ChatGPT

Model Context Protocol (MCP) – 2025-06-18 Specification Guide

Overview of MCP

The **Model Context Protocol (MCP)** is an open standard for connecting AI language model applications (clients) with external services (servers) that provide tools, data, or prompts [1]. It defines a **client–server protocol** using JSON-RPC 2.0 for message exchange [2]. By following MCP, AI applications (e.g. IDE extensions, chatbots) can seamlessly interact with external APIs, files, or custom functions in a safe and standardized way [1] [3]. MCP draws inspiration from Microsoft's Language Server Protocol, but focuses on **integrating contextual data and tools** into AI workflows [4].

Key features of MCP include [5] [6]:

- **Resources** – structured data or context (e.g. files, database entries) that the server can share.
- **Prompts** – pre-defined prompt templates or workflows the server can provide for users.
- **Tools** – executable actions or functions the server exposes for the model to invoke.
- **Client-initiated features** – such as **Roots** (to define filesystem/project access boundaries), **Sampling** (server requests the client to perform an LLM generation), and **Elicitation** (server requests additional user input via the client).

These capabilities are negotiated during a connection handshake (capability discovery) so that both client and server agree on what features will be used [7] [8]. The protocol emphasizes **security and user control** – for example, requiring user consent for accessing data or running tools [9] [10]. All messages are encoded in JSON and exchanged over a supported transport channel (stdio, HTTP/SSE, or WebSocket). The sections below describe the transports, lifecycle (handshake and session), message types, and integration details in depth.

Supported Transports: stdio, HTTP, and WebSocket

MCP is **transport-agnostic**, meaning it can work over any bidirectional channel that carries JSON-RPC messages [11]. The official spec defines two standard transports – **stdio** and **Streamable HTTP** – but other transports (like WebSockets) are also used in practice:

STDIO (Standard I/O)

In the stdio transport, the MCP server runs as a subprocess of the client (or host app). The client writes JSON-RPC messages to the server's STDIN, and the server writes responses/notifications to STDOUT [12] [13].

Each message is delimited by a newline (no embedded newlines allowed in JSON) [14]. This transport is simple and efficient for local servers:

- **Launching**: The client spawns the server process and establishes a stdin/stdout pipe [12]. Nothing other than valid JSON-RPC messages should go over these streams (both sides **MUST NOT** send non-MCP output) [15].
- **Logging**: The server may write logs or diagnostics to STDERR if needed; clients can capture or ignore that as it's not part of the protocol data [16].
- **Use case**: Ideal for local integrations (like an IDE extension bundling a server). For example, VS Code can spawn an MCP server and communicate over stdio much like it does for language servers.

Streamable HTTP (HTTP + SSE)

MCP also supports a **streamable HTTP transport** for remote or out-of-process servers [17] [18]. In this mode, the server runs independently (e.g. a web service) and the client connects over HTTP. Communication uses a combination of **HTTP POST** and **Server-Sent Events (SSE)** streams [18]:

- **Endpoint**: The server provides a single HTTP endpoint (URL path) for MCP, e.g. `https://example.com/mcp` [19] [20]. The client will use this for all requests (POSTs) and optionally to open an SSE stream (GET).

- **Client-to-Server Messages (HTTP POST)**: Each JSON-RPC message the client sends (requests, notifications, even replies to server) is sent as a separate HTTP `POST` to the MCP endpoint [21] [22]. The client must include an `Accept` header indicating it can handle JSON and event streams [23]. Depending on the message type:

- If the client POSTs a **notification or response** (no reply expected), the server should respond immediately with **HTTP 202 Accepted** (no body) to acknowledge it [24] [25]. If the input is invalid, the server can respond with an HTTP error (e.g. 400) and optionally include a JSON-RPC error object in the body (with no `id` since it's not replying to a request) [22] [26].
- If the client POSTs a **request** (expects a result), the server has two options:
    1. **SSE streaming response** – respond with `Content-Type: text/event-stream` and keep the connection open to stream results/events [27].
    2. **Single JSON response** – respond with `Content-Type: application/json` and a normal JSON-RPC result in the body [27].

The client **MUST** handle both cases [27]. In practice, simple servers may choose to just return a single JSON result (closing the connection immediately), whereas more advanced servers will switch to SSE to allow sending *multiple messages* (e.g. progress updates or follow-up queries) before the final result.

- **Server-to-Client Streaming (SSE via GET)**: The client may also initiate a long-lived SSE connection by sending an HTTP `GET` to the MCP endpoint (with `Accept: text/event-stream`) [28]. This opens a persistent channel for the server to push **notifications or requests** to the client asynchronously, without waiting for a client request [29] [30]. For example, the server can send a log message or a "resource changed" event at any time on this stream. If the server does not support an SSE push stream, it can respond to the GET with HTTP 405 (Method Not Allowed) [28]. When an SSE stream is active:

- The server may send JSON-RPC **requests** or **notifications** as SSE events (one event per JSON message) [30]. These messages should **not** be responses to earlier client requests (except in a resume scenario); they are either spontaneous notifications or new server-initiated requests [30] [31].

- Either side can close the stream at any time. The server might close it to free resources or if the session ends; the client might close if no longer interested in push events [32]. The client can also open **multiple SSE streams simultaneously** if needed (the server will distribute messages among them, not duplicate) [33].

- **Event Stream Format**: When using SSE, each JSON message is sent as a separate SSE **event**. The server may include an `id` field on each SSE event (not to be confused with JSON-RPC id) to facilitate reconnection [34] [35]. If a stream breaks, the client can send a new GET with `Last-Event-ID` header and the server can replay any missed events from after that ID [36]. This makes the stream *resumable* so that transient disconnects do not cause message loss [37] [38].

- **Session Management**: Because HTTP is stateless, MCP defines a **session mechanism** to tie together a series of requests. When the client first sends the `initialize` request (handshake) via HTTP, the server **may assign a session ID** and return it in an `Mcp-Session-Id` response header [39] [40]. If provided, the client must include this `Mcp-Session-Id` header in **all subsequent HTTP requests** for the session [41] [42]. This ensures the server can associate all calls with the same session state. Important points [43] [44]:

- The session ID should be globally unique and hard to guess (e.g. a UUID or secure token) [40].
- If a request is received without the required session header (and after initialization), the server should reject it (e.g. 400 Bad Request) [45] [46].
- Servers can terminate a session at any time (e.g. on timeout or error). After termination, they will respond with **404 Not Found** to any request using that session ID [47]. The client then knows the session is gone and should start a new handshake (new session) [48] [49].

