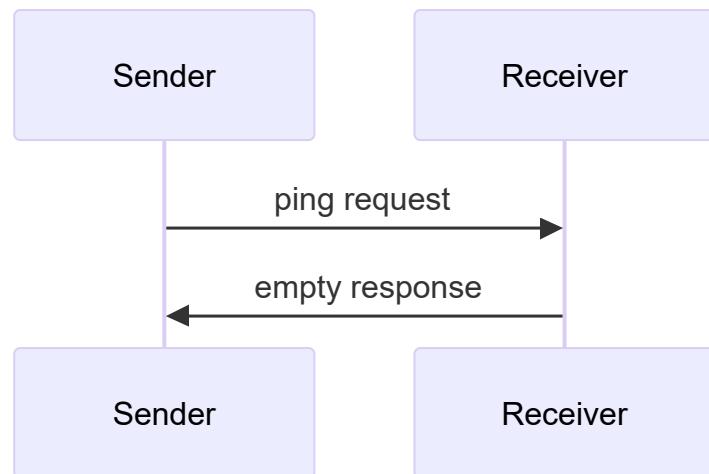
```
{  
  "jsonrpc": "2.0",  
  "id": "123",  
  "result": {}  
}
```

2. If no response is received within a reasonable timeout period, the sender **MAY**:

- o Consider the connection stale
- o Terminate the connection
- o Attempt reconnection procedures

## Usage Patterns

---



## Implementation Considerations

---

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

# Error Handling

---

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

**Protocol Revision:** 2025-06-18

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

# Progress Flow

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

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

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

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

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

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

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

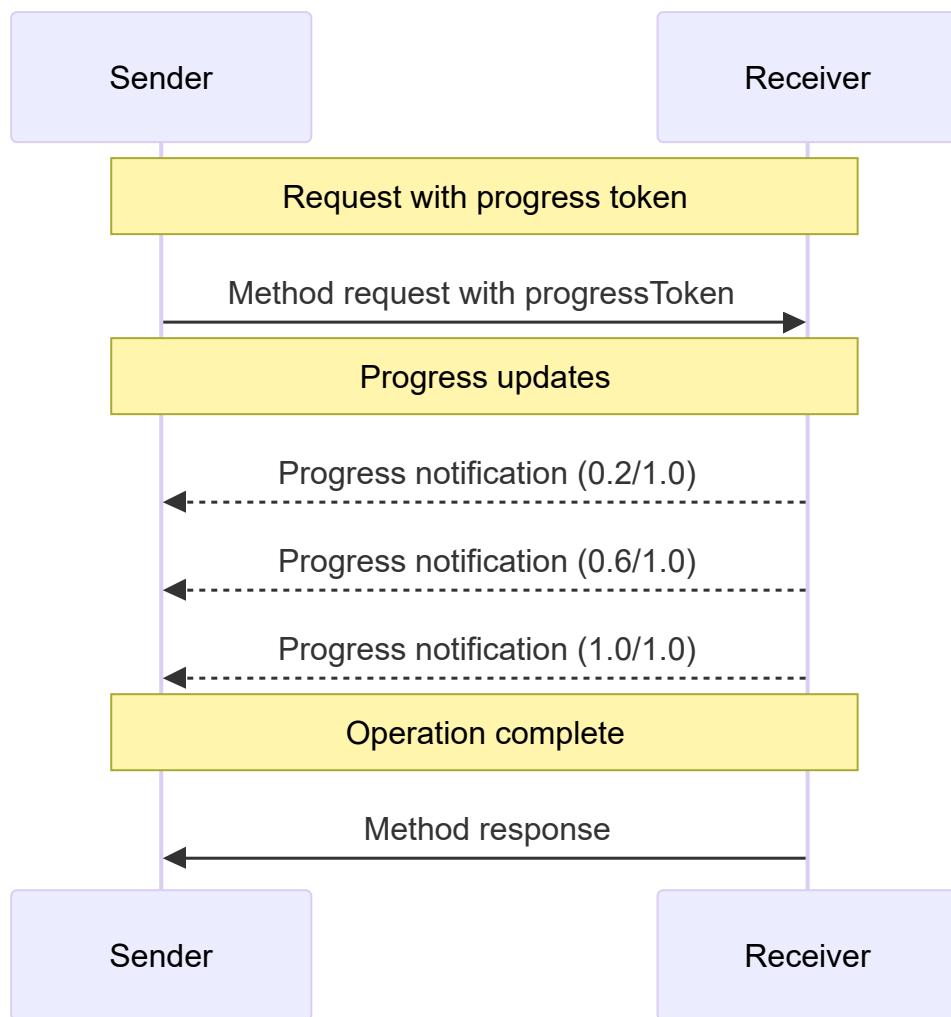
# Behavior Requirements

1. Progress notifications **MUST** only reference tokens that:

- Were provided in an active request
- Are associated with an in-progress operation

2. Receivers of progress requests **MAY**:

- Choose not to send any progress notifications
- Send notifications at whatever frequency they deem appropriate
- Omit the total value if unknown



# Implementation Notes

---

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

**Protocol Revision:** 2025-06-18

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

## User Interaction Model

---

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

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

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

# Capabilities

---

Clients that support roots **MUST** declare the `roots` capability during [initialization](#):

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

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

# Protocol Messages

---

## Listing Roots

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

**Request:**

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

**Response:**

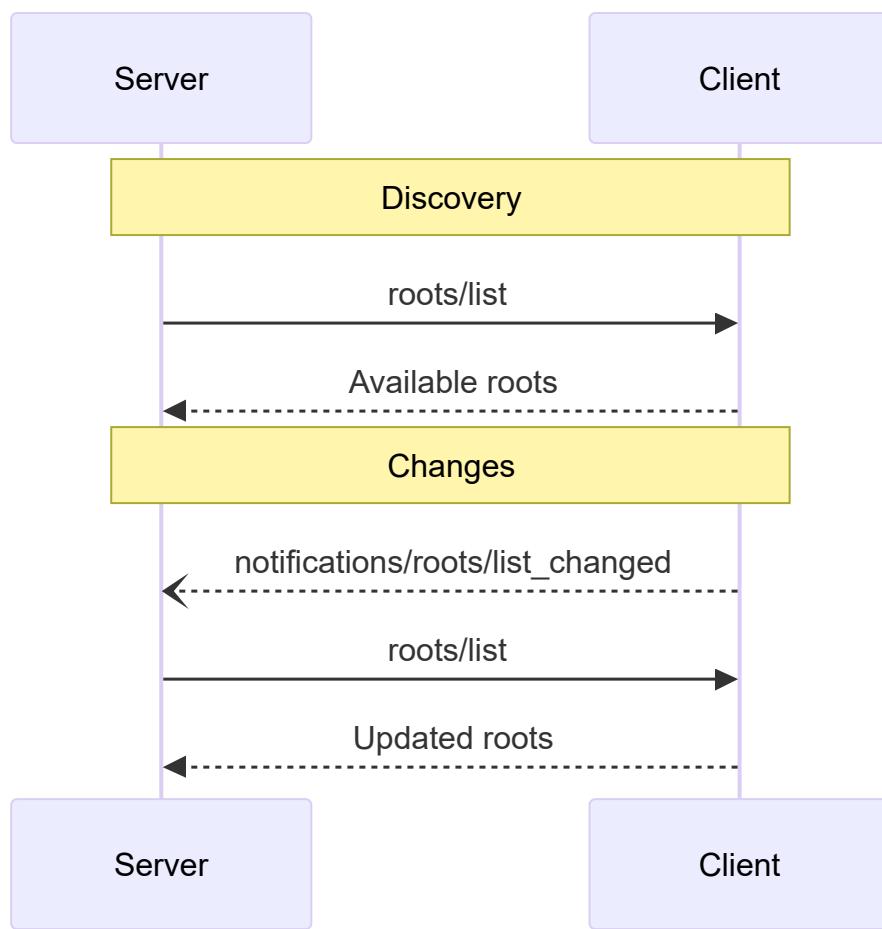
```
{  
  "jsonrpc": "2.0",  
  "id": 1,  
  "result": {  
    "roots": [  
      {  
        "uri": "file:///home/user/projects/myproject",  
        "name": "My Project"  
      }  
    ]  
  }  
}
```

## Root List Changes

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

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

## Message Flow



# Data Types

---

## Root

A root definition includes:

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

Example roots for different use cases:

## Project Directory

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

## Multiple Repositories

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

# Error Handling

---

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

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

Example error:

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

# Security Considerations

---

## 1. Clients **MUST**:

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

## 2. Servers **SHOULD**:

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

# Implementation Guidelines

---

## 1. Clients **SHOULD:**

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

## 2. Servers **SHOULD:**

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

**Protocol Revision:** 2025-06-18

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.