- Clients can explicitly end a session by sending an HTTP `DELETE` to the endpoint with the session header [50]. This signals the server to clean up session state. The server may reply with 200 OK or 204, or it may send 405 Method Not Allowed if it doesn't allow clients to initiate termination [51]. (Not all servers permit client-driven logout; some might treat session expiration as server-side only.)

- **Protocol Version Header**: After negotiation, the client must include `MCP-Protocol-Version: 2025-06-18` (or whatever version was agreed on) in **every request** header [52] [53]. This helps servers that handle multiple protocol versions. If missing, the server will assume an older default (2025-03-26) for backward compatibility [54]. If an unsupported version is sent, the server returns 400 Bad Request [55].

- **Security Considerations**: When implementing an HTTP MCP server, it is critical to secure it like any web service. The spec **mandates** validating the HTTP `Origin` header to block unknown websites from connecting (prevents DNS rebinding attacks) [56]. Servers should listen only on localhost for local use-cases, or require authentication for remote access [57]. Without these, a malicious webpage could potentially talk to a local MCP server in the background [58].

Example HTTP flow: The client posts an `initialize` request. The server responds with `Content-Type: text/event-stream` and starts an SSE stream. It then streams a few messages (e.g. a `logging/message` and a `tools/list_changed` notification) and finally an `initialize` response result, then closes the stream. The client later POSTs a `tools/call` request, to which the server immediately returns a JSON result (if quick), or again streams via SSE if the operation is long-running and needs progress events. Meanwhile, an SSE push stream (from a GET request) might deliver spontaneous notifications (like incoming data updates).

WebSocket (Custom Transport)

Although not officially specified in MCP 2025-06-18, **WebSocket** is a commonly used custom transport for MCP servers. WebSockets provide a persistent full-duplex connection that naturally fits MCP's bidirectional JSON message exchange. In practice, many implementers have created WebSocket adaptations of MCP [59] [60]:

- **Approach**: The client opens a WebSocket (e.g. to `wss://example.com/mcp/ws`). Once connected, the client and server send JSON-RPC messages as text frames. The framing is simpler than SSE/HTTP: each message is just a text frame containing one JSON object (no need for newline delimiters or separate HTTP requests). This is analogous to stdio but over a network socket.
- **Lifecycle**: The handshake process (initialize, capabilities, etc.) is identical and occurs within the WebSocket channel. Because the connection is stateful, the concept of `Mcp-Session-Id` might not be needed (the session can be tied to the socket). However, a server could still use session tokens for reconnection – for example, the client might send an initial resume message with a prior session ID if supported.
- **Usage**: WebSocket transport can reduce overhead in high-throughput scenarios, since it avoids repeated HTTP setup and can multiplex messages easily. It's suitable for real-time applications where server-to-client messages are frequent (similar to SSE but bi-directional in one channel).
- **Status**: As of 2025, WebSocket support is by convention rather than part of the spec, but many community SDKs provide it [61] [62]. Developers should ensure that their WebSocket transport obeys all MCP message format and ordering rules (e.g. send `initialize` first, etc.) [11]. If using WebSockets, apply standard WebSocket security (TLS, authentication tokens or headers, etc., similar to securing a Web API).

Note: The MCP spec allows **custom transports** generally, as long as the JSON-RPC message format and the connection lifecycle semantics are preserved [11]. This means any environment (Unix sockets, gRPC streams, etc.) could carry MCP, but stdio and HTTP are the baseline for interoperability. WebSockets may become officially recognized in future revisions as an alternative transport given their popularity.

Connection Lifecycle: Handshake, Operation, and Shutdown

MCP defines a clear **lifecycle** for the client-server connection, consisting of an **Initialization phase**, **Operational phase**, and **Shutdown** [63]. This handshake sequence ensures both sides agree on protocol version and enabled features before doing any work. Below we detail each phase:

Initialization Phase (Handshake)

Initialization is the first interaction in any MCP session and handles **protocol version and capability negotiation** [64] [65]. It begins with the client sending an `initialize` request and ends when both sides are ready for normal operation.

- **Client sends `initialize`:** As soon as the transport is connected (process started, HTTP connected, etc.), the client must send an `"initialize"` JSON-RPC **request** [66]. This message includes:
- `protocolVersion` – The MCP protocol version the client supports (typically the latest version string). For example: `"2025-06-18"` (or a prior version if the client is older) [67].
- `capabilities` – An object declaring which optional features the client can handle [7]. For instance, a client might send `"capabilities": {"roots": {"listChanged": true}, "sampling": {}, "elicitation": {}}` to indicate it supports workspace roots (and will notify changes), and also supports sampling and elicitation [68] [69]. If the client has any experimental or extension features, those can be indicated under an `"experimental"` sub-field.
- `clientInfo` – Identifying info about the client (name, version, possibly UI name) [70] [71]. This is for logging or display; e.g. the client might send its application name and version.

Example: A minimal initialize request might look like:

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

This indicates the client supports roots (and will send notifications if roots change), sampling, and elicitation [72] [7]. The client is requesting version **2025-06-18** of the protocol.

- **Server responds to `initialize`:** The server must reply with either a successful result (if it can speak some version in common) or an error (if it cannot proceed). On success, the JSON-RPC **result** includes:

- `protocolVersion` – The version the server will use. If the server supports the client's requested version, it echoes that; otherwise it may respond with a different version string that it does support [73]. The client should check this. If the server proposes a version the client can't handle, the client should gracefully disconnect (no common protocol) [74].
- `capabilities` – The server's capabilities object, listing features it provides [75] [8]. For example, a server might include `"prompts"`, `"resources"`, `"tools"`, `"logging"`, etc., potentially with sub-features. E.g. `"tools": { "listChanged": true }` means it supports tools and will send notifications if the tool list changes [8]. `"resources": { "subscribe": true, "listChanged": true }` means it supports resource subscriptions and will notify on resource list changes [76]. Any capabilities not listed are considered not supported by the server.
- `serverInfo` – Identifying info about the server (name and version, similar to clientInfo) [77] [78].
- `instructions` – (Optional) A human-readable message or tips from the server to the client [78]. For instance, a server might send a note about required authentication or a welcome message. Clients may display this in a log or UI to assist users.