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

Applications **SHOULD**:

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

# Capabilities

---

Clients that support sampling **MUST** declare the `sampling` capability during [initialization](#):

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

# Protocol Messages

## Creating Messages

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

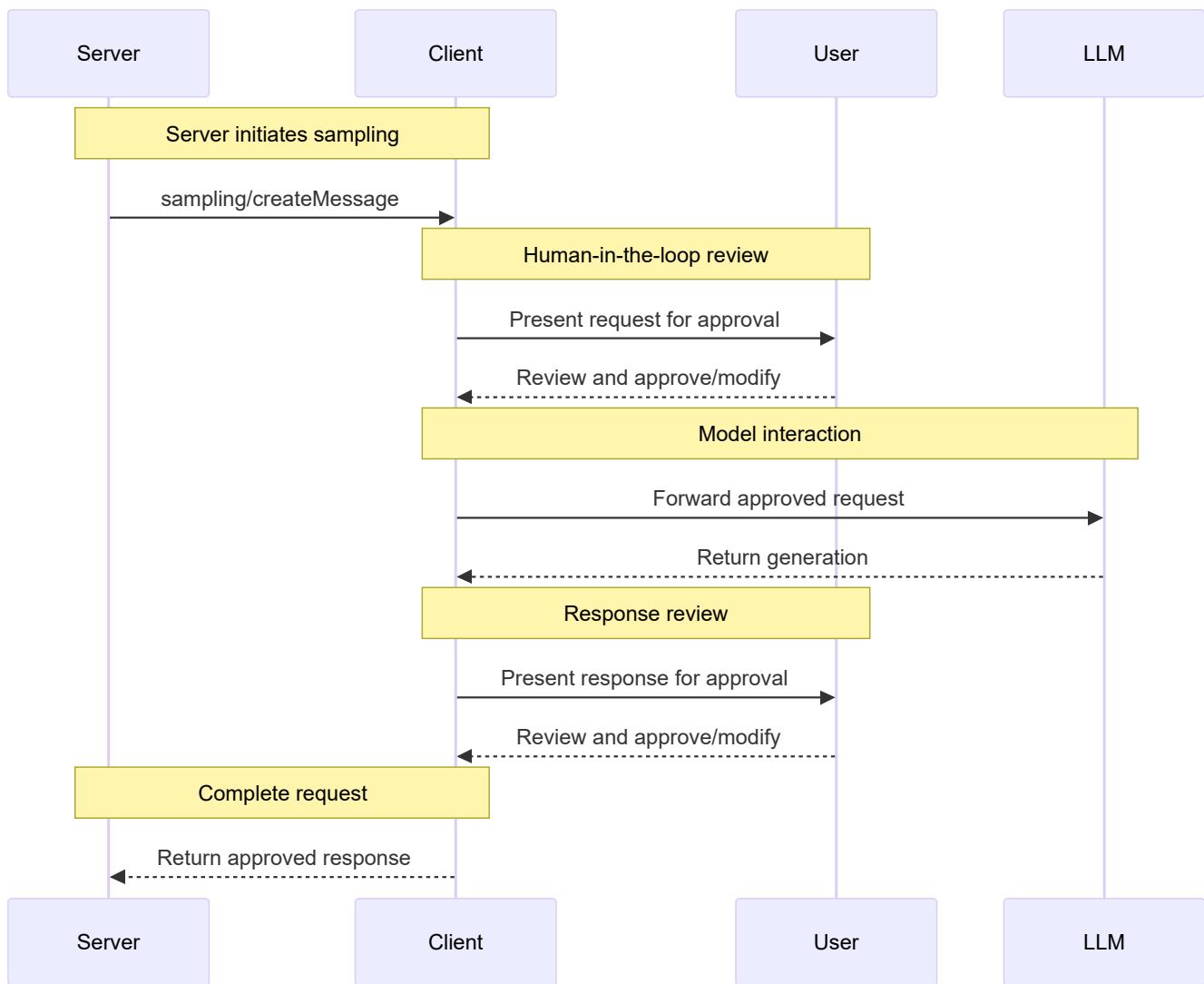
### Request:

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

### Response:

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

# Message Flow



# Data Types

## Messages

Sampling messages can contain:

### Text Content

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

### Image Content

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

### Audio Content

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

## Model Preferences

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

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

### Capability Priorities

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

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

## Model Hints

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

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

For example:

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

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

# Error Handling

---

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

Example error:

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

# Security Considerations

---

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

**Protocol Revision:** 2025-06-18

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

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

# User Interaction Model

---

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

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

For trust & safety and security:

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

Applications **SHOULD**:

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

# Capabilities

---

Clients that support elicitation **MUST** declare the `elicitation` capability during [initialization](#):

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

# Protocol Messages

## Creating Elicitation Requests

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

### Simple Text Request

Request:

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

Response:

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

### Structured Data Request

Request:

```
{  
    "jsonrpc": "2.0",  
    "id": 2,  
    "method": "elicitation/create",  
    "params": {  
        "message": "Please provide your contact information",  
    }  
}
```

```

"requestedSchema": {
    "type": "object",
    "properties": {
        "name": {
            "type": "string",
            "description": "Your full name"
        },
        "email": {
            "type": "string",
            "format": "email",
            "description": "Your email address"
        },
        "age": {
            "type": "number",
            "minimum": 18,
            "description": "Your age"
        }
    },
    "required": ["name", "email"]
}
}
}

```

### Response:

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

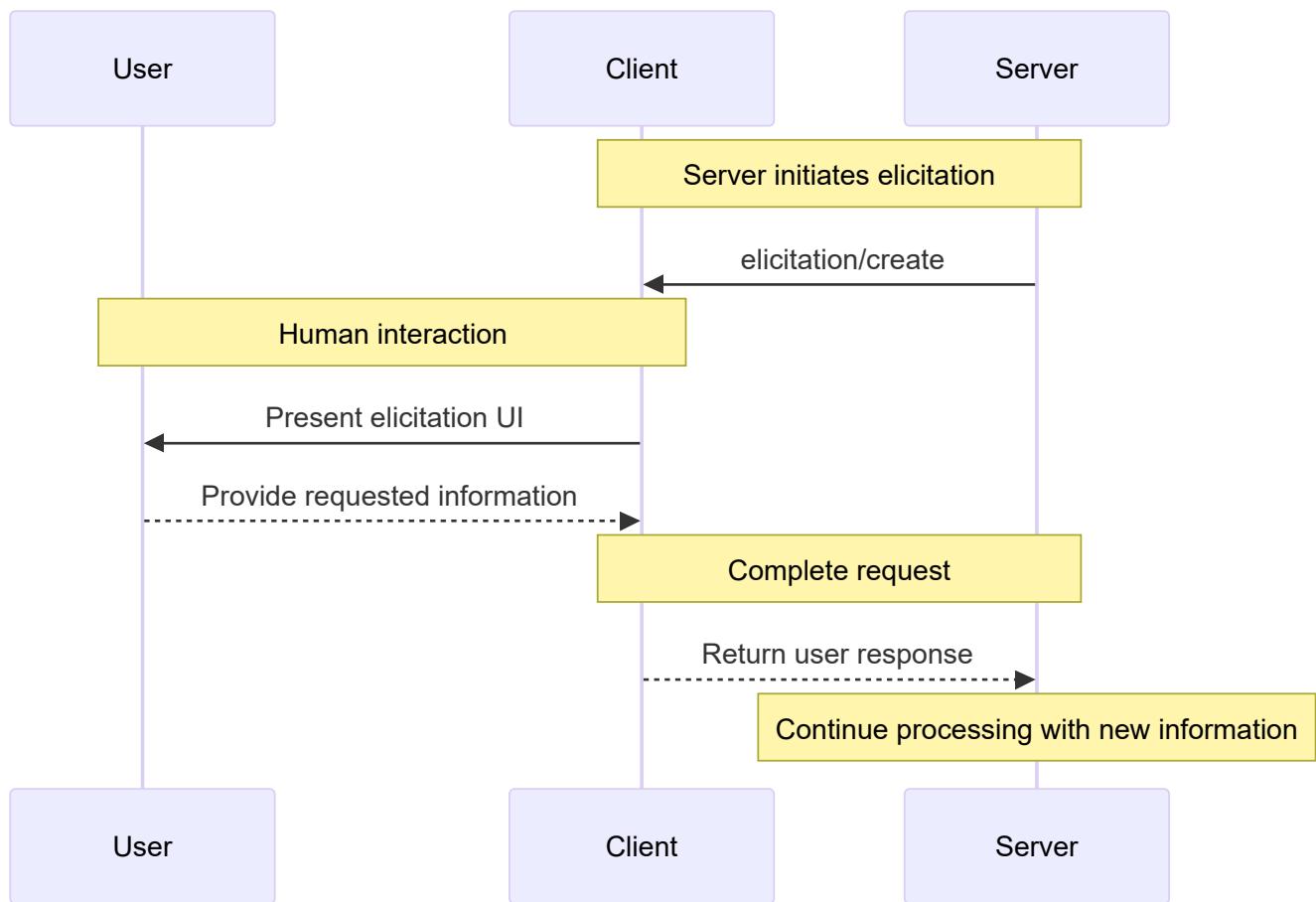
### Reject Response Example:

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

### Cancel Response Example:

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

# Message Flow



# Request Schema

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

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

## Supported Schema Types

The schema is restricted to these primitive types:

### 1. String Schema

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

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

### 2. Number Schema

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

### 3. Boolean Schema

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

#### 4. Enum Schema

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

Clients can use this schema to:

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

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

# Response Actions

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

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

The three response actions are:

1. **Accept** (`action: "accept"`): User explicitly approved and submitted with data
  - o The `content` field contains the submitted data matching the requested schema
  - o Example: User clicked "Submit", "OK", "Confirm", etc.
2. **Decline** (`action: "decline"`): User explicitly declined the request
  - o The `content` field is typically omitted
  - o Example: User clicked "Reject", "Decline", "No", etc.
3. **Cancel** (`action: "cancel"`): User dismissed without making an explicit choice
  - o The `content` field is typically omitted
  - o Example: User closed the dialog, clicked outside, pressed Escape, etc.

Servers should handle each state appropriately:

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

# Security Considerations

---

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

# Server Features - Overview

**Protocol Revision:** 2025-06-18

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

**Protocol Revision:** 2025-06-18

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

# User Interaction Model

---

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

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

For example, as slash commands:



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

# Capabilities

---

Servers that support prompts **MUST** declare the `prompts` capability during [initialization](#):

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

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

# Protocol Messages

## Listing Prompts

To retrieve available prompts, clients send a `prompts/list` request. This operation supports [pagination](#).

### Request:

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

### Response:

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

## Getting a Prompt

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

### Request:

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

## Response:

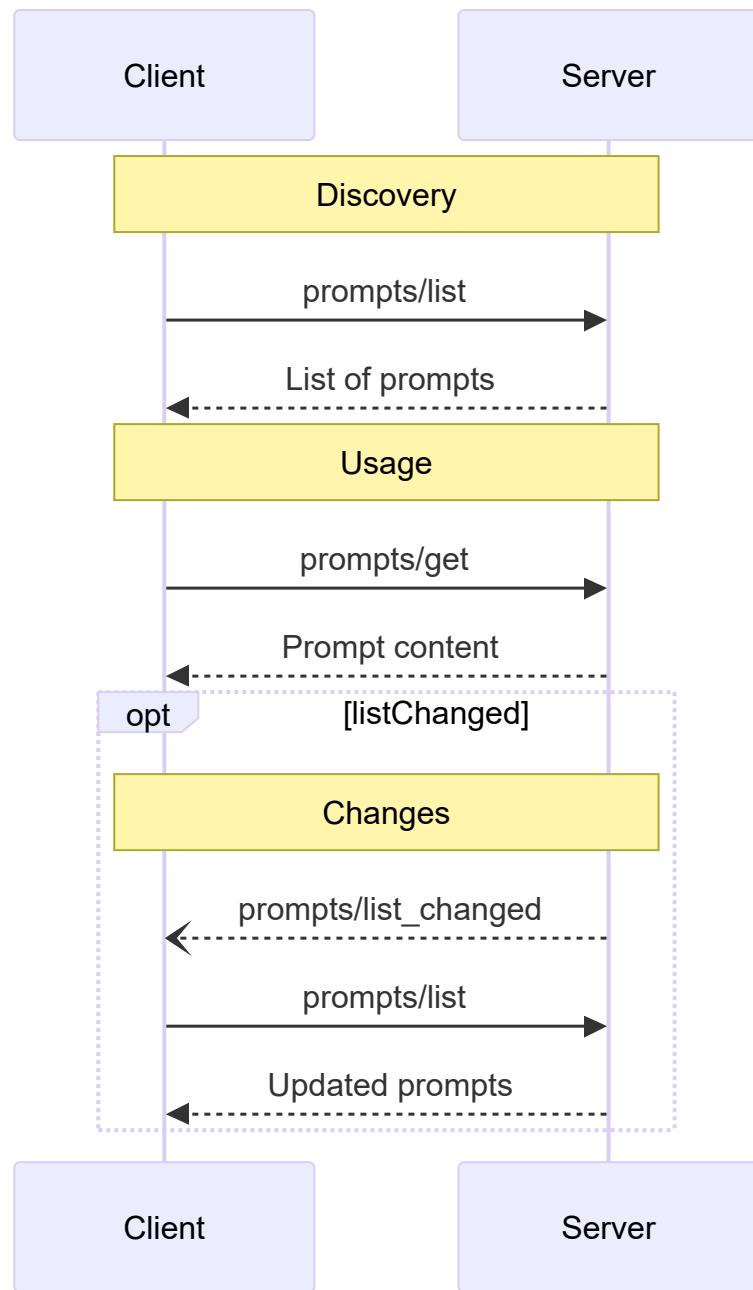
```
{  
    "jsonrpc": "2.0",  
    "id": 2,  
    "result": {  
        "description": "Code review prompt",  
        "messages": [  
            {  
                "role": "user",  
                "content": {  
                    "type": "text",  
                    "text": "Please review this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

## List Changed Notification

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

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

# Message Flow



# Data Types

---

## Prompt

A prompt definition includes:

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

## PromptMessage

Messages in a prompt can contain:

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

All content types in prompt messages support optional [annotations](#) for metadata about audience, priority, and modification times.

## Text Content

Text content represents plain text messages:

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

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

## Image Content

Image content allows including visual information in messages:

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

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

## Audio Content

Audio content allows including audio information in messages:

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

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

## Embedded Resources

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

```
{  
  "type": "resource",  
  "resource": {  
    "uri": "resource://example",  
    "name": "example",  
    "title": "My Example Resource",  
    "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.

# Resources

**Protocol Revision:** 2025-06-18

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

Each resource is uniquely identified by a

[URI](#).

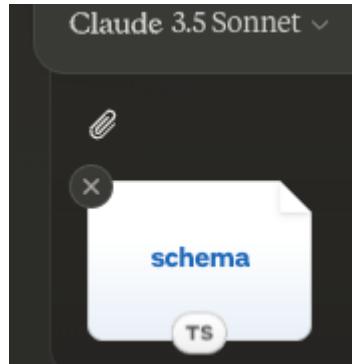
# User Interaction Model

---

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

For example, applications could:

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



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

# Capabilities

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

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

The capability supports two optional features:

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

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

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

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

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

# Protocol Messages

## Listing Resources

To discover available resources, clients send a `resources/list` request. This operation supports [pagination](#).