Example: A server's initialize response might be:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "protocolVersion": "2025-06-18",
    "capabilities": {
      "logging": {},
      "prompts": { "listChanged": true },
      "resources": { "subscribe": true, "listChanged": true },
      "tools": { "listChanged": true }
    },
    "serverInfo": {
      "name": "ExampleServer",
      "version": "1.0.0"
    },
    "instructions": "Welcome! This server requires login for API calls."
  }
}
```

Here the server agreed on **2025-06-18** and advertised that it supports logging, prompts (with dynamic list updates), resources (with subscriptions and updates), and tools (with dynamic list updates) [79] [80]. The client, seeing this, now knows it can use those features. The server also provided a friendly instruction string.

- **Client sends** `initialized`**:** After receiving the server's response, the client concludes the handshake by sending a **notification** `"notifications/initialized"` (with no params) [81] [82]. This tells the server that the client is ready to proceed. At this point, initialization is complete.

During initialization, *no other messages should be in-flight*: The client **SHOULD NOT** send any requests besides possibly a ping until the server responds to `initialize` [83]. Likewise, the server **SHOULD NOT** send any requests (other than trivial pings or logging messages) until it receives the `initialized` notification from client [84]. This ordering prevents race conditions by ensuring both sides finalize negotiation before doing work.

Version negotiation details: If the client and server don't immediately agree on a version, the server may respond to `initialize` with a different `protocolVersion` that it supports [73]. For example, if a client built for 2025-06-18 connects to an older server that only supports 2025-03-26, the server might respond with `"protocolVersion": "2025-03-26"`. The client can decide if it supports that older version; if yes, it continues using 2025-03-26 for this session, or if not, it should terminate the session [74]. Typically, the client will offer its highest version, and the server will downgrade if needed (or vice versa if the server is newer and client older). A mismatch usually results in an **error response**. For instance, a server might reply with an error code if it truly cannot communicate:

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

This example shows the server didn't understand the requested version and lists what it does support [85]. (Here the client oddly asked for "1.0.0", and server says it only supports "2024-11-05".) The error code `-32602` (Invalid params) is used in this case, with a data object conveying the supported vs requested versions [86].

Capability Negotiation and Discovery

Part of initialization is exchanging **capabilities**, which dictates which features will be active. Each side only sends requests/notifications related to features that both sides have declared. Here's how capabilities work:

- **Server capabilities:** Common server-provided features include:
- `prompts` – the server can provide prompt templates for the client/user [68].
- `resources` – the server can serve resource data (files, etc.) [87].
- **`tools`** – the server exposes tools/functions the model can call [87].
- `logging` – the server can emit logging messages to the client [87].
- `completions` – the server supports *argument autocompletion* via a separate API (used to help fill in parameters for prompts or resource templates, for example) [88].

- `experimental` – a placeholder for any non-standard features (both sides can advertise experimental support flags) [89] [88].

- **Client capabilities:** Common client-side optional features include:

- `roots` – the client can provide *workspace roots* (filesystem locations) and notify the server if they change [90].
- `sampling` – the client supports server-initiated LLM invocations (the server can request the client's AI to generate a completion) [90].
- `elicitation` – the client supports server requests for user input (the server can ask the client to prompt the user for more info) [90].
- `experimental` – again, for any experimental client features or flags.

If a capability is not listed, it's assumed not supported. The intersection of client and server capabilities determines what's usable in the session. For instance, if a server offers `tools` but the client did not declare `tools` capability, the client would not attempt to use tool-related requests (and if it did, the server might respond with Method Not Found). Conversely, if a client supports `sampling` but server didn't declare it, the server won't send any `sampling` requests.

Many capabilities have **sub-capabilities (flags)** that refine their behavior [91]:

- `listChanged`: If `true`, it means the party will send notifications when the *list of items* changes. The server can set this for prompts, resources, or tools to indicate it can notify the client of additions/removals [92] [93]. The client can set `listChanged` for roots to indicate it will notify the server if workspace roots change.
- `subscribe`: (Server side, for resources) If `true`, the server allows the client to subscribe to individual resource updates [91] [76]. We'll see how subscriptions work in the Resources section.
- Other flags specific to certain features may be introduced over time.

The **result of negotiation** is that both sides know which features are active. For example, if both advertise `tools`, then tool discovery and invocation can occur. If one side lacks it, those methods should not be used. This negotiation prevents unsupported method calls.

After initialization succeeds, the server will typically perform **capability-specific discovery** to inform the client of available content: - If `prompts` are supported, the client might call `prompts/list` to get the list of prompt templates. - If `tools` are supported, the client (or the server) may initiate a `tools/list` to get the available tools. - If `resources` are supported, the client might call `resources/list` to enumerate accessible resources. - If `roots` are supported (client capability), the server can call `roots/list` to see what directories the client has exposed.

This initial discovery phase lets each side populate UIs or internal structures. Alternatively, the server may proactively send some info (though generally the client drives the discovery by requesting lists).

Operational Phase (Normal Use)

Once initialized, the connection enters the **operation phase**, where the client and server exchange requests and notifications to utilize the negotiated features [94]. During this phase: - Both parties **must adhere to**

the agreed protocol version [95]. (No switching versions mid-session; if either side upgrades their software, they'd need to reconnect.) - They must use only the capabilities negotiated. For example, if the server didn't advertise `tools`, the client should not send `tools/list` or `tools/call` (and if it does, the server will likely respond with an error or ignore it) [95].

Message exchange in operational phase is via JSON-RPC: - The **client** will typically send requests to invoke server features (like calling a tool, reading a resource, etc.). - The **server** can also send requests to the client if the client supports those features (e.g. asking the client to create a sample LLM message via `sampling/createMessage`, or to provide user input via `elicitation/create`, or simply requesting the list of roots). - Both can send notifications: the server might send `.../list_changed` events or log messages; the client might send `roots/list_changed` or cancellation notifications, etc.