### Request:

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

### Response:

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

## Reading Resources

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

### Request:

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

## Response:

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

## Resource Templates

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

### Request:

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

### Response:

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

## List Changed Notification

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

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

## Subscriptions

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

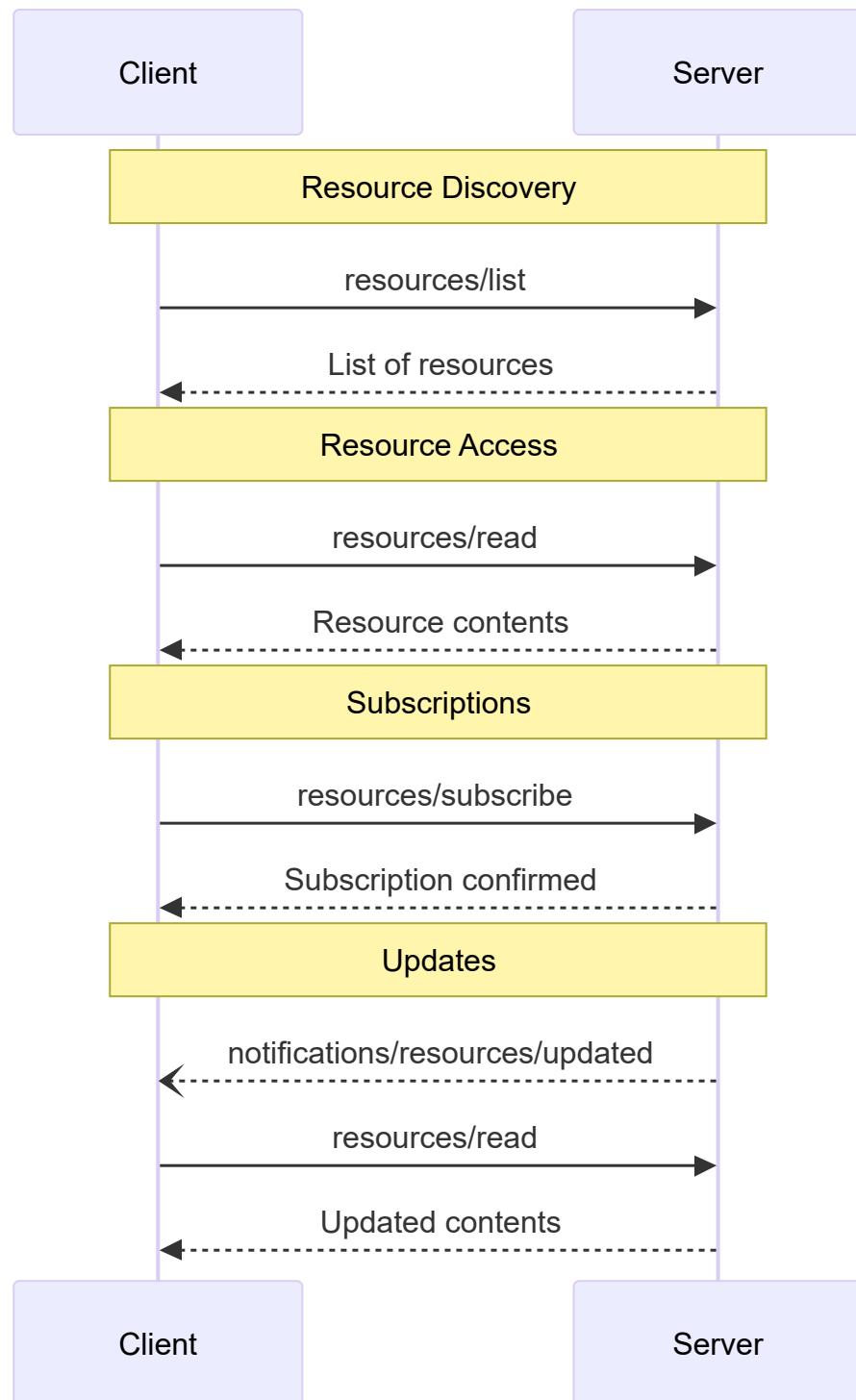
### Subscribe Request:

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

### Update Notification:

```
{  
  "jsonrpc": "2.0",  
  "method": "notifications/resources/updated",  
  "params": {  
    "uri": "file:///project/src/main.rs",  
    "title": "Rust Software Application Main File"  
  }  
}
```

## Message Flow



# Data Types

## Resource

A resource definition includes:

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

## Resource Contents

Resources can contain either text or binary data:

### Text Content

```
{  
  "uri": "file:///example.txt",  
  "name": "example.txt",  
  "title": "Example Text File",  
  "mimeType": "text/plain",  
  "text": "Resource content"  
}
```

### Binary Content

```
{  
  "uri": "file:///example.png",  
  "name": "example.png",  
  "title": "Example Image",  
  "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 into account.

# Error Handling

---

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

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

Example error:

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

# Security Considerations

---

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

**Protocol Revision:** 2025-06-18

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

# User Interaction Model

---

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

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

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

Applications **SHOULD**:

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

# Capabilities

---

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

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

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

# Protocol Messages

## Listing Tools

To discover available tools, clients send a `tools/list` request. This operation supports [pagination](#).

### Request:

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

### Response:

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

## Calling Tools

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

### Request:

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

## Response:

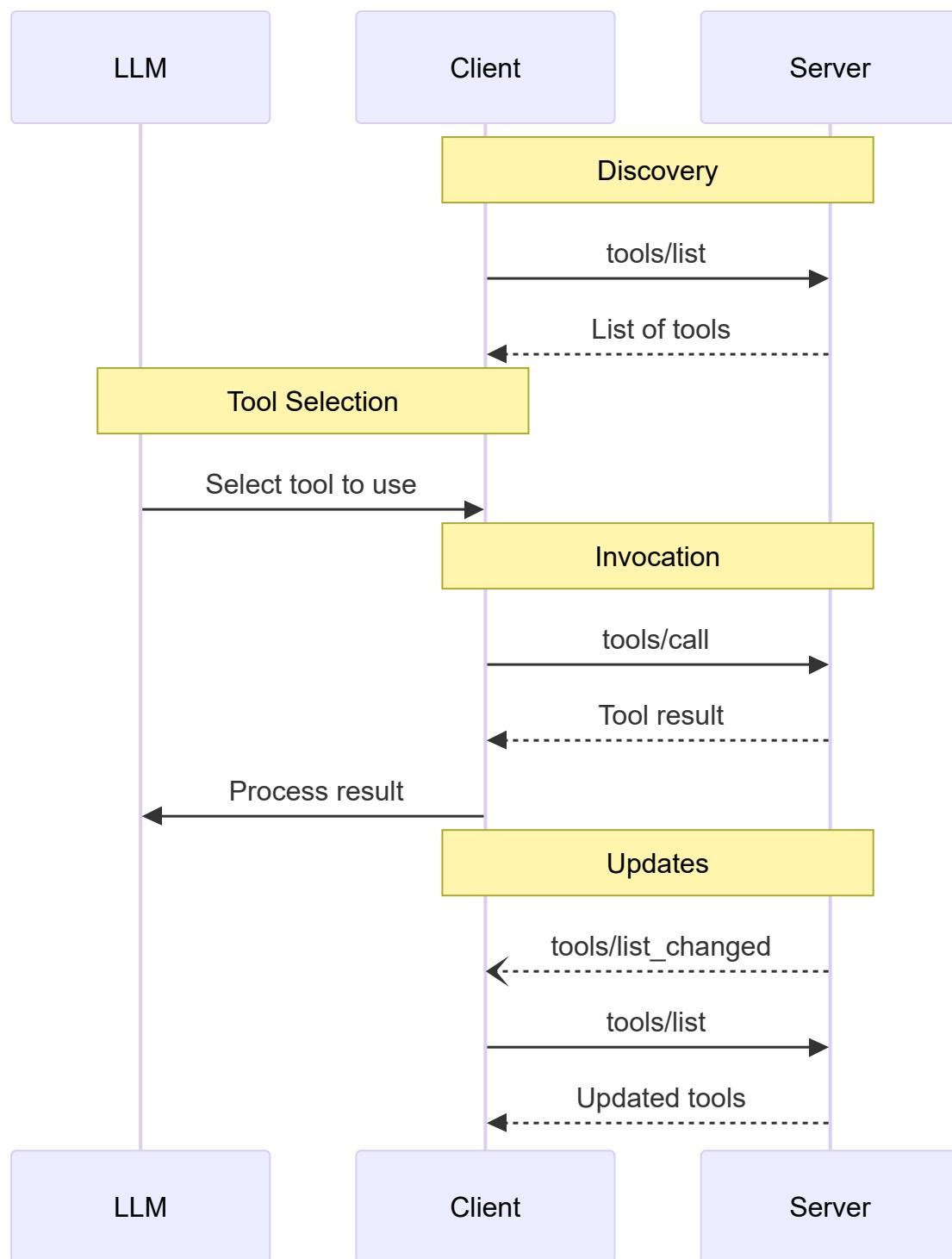
```
{  
  "jsonrpc": "2.0",  
  "id": 2,  
  "result": {  
    "content": [  
      {  
        "type": "text",  
        "text": "Current weather in New York:\nTemperature: 72°F\nConditions: Partly cloudy"  
      }  
    ],  
    "isError": false  
  }  
}
```

## List Changed Notification

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

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

# Message Flow



# Data Types

---

## Tool

A tool definition includes:

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

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

## Tool Result

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

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

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

## Text Content

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

## Image Content

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

This example demonstrates the use of an optional Annotation.

## Audio Content

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

## Resource Links

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

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

Resource links support the same [Resource annotations](#) as regular resources to help clients understand how to use them.

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",  
    "title": "Project Rust Main File",  
    "mimeType": "text/x-rust",  
    "text": "fn main() {\n        println!(\"Hello world!\");\n    }",  
    "annotations": {  
      "audience": ["user", "assistant"],  
      "priority": 0.7,  
      "lastModified": "2025-05-03T14:30:00Z"  
    }  
  }  
}
```

Embedded resources support the same [Resource annotations](#) as regular resources to help clients understand how to use them.