The protocol is *asynchronous*: multiple requests can be in flight concurrently (identified by unique IDs). The server and client should be able to handle out-of-order responses. If using HTTP, concurrency might be limited by how many requests can be outstanding depending on implementation (with SSE, server can handle multiple easily; with sequential POST, the client might need multiple connections to parallelize).

Some typical interactions in this phase: - **Tool invocation**: The model (via the client) decides to use a tool. The client sends `tools/call` to server. Server executes the tool and returns result (possibly streaming partial output). - **Resource access**: The user picks a resource to include in context. The client sends `resources/read`. Server returns the content (or an error if not available). - **Prompt usage**: The user selects a prompt template. The client sends `prompts/get`. Server returns the filled prompt messages, which the client can then insert into the chat or use as needed. - **Server notification**: The server detects an update (e.g. a file changed on disk). If the client subscribed or if it promised `listChanged`, the server sends `notifications/resources/updated` or `.../list_changed` to inform the client. - **Client notification**: The user adds a new workspace folder. If the client supports roots, it sends `notifications/roots/list_changed` to let the server know the root set changed (so server can call `roots/list` again to update its scope).

All these are elaborated in the sections below. The main idea is that after initialization, both sides operate according to the contract established by capabilities.

Session Continuity and Reconnection

MCP sessions can persist as long as the transport connection is alive (and the server has not explicitly ended the session). For long-running sessions or intermittent networks, a few mechanisms exist: - The **Mcp-Session-Id** (for HTTP transports) allows a client to reconnect to a new HTTP connection and continue the same logical session [39] [96]. If a client disconnects (or the server restarts but still recognizes session IDs), using the same `Mcp-Session-Id` on a new initialize attempt can resume context. However, typically if the TCP connection is lost but server stays up, the client would just re-`GET` the SSE stream with Last-Event-ID to resume events [36], and continue using the session ID for new POSTs. - For **WebSockets or stdio**, if the connection drops, the session is effectively lost unless the protocol defines a resume. The current spec doesn't define an official resume for those; the client would usually start a fresh session. Some servers might implement custom session tokens to allow quick reattach (not standardized in 2025 spec). - **Heartbeat/Ping**: The client can send periodic `ping` requests to check if the server is responsive (and keep the connection alive, preventing timeouts). The spec includes a `ping` method for this purpose [97]. The

server should respond promptly (likely with a simple result like `{"jsonrpc":"2.0","id":X,"result":{}}` or an echo) to indicate liveness. - **Timeouts**: The spec recommends clients and servers enforce timeouts for requests to avoid hanging forever [98] [99]. If a response isn't received in a reasonable time, the requester should send a cancellation (and possibly alert the user) [98]. Implementations might reset the timeout if progress notifications are coming in (meaning work is ongoing) [100], but there should be an ultimate cutoff to avoid infinite waits [101].

Graceful Shutdown

MCP doesn't have a specific "shutdown" message in the protocol (unlike LSP which has a shutdown request). Instead, termination is handled by the underlying transport being closed gracefully [102]. Typically, the **client initiates shutdown** when the user is done or the application is closing:

- **Stdio shutdown**: The client can close the stdin pipe to signal EOF to the server process [103]. The server should then exit on its side. If it doesn't exit timely, the client can send a SIGTERM to gently terminate [104], and if that fails, a SIGKILL as last resort [105]. Servers may also choose to exit if they detect their stdout is closed (client gone).
- **HTTP shutdown**: Simply closing or not making further HTTP requests will eventually end the session. For long-lived SSE streams, closing the HTTP connection (or issuing a DELETE with the session ID as mentioned) signals shutdown [106]. The server should release any session state when it observes the connection closed or the session ended.
- **WebSocket shutdown**: Closing the WebSocket connection (normal closure handshake) suffices. The server should clean up session on socket close. If the server wants to shut down, it can also close the socket from its side.
- **In-protocol notice**: While no explicit `shutdown` RPC exists, either side could send a final log message or custom notification indicating it will close, but that's not standardized. It's generally sufficient to just close the transport.

After shutdown, the client and server should free resources. If a new session is needed later, they start again with a fresh `initialize`.

Server-Provided Capabilities and Messages

Once connected, servers can provide three main categories of functionality to the client: **Prompts**, **Resources**, and **Tools** [107]. Each of these corresponds to a capability that must have been negotiated. We discuss each in detail below, including how the client discovers and uses them, and how the server can send updates.

Prompt Provisioning (Pre-defined Templates)

Prompts are reusable, structured messages or instructions that a server offers to help users interact with the model [108] [109]. They can be thought of as "canned" chat inputs or workflows. For example, a server might have a prompt for "Code Review" that, when invoked, provides a formatted request to the model to review some code.

- **Capability**: The server must declare the `prompts` capability in its initialize response to indicate it offers prompts [110] [111]. The presence of `prompts` tells the client that it can query for a list of

prompts and fetch prompt content. A sub-flag `listChanged` indicates the server will notify the client if the set of available prompts changes at runtime [112] [111].

- **User Interaction**: Prompts are typically **user-triggered** on the client side [113]. For instance, in a chat UI the user might type a slash command (`/`), and the client can show a list of prompts provided by the server as suggestions. The user selects "/code_review" and the client then retrieves that prompt and inserts it into the chat. The MCP protocol itself is agnostic to *how* the user selects a prompt (could be a menu, command palette, etc.) [109], but best practice is to integrate them naturally. *(For example, VS Code surfaces prompts as slash commands in the chat input [114] [115].)*

- **Listing Prompts**: The client can request the list of available prompts by sending a `prompts/list` request. This returns a list of prompt **definitions**. The request may include a `cursor` for pagination if the list is long (the spec supports paged results, though many servers will return all in one go) [116] [117].

Example:

Request: `{"jsonrpc":"2.0", "id":42, "method":"prompts/list", "params":{}}`
Response:

```
{
  "jsonrpc": "2.0",
  "id": 42,
  "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": null
  }
}
```

In this example, the server listed one prompt with name `"code_review"`, a human-friendly title and description, and an argument called `"code"` that the user must provide [118] [119]. The `arguments` field

describes any inputs the prompt needs – here it requires a snippet of code. `nextCursor` would be used if there were more prompts beyond this page [120].

- **Prompt Definition Format**: Each prompt in the list has:
- `name`: an identifier (unique key to request that prompt) [121].
- `title`: a short name for UI display (optional) [121] [122].
- `description`: longer explanation (optional, for hover text or details) [123] [124].
- `arguments`: a list of expected arguments (each with its name, description, and whether required) [125] [126]. Arguments allow prompts to be parameterized. For example, a "Translate Text" prompt might take a target language argument.

- **Getting a Prompt**: To use a prompt, the client sends `prompts/get` with the prompt `name` and an `arguments` map providing the required values [127] [128]. The server responds with the actual prompt content, typically one or more chat messages that form the template.

Example:

Request:

```
{
  "jsonrpc": "2.0",
  "id": 43,
  "method": "prompts/get",
  "params": {
    "name": "code_review",
    "arguments": {
      "code": "def hello():\n    print('world')"
    }
  }
}
```

Response:

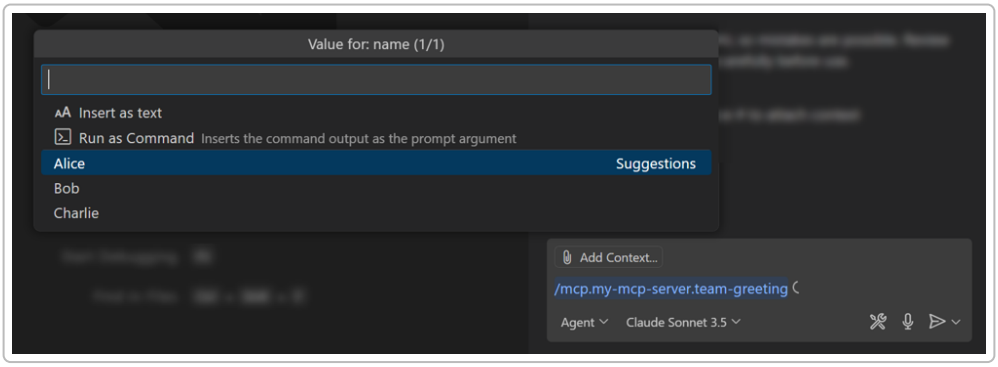
```
{
  "jsonrpc": "2.0",
  "id": 43,
  "result": {
    "description": "Code review prompt",
    "messages": [
      {
        "role": "user",
        "content": {
          "type": "text",
          "text": "Please review this Python code:\n def hello():\n
```

```
print('world')"
        }
      }
    ]
  }
}
```

Here, the server returned a description and a list of message(s) that make up the prompt [129] [130]. In this case it's a single user-role message instructing the assistant to review the provided code. The client would typically insert this message into the chat conversation on behalf of the user (or directly send it to the model as the next turn). The prompt content can include different content types – in this example, just plain text. (If the prompt had images or other media, the content could have `"type": "image"` etc., but most prompts are text-based instructions.)

> **Note:** The spec mentions that arguments might be *auto-completable via the completion API* [131]. If the server supports the `completions` capability, the client can request suggestions for argument values (for example, suggesting file names or user names). In VS Code, if a prompt's arguments have completion hints, it will show a dialog for input with suggestions [132] [133] (as in an image below).

- **Using Prompts in UI**: A well-integrated client will show available prompts in a user-friendly way. For instance, prompts can be listed when the user types "/" in a chat input. The client might display the `title` and `description` so the user knows what each prompt does. When selected, the client obtains the prompt content (via `prompts/get`) and can either directly send it to the model or allow the user to confirm/edit it. If arguments are required, the client should prompt the user to fill them in. In some UIs, a form or quick pick is shown for each argument. **Example (VS Code)**: The user triggers *"Use MCP Prompt"*, selects **Team Greeting** prompt, and VS Code pops up a small input asking for the `name` parameter, with suggestions "Alice/Bob/Charlie" provided by the server [134] [133].



Example: VS Code collecting an input argument for an MCP prompt. Here the prompt expects a `name` value; the extension shows a dialog with suggestions (Alice, Bob, Charlie) provided via the server's completion hints. [115] [133]

- **Prompt list updates**: If the server's available prompts can change (e.g., new prompts become available or some are removed based on context), and if it advertised `prompts.listChanged: true`, it should send a **notification** `"notifications/prompts/list_changed"` to the client [135] [136]. This tells the client that the prompt list it previously retrieved is now stale. The client can

then refresh by calling `prompts/list` again. The `list_changed` notification has no parameters (just a signal) [137] [138]. Servers may send this at any time. For example, a server might offer different prompts depending on the active project or user role, and notify the client when those change.

- **Error handling**: If the client requests a prompt name that doesn't exist, the server should return a JSON-RPC error (e.g., code `-32001` or `-32602` with message "Prompt not found"). If arguments are missing or invalid, code `-32602` (Invalid params) is appropriate, possibly with details in `data`. These errors let the client inform the user that the prompt couldn't be retrieved. Usually, prompt retrieval is straightforward so errors are rare outside of mismatches.

In summary, the **Prompts** feature gives users quick access to server-supplied expertise or multi-step instructions. It helps standardize "canned prompts" so they are discoverable and parameterizable, rather than burying them in documentation. From a developer perspective, implementing `prompts/list` and `prompts/get` allows you to guide user interactions in powerful ways without hardcoding logic in the client.

Resources (Contextual Data Sharing)

Resources in MCP are pieces of data or content that a server makes available to the client and potentially to the language model [139]. Think of resources as the extended context or knowledge base – files, documents, database entries, or any structured information that can inform the model's responses. Each resource is identified by a **URI** (Uniform Resource Identifier) [140], which often conveys the type or location of the resource (e.g. a file path).

- **Capability**: The server must declare the `resources` capability to use this feature [141]. Subfields:
- `subscribe`: if true, the server supports subscriptions to resource changes [142] [143].
- `listChanged`: if true, the server will send notifications if the overall list of resources changes (e.g., new or removed resources) [142] [144].

- Both can be true, false, or either one true as needed (or the server can support neither extra feature) [145] [146]. The client will adapt based on what's declared.

- **User Interaction**: Resources are generally **application-driven** and user-controlled in how they are presented [147]. For example, an IDE might show a list or tree of resources (files, etc.) that the user can choose to attach to the AI chat or open in an editor. Or the AI model might itself request a resource (like "open file X") if it's aware of them, but typically the user explicitly shares resources. The protocol does not mandate any UI, but common patterns include:

- A "Resources" panel or command where the user can browse available resources (like a file browser).
- The ability for the user to search or filter resources by name.
- Automatic inclusion: in some advanced cases, the client might automatically attach high-priority resources based on context (though with user knowledge ideally).

Example: VS Code has an **"MCP Resources" Quick Pick** where it lists resources and resource templates provided by the server [148] [149]. The user can pick one to open or attach as context. Resources can also be added to a chat via an "Add Context" button.

- **Listing Resources**: The client sends `resources/list` to get the list of available resources. The server returns a list of resource descriptors. This may be a subset if pagination is used (with `cursor`), similar to prompts/tools listing [150] [151].

Example:

Request: `{"jsonrpc":"2.0","id":10,"method":"resources/list","params":{}}`

Response:

```
{
  "jsonrpc": "2.0",
  "id": 10,
  "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": null
  }
}
```

Here the server exposed one resource: a file URI `file:///project/src/main.rs`, with a human name and title describing it [152] [153]. The MIME type is provided (`text/x-rust`) which hints it's a Rust source file [153]. The `description` could be a short blurb about the resource. The `name` is a short identifier (in this case just the filename). The combination of `name` and `title` helps UI: e.g., the client might display "main.rs – Rust Software Application Main File".

If there were more resources beyond the first page, `nextCursor` would be non-null and the client could call `resources/list` again with that cursor to get the next set [150] [154].

- **Resource Descriptor Fields**:
- `uri`: A unique identifier (and possibly location) for the resource [140] [155]. URIs use schemes like `file://`, `https://`, etc., or custom schemes (e.g. `repo://` for a repository content).
- `name`: A short name for display, perhaps a filename or key [155] [156].
- `title`: Optional longer name for display [155] [156].

- `description`: Optional description of the resource [155] [156].
- `mimeType`: Optional MIME type indicating content type (e.g. `text/plain`, `image/png`) [153] [155].
- `size`: (Optional) size in bytes, if known – not shown in example but part of schema [155].
- `annotations`: (Optional) metadata like audience, priority, etc. (discussed below) [157] [158].

- **Reading a Resource**: To get the actual content of a resource, the client sends `resources/read` with the `uri` of the desired resource [159] [160]. The server responds with the content. The response format is a bit nested: it typically contains a `contents` array with one or more content blocks. In most cases, a single resource URI corresponds to a single content item (for example, reading a file returns that file's content). The array allows for composite resources or batched reads.

Example:

Request: `{"jsonrpc":"2.0","id":11,"method":"resources/read","params":{"uri":"file:///project/src/main.rs"}}`

Response:

```
{
  "jsonrpc": "2.0",
  "id": 11,
  "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}"
      }
    ]
  }
}
```

The content block echoes some metadata (uri, name, title, mimeType) and then provides either a `text` field or `blob` field with the data [161] [162]. Here, since it's a text file, the server returned a `text` string containing the source code [161] [162]. For a binary resource, it would include a `blob` field with base64-encoded data instead [163]. The client, upon receiving this, might display the content (e.g., open the text in an editor tab or feed it into the AI's context if the user attached it to a question).

If the resource is large, the server might in some cases split it into multiple chunks in the `contents` array, but typically one item is used. Some servers might also include related resources (for example, if reading a directory, the server might return multiple entries as separate content objects). The protocol is flexible here.

- **Resource Templates**: Servers can also expose *parameterized resources* using URI templates. This is useful for things like dynamic queries. For example, a server could define a template for a GitHub repository content: `repo://{owner}/{repo}/contents/{path}`. The client can then ask the user to fill in `{owner}`, `{repo}`, etc. The MCP method `resources/templates/list` returns a list of such templates [164] [165].

Example:

Response to `resources/templates/list`:

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

This shows a template with a placeholder `{path}` in the URI [166]. Essentially it indicates the server can serve arbitrary files if the client supplies the path. The client could use such templates to prompt the user (e.g., "Enter a file path") and then construct the URI and call `resources/read`. Another example (from VS Code's GitHub MCP extension) might be templates for "Repository Content for specific branch" or "for specific commit", etc., where placeholders are branch names or commit hashes【39†】. Resource templates often work in conjunction with **completions** (if the server supports it) to auto-suggest valid values for those placeholders (like suggesting branch names or file paths).

- **Resource Annotations**: Similar to prompt messages and tool outputs, resources can carry **annotations** to guide how they should be used [157] [167]:
- `audience`: `["user"]`, `["assistant"]`, or `["user","assistant"]` indicates who the content is primarily meant for [168] [169]. For instance, a large documentation file might be marked `["assistant"]` meaning it's mainly for the AI to read, not to show directly to the user. Or a UI screenshot image might be `["user"]` meaning it's for the user to view.

- `priority`: A number 0.0 to 1.0 indicating importance [169] [170]. `1.0` means this resource is highly relevant (almost required context), `0.0` means purely optional. Clients can use this to auto-select or highlight resources; e.g. always include priority 1.0 items in the prompt to the model.
- `lastModified`: A timestamp of when the resource was last changed [171]. Useful to display recency or to ignore stale data. Format is ISO 8601 (e.g. `"2025-01-12T15:00:58Z"`) [171].

Example of a 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"
  }
}
```

This indicates the README is moderately important (0.8), intended for the user (perhaps the user might want to read it), and was last edited on Jan 12, 2025 [172] [173]. A client UI might show the date or sort resources by last modified, and maybe automatically suggest including high-priority ones when the AI is asked a question (with user permission).

- **Using Resources in Chat**: Typically, if a user wants the AI to use some data, the client will fetch that resource and then provide its content to the model (for example, by prepending it to the prompt or using some insertion mechanism). Some clients allow attaching resources directly to a chat turn (e.g., "Add Context" button in VS Code to attach an MCP resource, which the extension then includes in the prompt). The model can then see that content. If a resource is large, the client might only include a summary or skip it based on priority to avoid token limits.