## Structured Content

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

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

## Output Schema

Tools may also provide an output schema for validation of structured results.

If an output schema is provided:

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

Example tool with output schema:

```
{  
  "name": "get_weather_data",  
  "title": "Weather Data Retriever",  
  "description": "Get current weather data for a location",  
  "inputSchema": {  
    "type": "object",  
    "properties": {  
      "location": {  
        "type": "string",  
        "description": "City name or zip code"  
      }  
    },  
    "required": ["location"]  
  }  
}
```

```

},
"outputSchema": {
  "type": "object",
  "properties": {
    "temperature": {
      "type": "number",
      "description": "Temperature in celsius"
    },
    "conditions": {
      "type": "string",
      "description": "Weather conditions description"
    },
    "humidity": {
      "type": "number",
      "description": "Humidity percentage"
    }
  },
  "required": ["temperature", "conditions", "humidity"]
}
}

```

Example valid response for this tool:

```

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

```

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

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

# Error Handling

Tools use two error reporting mechanisms:

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

- o Unknown tools
- o Invalid arguments
- o Server errors

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

- o API failures
- o Invalid input data
- o Business logic errors

Example protocol error:

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

Example tool execution error:

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

# Security Considerations

---

## 1. Servers **MUST**:

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

## 2. Clients **SHOULD**:

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

**Protocol Revision:** 2025-06-18

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

## User Interaction Model

---

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

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

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

# Capabilities

---

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

```
{  
  "capabilities": {  
    "completions": []  
  }  
}
```

# Protocol Messages

## Requesting Completions

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

### Request:

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

### Response:

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

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

### Request:

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

```

        "name": "code_review"
    },
    "argument": {
        "name": "framework",
        "value": "fla"
    },
    "context": {
        "arguments": {
            "language": "python"
        }
    }
}
}

```

## Response:

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

## Reference Types

The protocol supports two types of completion references:

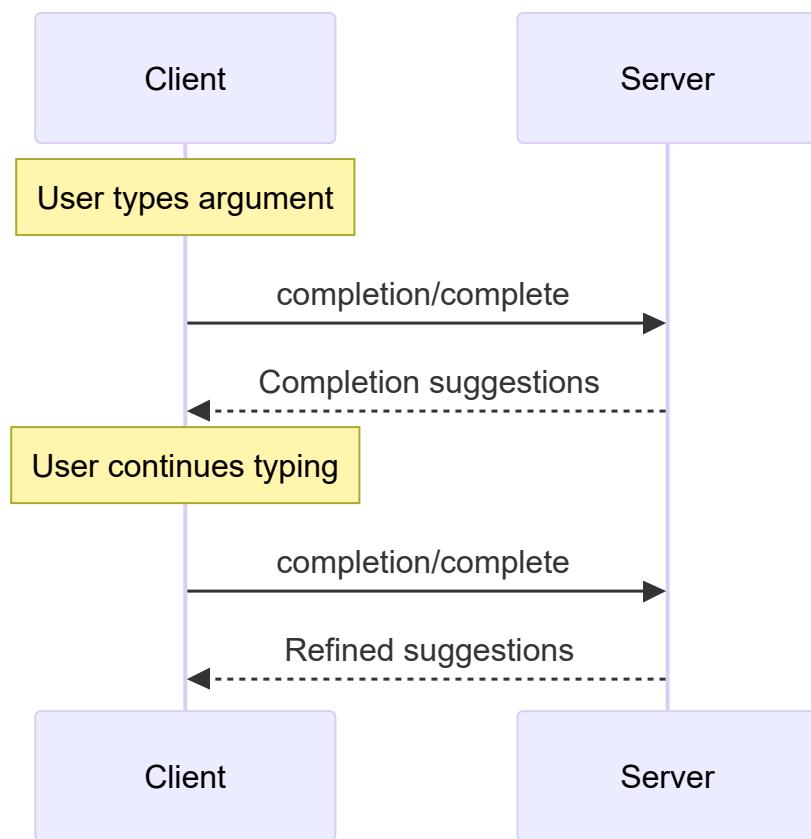
| Type         | Description                 | Example                                         |
|--------------|-----------------------------|-------------------------------------------------|
| ref/prompt   | References a prompt by name | {"type": "ref/prompt", "name": "code_review"}   |
| ref/resource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

## Completion Results

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

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

# Message Flow



# Data Types

---

## CompleteRequest

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

## CompleteResult

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

# Error Handling

---

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

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

# Implementation Considerations

---

## 1. Servers **SHOULD:**

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

## 2. Clients **SHOULD:**

- o Debounce rapid completion requests
- o Cache completion results where appropriate
- o 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# Logging

**Protocol Revision:** 2025-06-18

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

## User Interaction Model

---

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

# Capabilities

---

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

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

# Log Levels

---

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

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

# Protocol Messages

## Setting Log Level

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

**Request:**

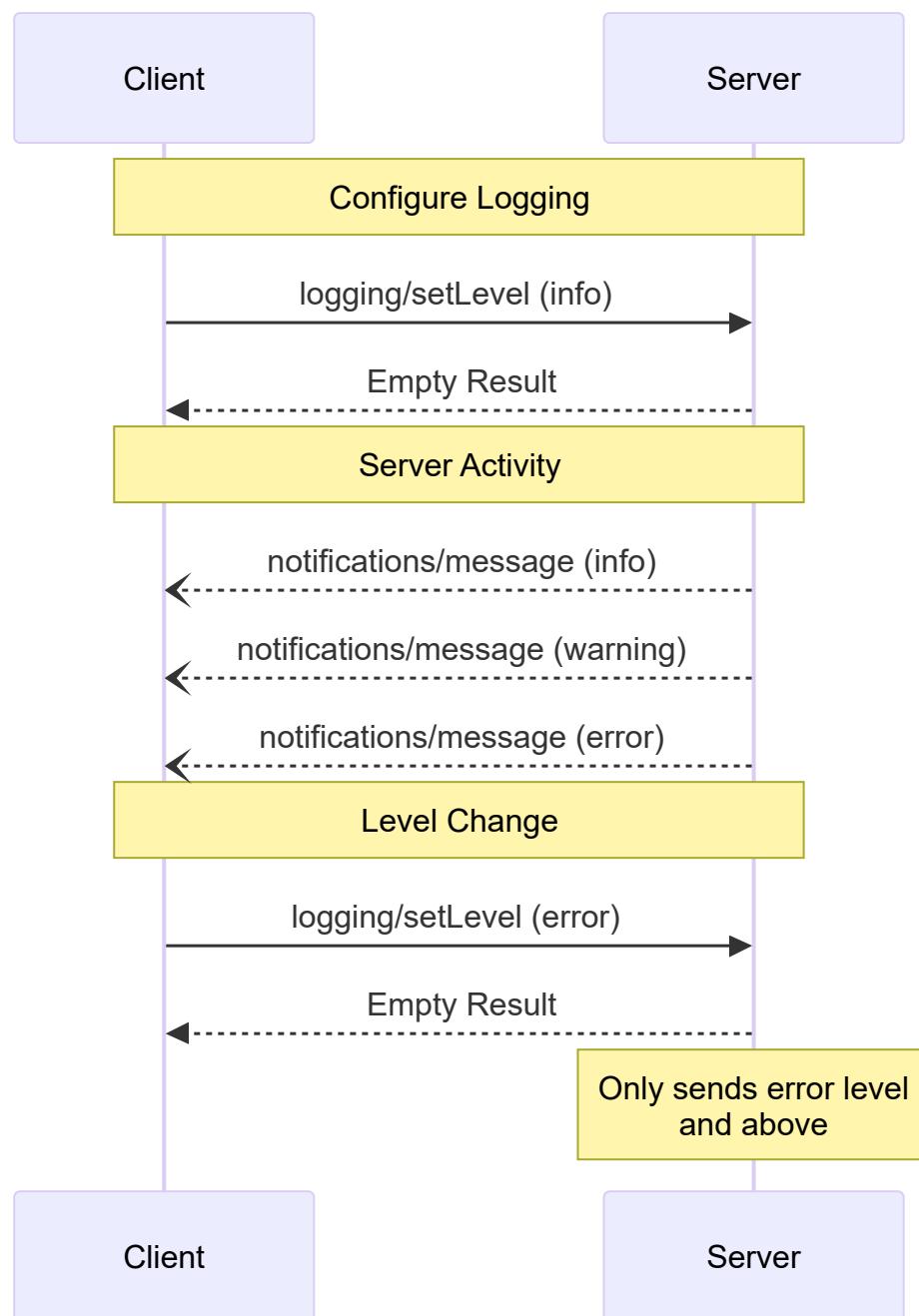
```
{  
  "jsonrpc": "2.0",  
  "id": 1,  
  "method": "logging/setLevel",  
  "params": {  
    "level": "info"  
  }  
}
```