- **Live Updates via Subscription**: If the server set `subscribe: true`, the client can subscribe to a resource to get notified when it changes [174] [175]. This is useful for real-time updates. For instance, a server providing a log file could allow subscription so that as new log entries appear, it notifies the client, which can then read the new content or update an open editor.

To subscribe, the client sends `resources/subscribe` with the resource's URI [174] [176]. The server will respond (likely just with an acknowledgment result, possibly empty). From then on, when that resource (or something within it) updates, the server sends `notifications/resources/updated` notifications. The `updated` notification includes the `uri` of the resource that changed [177] [178].

Example:

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

This tells the client that `main.rs` has new content. The client can then call `resources/read` again to get the latest content, or if it has it open in an editor, refresh it. The spec notes that the updated URI could be a "sub-resource" of what was subscribed – e.g., if you subscribed to a directory, an update may indicate which file changed within it [179] [180].

If a client no longer wants updates, it can send `resources/unsubscribe` with the URI [181] [182]. (Unsubscribe was listed in the schema, meaning servers should implement it for completeness.)

- **Resource List Changes**: Independently of individual content updates, the server can notify if the *catalog* of resources changes via `notifications/resources/list_changed` (if `listChanged: true`) [183] [184]. This is similar to the prompts case. For example, if a new resource becomes available (say a new file was added to the project and now the server exposes it), the server should send `resources/list_changed`. The client would then refresh its resource list (call `resources/list` again) to get the new entry [185] [186].

Use case: In an IDE, if the user opens a new project file, the extension might add it to the context list on the server side and notify the client to update the UI.

- **Common URI Schemes**: The spec outlines a few standard URI schemes and their intended use [187] [188]:
- `https://` – for web resources. Should be used only if the client can fetch that URL directly itself (i.e., it's publicly accessible). Otherwise, if the server itself needs to fetch/provide the data, a custom scheme is better [189].
- `file://` – for filesystem-like resources (files, directories). It doesn't necessarily have to map to a physical filesystem (server could generate content on the fly when file:// URI is read) [190]. The spec suggests using appropriate MIME types (like `inode/directory` for directories) [190].
- `git://` – a scheme for Git repository content [191]. The details aren't heavily specified, but presumably a server could use URIs like `git://repo/path` to identify content in a git repo.

- **Custom** – Servers can define their own (e.g., `repo://` or `db://`). These must conform to URI syntax rules [192] and ideally be documented so the client knows what they refer to. Clients treat them opaquely (just use the URI with the server).

- **Error Handling**: If a resource is not found or unavailable, servers should return a **JSON-RPC error**. The spec suggests `-32002` for "Resource not found" [193] [194], including the URI in the error data. For example:

```
{
  "jsonrpc": "2.0",
```

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

This lets the client inform the user that the requested file doesn't exist [194]. Other errors: if the server had an internal failure reading the file, `-32603` (Internal error) could be returned [193]. If the client calls a resource method but the server didn't advertise `resources` capability, it will likely respond with `-32601 Method not found` [195] [196] (as recommended in the Roots error handling, similar principle).

In summary, Resources provide a structured way to share data context with the model. The client can browse or query them, present them to the user, and fetch content on demand. The annotation system helps the client prioritize what to include in the prompt to the model. A key part of MCP's value is this ability to attach relevant data (code, docs, records) to model queries in a controlled manner, rather than dumping everything blindly.

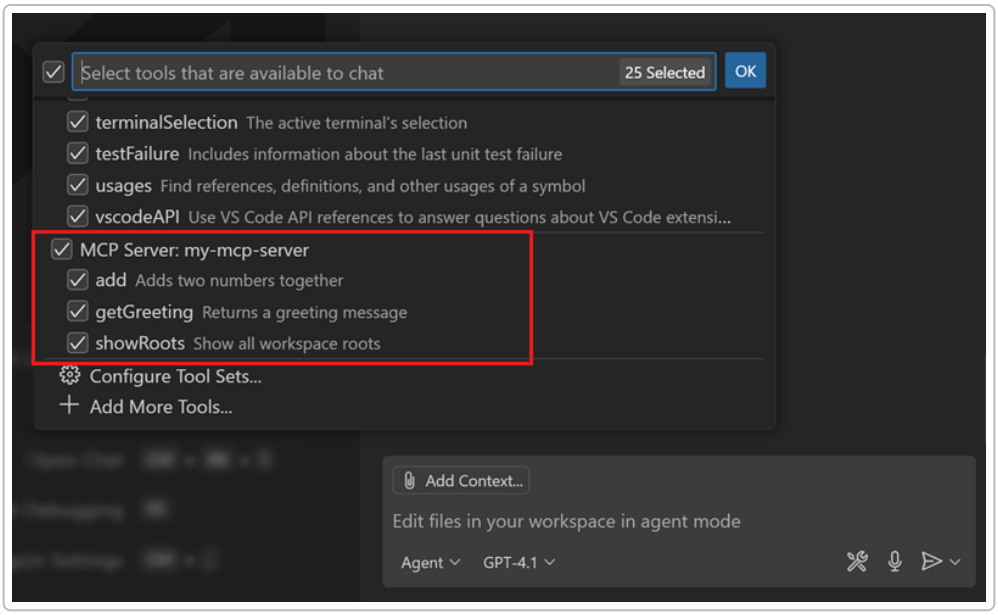
Tools (Functions and Actions)

Tools are perhaps the most powerful feature: they let the server expose arbitrary operations that the AI model can invoke via the client [197] [198]. This enables an AI to do things like run code, query an API, modify files, etc., through well-defined functions – bringing *"agent"* capabilities to the model in a safe, supervised way.