## Log Message Notifications

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

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

# Message Flow



# Error Handling

---

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

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

# Implementation Considerations

---

## 1. Servers **SHOULD**:

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

## 2. Clients **MAY**:

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

# Security

---

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

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

2. Implementations **SHOULD**:

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

**Protocol Revision:** 2025-06-18

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

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

## Pagination Model

---

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

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

# Response Format

---

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

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

```
{  
  "jsonrpc": "2.0",  
  "id": "123",  
  "result": {  
    "resources": [...],  
    "nextCursor": "eyJwYwdlIjogM30="  
  }  
}
```

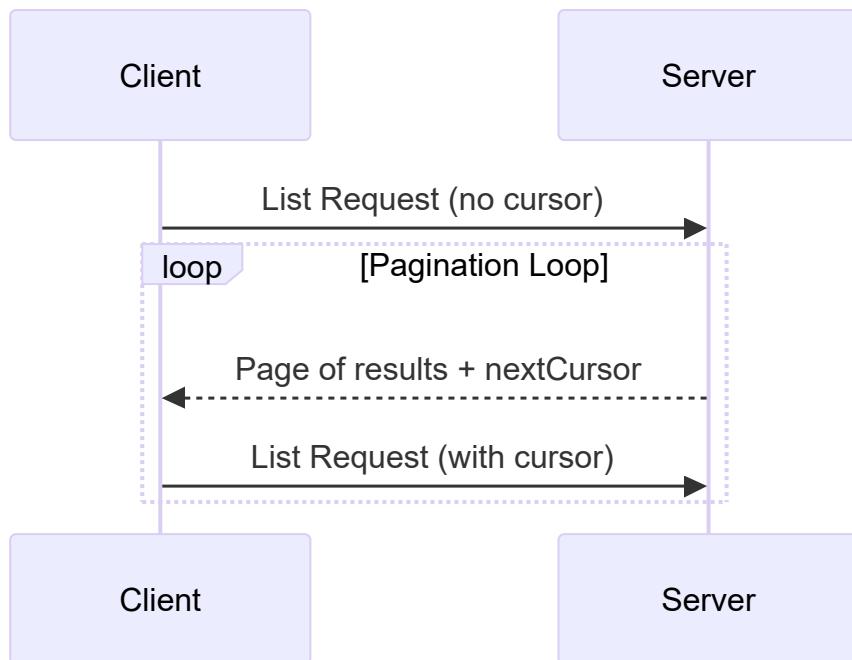
## Request Format

---

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

```
{  
    "jsonrpc": "2.0",  
    "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**:

- o Provide stable cursors
- o Handle invalid cursors gracefully

## 2. Clients **SHOULD**:

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

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

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

# Error Handling

---

Invalid cursors **SHOULD** result in an error with code -32602 (Invalid params).  
[# Schema Reference](#)

# Common Types

---

## Annotations

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

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

`optional` audience

audience?: [Role](#)[]

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

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

**optional** lastModified

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.

`optional` priority

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.

## AudioContent

```
interface AudioContent {\  
  \ meta?: \{ \[key: string]: unknown \};  
  annotations?: Annotations;  
  data: string;  
  mimeTye: string;
```

```
type: "audio";  
}
```

Audio provided to or from an LLM.

`optional``\_meta`

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

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

`optional` annotations

annotations?: [Annotations](#)

Optional annotations for the client.

data

data: string

The base64-encoded audio data.

mimeType

mimeType: string

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

## **BlobResourceContents**

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

The contents of a specific resource or sub-resource.

`optional``\_meta`

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

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

Inherited from [ResourceContents.\\\_meta](#)

blob

blob: string

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

`optional` `mime``Type`

`mime``Type`?: string

The MIME type of this resource, if known.

Inherited from [`ResourceContents.mime``Type`](#)

uri

uri: string

The URI of this resource.

Inherited from [ResourceContents.uri](#)

## BooleanSchema

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

## ClientCapabilities

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

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.

optional elicitation

elicitation?: object

Present if the client supports elicitation from the server.

`optional`experimental

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

Experimental, non-standard capabilities that the client supports.

`optional roots`

`roots?: \{ listChanged?: boolean }`

Present if the client supports listing roots.

Type declaration

- `optional listChanged?: boolean`

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

`optional`sampling

sampling?: object

Present if the client supports sampling from an LLM.

## **ContentBlock**

ContentBlock:

- | [TextContent](#)
- | [ImageContent](#)
- | [AudioContent](#)
- | [ResourceLink](#)
- | [EmbeddedResource](#)

## Cursor

Cursor: string

An opaque token used to represent a cursor for pagination.

## EmbeddedResource

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

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.

`optional``\_meta`

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

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

`optional` annotations

annotations?: [Annotations](#)

Optional annotations for the client.

## **EmptyResult**

EmptyResult: [Result](#)

A response that indicates success but carries no data.

## **EnumSchema**

```
interface EnumSchema {\n  description?: string;\n  enum: string[];\n  enumNames?: string[];
```

```
title?: string;  
type: "string";  
}
```

## ImageContent

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

An image provided to or from an LLM.

**optional** \\_meta

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

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

`optional` annotations

annotations?: [Annotations](#)

Optional annotations for the client.

data

data: string

The base64-encoded image data.

`mimeType`

`mimeType: string`

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

## Implementation

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

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

name

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

`optional` `title`

`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

## JSONRPCError

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

A response to a request that indicates an error occurred.

error

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

Type declaration

- code: number

The error type that occurred.

- `optional` `data?: unknown`

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

- `message: string`

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

## JSONRPCNotification

```
interface JSONRPCNotification {\n  jsonrpc: "2.0";\n  method: string;\n  params?: {\u00d7 _meta?: {\u00d7 [key: string]: unknown }; \u00d7 [key: string]: unknown };\n}
```

A notification which does not expect a response.

`optional` params

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

Type declaration

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

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

Inherited from `Notification.params`

## JSONRPCRequest

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

A request that expects a response.

`optional` params

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

Type declaration

- \[key: string]: unknown
- [optional](#) @\_meta?: \{ progressToken?: [ProgressToken](#); \[key: string]: unknown \}

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

- [Optional](#) progressToken?: [ProgressToken](#)

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

Inherited from Request.params

## JSONRPCResponse

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

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

## LogLevel

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

The severity of a log message.

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

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

## ModelHint

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

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.

optional name

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`

## Model Preferences

```
interface ModelPreferences {\n  costPriority?: number;\n  hints?: ModelHint[];\n  intelligencePriority?: number;\n  speedPriority?: number;\n}
```

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.

`optional` costPriority

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.

`optional hints`

`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.

`optional` intelligencePriority

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.

```
optional speedPriority
```

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.

## NumberSchema

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

## PrimitiveSchemaDefinition

PrimitiveSchemaDefinition:

- | [StringSchema](#)
- | [NumberSchema](#)
- | [BooleanSchema](#)
- | [EnumSchema](#)

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

## ProgressToken

ProgressToken: string | number

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

## Prompt

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

A prompt or prompt template that the server offers.

`optional``\_meta`

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

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

`optional` arguments

arguments?: [PromptArgument](#)[]

A list of arguments to use for templating the prompt.

`optional` description

description?: string

An optional description of what this prompt provides

name

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

`optional` `title`

`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

## PromptArgument

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

Describes an argument that a prompt can accept.

optional description

description?: string

A human-readable description of the argument.

name

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

`optional`|`required`

`required?: boolean`

Whether this argument must be provided.

`optional` `title`

`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

## PromptMessage

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

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.

## PromptReference

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

Identifies a prompt.

name

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

`optional` `title`

`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

## RequestId

RequestId: string | number

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

## Resource

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

A known resource that the server is capable of reading.

`optional``\_meta`

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

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

`optional` annotations

annotations?: [Annotations](#)

Optional annotations for the client.

`optional` description

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.

`optional` `mime_type`

`mime_type?: string`

The MIME type of this resource, if known.

name

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

`optional` size

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.

`optional` `title`

`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

uri: string

The URI of this resource.

## ResourceContents

```
interface ResourceContents {\n  meta?: {[key: string]: unknown };\n  mimeType?: string;\n  uri: string;\n}
```

The contents of a specific resource or sub-resource.

`optional``\_meta`

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

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

`optional` `mime_type`

`mime_type?: string`

The MIME type of this resource, if known.

uri

uri: string

The URI of this resource.

## ResourceLink

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

```
uri: string;
```

```
}
```

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.

```
optional \_meta
```

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

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

Inherited from [Resource.\\\_meta](#)

`optional` annotations

annotations?: [Annotations](#)

Optional annotations for the client.

Inherited from [Resource.annotations](#)

`optional` description

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](#)

`optional` `mime``Type`

`mime``Type`?: string

The MIME type of this resource, if known.

Inherited from [Resource.mime](#)`Type`

name

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](#)

`optional` `size`

`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](#)

`optional` `title`

`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

uri: string

The URI of this resource.

Inherited from [Resource.uri](#)

## ResourceTemplate

```
interface ResourceTemplate {\n  meta?: {[key: string]: unknown };\n  annotations?: Annotations;\n  description?: string;\n  mime_type?: string;\n  name: string;\n  title?: string;
```

```
uriTemplate: string;  
}
```

A template description for resources available on the server.

`optional``\_meta`

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

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

`optional` annotations

annotations?: [Annotations](#)

Optional annotations for the client.

`optional` description

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.

`optional` `mime_type`

`mime_type?: 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.

name

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

`optional` `title`

`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

uriTemplate: string

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

## ResourceTemplateReference

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

A reference to a resource or resource template definition.

uri

uri: string

The URI or URI template of the resource.

## Result

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

```
optional \_meta
```

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

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

## Role

Role: "user" | "assistant"

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

## Root

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

```
uri: string;
```

```
}
```

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

```
optional \_meta
```

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

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

`optional` name

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.

uri

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.

## SamplingMessage

```
interface SamplingMessage \{
  content: TextContent | ImageContent | AudioContent;
  role: Role;
}
```

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

## ServerCapabilities

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

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.

`optional` completions

completions?: object

Present if the server supports argument completion suggestions.

**optional** **experimental**

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

Experimental, non-standard capabilities that the server supports.

`optional` `logging`

`logging?: object`

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

`optional` prompts

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.

`optional`resources

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

Present if the server offers any resources to read.

Type declaration

- `optional`listChanged?: boolean

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

- `optional`subscribe?: boolean

Whether this server supports subscribing to resource updates.

`optional` tools

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.

## StringSchema

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

## TextContent

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

Text provided to or from an LLM.

`optional``\_meta`

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

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

`optional` annotations

annotations?: [Annotations](#)

Optional annotations for the client.

text

text: string

The text content of the message.

## **TextResourceContents**

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

The contents of a specific resource or sub-resource.

`optional``\_meta`

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

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

Inherited from [ResourceContents.\\\_meta](#)

`optional` `mime``Type`

`mime``Type`?: string

The MIME type of this resource, if known.

Inherited from [`ResourceContents.mime``Type`](#)

text

text: string

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

uri

uri: string

The URI of this resource.

Inherited from [ResourceContents.uri](#)

## Tool

```
interface Tool {  
  meta?: {[key: string]: unknown };  
  annotations?: ToolAnnotations;  
  description?: string;  
  inputSchema: {  
    properties?: {[key: string]: object };  
    required?: string[];  
    type: "object";  
  };
```

```
};

name: string;
outputSchema?: {
  properties?: {[key: string]: object };
  required?: string[];
  type: "object";
};

title?: string;
}
```

Definition for a tool the client can call.

`optional``\_meta`

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

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

`optional` annotations

annotations?: [ToolAnnotations](#)

Optional additional tool information.

Display name precedence order is: title, annotations.title, then name.

`optional` description

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

```
inputSchema: \{
  properties?: {[key: string]: object};
  required?: string[];
  type: "object";
}
```

A JSON Schema object defining the expected parameters for the tool.

name

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

`optional` outputSchema

```
outputSchema?: {
  properties?: {[key: string]: object};
  required?: string[];
  type: "object";
}
```

An optional JSON Schema object defining the structure of the tool's output returned in the structuredContent field of a CallToolResult.

`optional` `title`

`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`

## ToolAnnotations

```
interface ToolAnnotations {  
    destructiveHint?: boolean;  
    idempotentHint?: boolean;  
    openWorldHint?: boolean;  
    readOnlyHint?: boolean;  
    title?: string;  
}
```

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.

`optional` `destructiveHint`

`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

`optional``idempotentHint`

`idempotentHint?: boolean`

If true, calling the tool repeatedly with the same arguments  
will have no additional effect on the its environment.

(This property is meaningful only when `readonlyHint == false`)

Default: false

`optional` `openWorldHint`

`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

`optional` `readOnlyHint`

`readOnlyHint?: boolean`

If true, the tool does not modify its environment.

Default: false

`optional` title

title?: string

A human-readable title for the tool.

## completion/complete

---

### CompleteRequest

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

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

params

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

Type declaration

- argument: \{ name: string; value: string \}

The argument's information

- name: string

The name of the argument

- value: string  
The value of the argument to use for completion matching.
- optional context?: { arguments?: {[key: string]: string} }

Additional, optional context for completions

- optional arguments?: {[key: string]: string}

Previously-resolved variables in a URI template or prompt.

- ref: [PromptReference](#) | [ResourceTemplateReference](#)

Overrides Request.params

## CompleteResult

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

The server's response to a completion/complete request

`optional``\_meta`

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

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

Inherited from [Result.\\\_meta](#)

completion

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

Type declaration

- `optional` `hasMore?: boolean`

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

- `optional` `total?: number`

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

- `values: string\[]`

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



## elicitation/create

---

### ElicitRequest

```
interface ElicitRequest {\nmethod: "elicitation/create";\nparams: {\n  message: string;\n  requestedSchema: {\n    properties: {[key: string]: PrimitiveSchemaDefinition};\n    required?: string[];\n    type: "object";\n  };\n};\n}
```

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

params

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

Type declaration

- message: string

The message to present to the user.

- requestedSchema: \  
  properties: \{ \[key: string]: [PrimitiveSchemaDefinition](#) \};  
  required?: string\[];  
  type: "object";  
}

A restricted subset of JSON Schema.

Only top-level properties are allowed, without nesting.

Overrides Request.params

## ElicitResult

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

The client's response to an elicitation request.

`optional``\_meta`

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

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

Inherited from [Result.\\\_meta](#)

action

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

`optional` content

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

The submitted form data, only present when action is "accept".

Contains values matching the requested schema.

# initialize

---

## InitializeRequest

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

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

params

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

Type declaration

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

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

Overrides Request.params

## InitializeResult

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

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

`optional \_meta`

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

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

Inherited from [Result.\ meta](#)

**optional** instructions

instructions?: string

Instructions describing how to use the server and its features.

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

protocolVersion

protocolVersion: string

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

## logging/setLevel

---

### SetLevelRequest

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

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

params

params: \{ level: [LogLevel](#) \}

Type declaration

- level: [LogLevel](#)

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

Overrides Request.params

# notifications/cancelled

---

## CancelledNotification

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

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

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

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

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

params

params: \{ reason?: string; requestId: [RequestId](#) }

Type declaration

- `optional` reason?: string

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

- requestId: [RequestId](#)

The ID of the request to cancel.

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

Overrides Notification.params

## notifications/initialized

---

### InitializedNotification

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

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

optional params

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

Type declaration

- [key: string]: unknown

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

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

Inherited from `Notification.params`

## notifications/message

---

### LoggingMessageNotification

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

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

params

```
params: \{ data: unknown; level: LoggingLevel; logger?: string }  
Type declaration
```

- data: unknown

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

- level: [LoggingLevel](#)

The severity of this log message.

- `optional` logger?: string

An optional name of the logger issuing this message.

Overrides Notification.params

## notifications/progress

---

### ProgressNotification

```
interface ProgressNotification {\nmethod: "notifications/progress";\nparams: {\n  message?: string;\n  progress: number;\n  progressToken: ProgressToken;\n  total?: number;\n};\n}
```

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

params

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

Type declaration

- `optional` `message?: string`

An optional message describing the current progress.

- `progress: number`

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

- `progressToken`: [ProgressToken](#)

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

- `optional total?: number`

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

Overrides `Notification.params`

## notifications/prompts/list\_changed

---

### PromptListChangedNotification

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

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

`optional` params

```
params?: \{ _meta?: \{ [key: string]: unknown \}; [key: string]: unknown \}  
Type declaration
```

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

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

Inherited from `Notification.params`

## notifications/resources/list\_changed

---

### ResourceListChangedNotification

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

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

`optional` params

```
params?: \{ \_meta?: \{ \[key: string]: unknown \}; \[key: string]: unknown \ }  
Type declaration
```

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

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

Inherited from `Notification.params`

## notifications/resources/updated

---

### ResourceUpdatedNotification

```
interface ResourceUpdatedNotification {\nmethod: "notifications/resources/updated";\nparams: \{ uri: string \};\n}
```

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

params

params: \{ uri: string \}

Type declaration

- `uri: string`

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

Overrides `Notification.params`

## notifications/roots/list\_changed

---

### RootsListChangedNotification

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

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

`optional` params

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

Type declaration

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

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

Inherited from `Notification.params`

## notifications/tools/list\_changed

---

### ToolListChangedNotification

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

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

`optional` params

```
params?: \{ \_meta?: \{ \[key: string]: unknown \}; \[key: string]: unknown \ }  
Type declaration
```

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

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

Inherited from `Notification.params`

# ping

---

## PingRequest

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

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

`optional` params

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

Type declaration

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

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

- `optional` progressToken?: [ProgressToken](#)

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

Inherited from Request.params

## `prompts/get`

---

### `GetPromptRequest`

```
interface GetPromptRequest {\nmethod: "prompts/get";\nparams: \{ arguments?: \{ \[key: string]: string \}; name: string \};\n}
```

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

params

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

Type declaration

- `optional arguments?: \{ \[key: string]: string \}`

Arguments to use for templating the prompt.

- name: string  
The name of the prompt or prompt template.

Overrides Request.params

## GetPromptResult

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

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

`optional``\_meta`

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

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

Inherited from [Result.\\\_meta](#)

`optional` description

description?: string

An optional description for the prompt.

## `prompts/list`

---

### `ListPromptsRequest`

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

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

`optional` params

`params?:` { `cursor?:` string }

Type declaration

- `optional cursor?:` string

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

Inherited from PaginatedRequest.params

## ListPromptsResult

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

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

`optional``\_meta`

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

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

Inherited from `PaginatedResult.\_meta`

`optional` `nextCursor`

`nextCursor?: string`

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

Inherited from `PaginatedResult.nextCursor`

## resources/list

---

### ListResourcesRequest

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

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

`optional` params

`params?:` `\{ cursor?: string \}`

Type declaration

- `optional cursor?: string`

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

Inherited from PaginatedRequest.params

## ListResourcesResult

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

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

`optional``\_meta`

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

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

Inherited from `PaginatedResult.\_meta`

`optional` `nextCursor`

`nextCursor?: string`

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

Inherited from `PaginatedResult.nextCursor`

## resources/read

---

### ReadResourceRequest

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

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

params

params: { uri: string }

Type declaration

- uri: string

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

Overrides Request.params

## ReadResourceResult

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

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

`optional`\_meta

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

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

Inherited from [Result.\\\_meta](#)

## resources/subscribe

---

### SubscribeRequest

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

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

params

params: { uri: string }

Type declaration

- `uri: string`

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

Overrides Request.params

## resources/templates/list

---

### ListResourceTemplatesRequest

```
interface ListResourceTemplatesRequest {\nmethod: "resources/templates/list";\nparams?: \{ cursor?: string \};\n}
```

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

optional params

params?: \{ cursor?: string \}

Type declaration

- optional cursor?: string

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

Inherited from PaginatedRequest.params

## ListResourceTemplatesResult

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

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

`optional``\_meta`

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

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

Inherited from `PaginatedResult.\_meta`

`optional` `nextCursor`

`nextCursor?: string`

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

Inherited from `PaginatedResult.nextCursor`

## resources/unsubscribe

---

### UnsubscribeRequest

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

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

params

params: \{ uri: string \}

Type declaration

- `uri: string`

The URI of the resource to unsubscribe from.

Overrides Request.params

## roots/list

---

### ListRootsRequest

```
interface ListRootsRequest {\nmethod: "roots/list";\nparams?: {\n  _meta?: {\n    progressToken?: ProgressToken; \[key: string]: unknown\n  };\n  \[key: string]: unknown;\n};\n}
```

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.

`optional` params

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

Type declaration

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

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

- `optional` progressToken?: [ProgressToken](#)

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

Inherited from Request.params

## ListRootsResult

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

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

This result contains an array of Root objects, each representing a root directory or file that the server can operate on.

`optional`\_meta

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

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

Inherited from [Result.\\\_meta](#)

## sampling/createMessage

---

### CreateMessageRequest

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

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.

params

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

Type declaration

- `optional` includeContext?: "none" | "thisServer" | "allServers"

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

- maxTokens: number

The maximum number of tokens to sample, as requested by the server. The client MAY choose to sample fewer tokens than requested.

- messages: [SamplingMessage](#)[]
- [optional](#) metadata?: object

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

- [optional](#) modelPreferences?: [ModelPreferences](#)

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

- [optional](#) stopSequences?: string[]
- [optional](#) systemPrompt?: string

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

- [optional](#) temperature?: number

Overrides Request.params

## CreateMessageResult

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

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

`optional``\_meta`

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

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

Inherited from [Result.\\\_meta](#)

model

model: string

The name of the model that generated the message.

`optional` stopReason

stopReason?: string

The reason why sampling stopped, if known.

## tools/call

---

### CallToolRequest

```
interface CallToolRequest {  
  method: "tools/call";  
  params: {[key: string]: unknown}; name: string;  
}
```

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

### CallToolResult

```
interface CallToolResult {  
  \_meta?: {[key: string]: unknown};  
  content: ContentBlock[];  
  isError?: boolean;  
  structuredContent?: {[key: string]: unknown};  
  {[key: string]: unknown};  
}
```

The server's response to a tool call.

`optional``\_meta`

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

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

Inherited from [Result.\\\_meta](#)

content

content: [ContentBlock](#)[]

A list of content objects that represent the unstructured result of the tool call.

`optional` `isError`

`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.

`optional` `structuredContent`

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

An optional JSON object that represents the structured result of the tool call.

## tools/list

---

### ListToolsRequest

```
interface ListToolsRequest {\nmethod: "tools/list";\nparams?: \{ cursor?: string \};\n}
```

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

optional params

params?: \{ cursor?: string \}

Type declaration

- optional cursor?: string

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

Inherited from PaginatedRequest.params

## ListToolsResult

```
interface ListToolsResult {\n  \_meta?: {[key: string]: unknown };\n  nextCursor?: string;\n  tools: Tool[];\n  {[key: string]: unknown};\n}
```

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

`optional``\_meta`

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

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

Inherited from `PaginatedResult.\_meta`

`optional` `nextCursor`

`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`