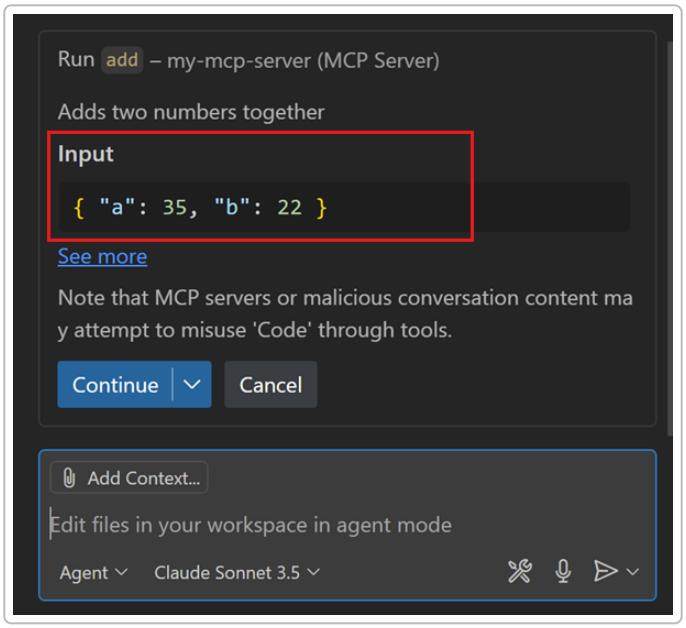
- **Capability**: The server must declare `tools` capability to provide tools [199]. An optional `listChanged` flag indicates it will notify if tools are added/removed at runtime [200].

- **User Interaction & Control**: Tools are conceptually **model-controlled** – the AI decides when to use a tool based on the conversation [198]. However, due to the potential risk (tools can execute code or change data), **there must always be a human in the loop granting final approval** [201]. This is emphasized strongly: applications **SHOULD** present a confirmation to the user before actually executing a tool call [202]. The client's UI should make it clear what tools are available to the AI and when one is used. For example:

- VS Code shows an AI **"tools picker"** where users can enable/disable specific tools for the AI to use [203]. It also shows a confirmation dialog each time a tool is about to run (displaying the tool name and the input the AI provided) [204] [205].



Example: VS Code's tool selection interface (agent tools picker). "MCP Server: my-mcp-server" tools like add, getGreeting, showRoots are listed (with descriptions). The user can check which tools are available to the AI [203].



Example: VS Code's tool execution confirmation dialog. Here the AI chose to run the `add` *tool (from "my-mcp-server"), which "*Adds two numbers together*". The dialog displays the input parameters* `{ "a": 35, "b": 22 }` *that the AI provided* [204]*, so the user can review them. The user must click Continue to actually execute the tool, or Cancel to refuse.* [205]

The spec suggests clients should always confirm tool usage unless a tool is explicitly marked as safe for auto-execution [202] [206]. Some tools might be read-only (not causing side-effects); for those, clients might skip confirmation if indicated (see **tool annotations** below).

- **Listing Tools**: The client (or server) can list available tools via `tools/list`. Typically, right after initialization, the client will do this to know what functions exist. The response gives a list of tool definitions.

Example:

Response to `tools/list`:

```
{
  "jsonrpc": "2.0",
  "id": 51,
  "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"]
        }
        // outputSchema could be here if provided
      }
    ],
    "nextCursor": null
  }
}
```

This shows one tool **get_weather** that provides weather info [207] [208]. The tool definition fields: - `name`: identifier for the tool (used to invoke it) [207]. - `title`: short human-friendly name (optional) [209]. - `description`: description of what it does (for UI and for the model to understand) [210]. - `inputSchema`: a JSON Schema object describing the tool's expected input parameters [211]. In this case, it requires a string `"location"`. The schema can include `properties` with types, descriptions, and a `required` list [212]. This schema is crucial: it tells the client (and effectively the model, via description) what inputs to provide. - `outputSchema`: (optional) JSON Schema for the tool's output if the server returns structured data [213] [214].

Not present in the above snippet, but we'll discuss it later. - `annotations`: optional extra metadata about the tool's behavior or usage [213] [215].

The client can use this list to display tools in a UI (with title/description) and to inform the model about the available tools. Many AI systems will inject a list of functions (tools) into the model's context so the model knows what it can call. For instance, the client might pass to the model something like: *"You have access to the following tools: get_weather(location: string) – Get current weather information for a location."* The MCP spec doesn't dictate how the model is informed, but it's up to the client integration. (In VS Code, the tools are integrated into the "agent mode" where the model picks them, likely similar to OpenAI function calling in principle.)

- **Tool Invocation (`tools/call`)**: When the model (through the client) decides to use a tool, the client sends a `tools/call` request to the server. The parameters include the `name` of the tool and an `arguments` object that conforms to the tool's input schema [216] [217].

Example:

Request:

```
{
  "jsonrpc": "2.0",
  "id": 52,
  "method": "tools/call",
  "params": {
    "name": "get_weather",
    "arguments": { "location": "New York" }
  }
}
```

This asks the server to execute the `get_weather` tool with `"location": "New York"`.

The server will perform whatever action is needed (maybe call a weather API) and then return a result. The result includes: - Either a **success** result with output data, - Or an indication of an error (either via JSON-RPC error or in-band error as `isError` true – explained below).

Successful Response Example:

```
{
  "jsonrpc": "2.0",
  "id": 52,
  "result": {
    "content": [
      {
        "type": "text",
        "text": "Current weather in New York:\nTemperature: 72°F\nConditions:
```

```
Partly cloudy"
      }
    ],
    "isError": false
  }
}
```

Here the server returned a human-readable text output with the weather info [218]. It put that in the `content` array as a Text content block, and marked `isError: false` indicating the tool ran successfully [218].

However, tool results can be more complex. The `content` array can contain multiple pieces of content (text, images, etc.), or even structured data (via a separate field): - **Content Blocks**: Each content element has a `type` (e.g., `"text"`, `"image"`, `"audio"`, `"resource_link"`, or `"resource"`) [219] [220]. We saw text example above. If the tool returned an image, it would be:

```
{
  "type": "image",
  "data": "<base64 image data>",
  "mimeType": "image/png",
  "annotations": { ... }
}
```

as part of the `content` array [221] [222]. The client could then display that image to the user (or provide it to the model if the model can handle images, though current mainstream LLMs are text-based – an image might be more for user benefit). - The example in spec shows an image with an annotation marking it for user consumption, priority, etc. [223] [224]. - Audio content similarly would have `data` (base64) and `mimeType` in an object with `"type": "audio"` [225]. - **Resource Links**: A tool might not return raw data, but a reference to a resource that was produced or identified. For example, a tool that searches a database might return a URI to a resource containing detailed results. In that case, the content block might be:

```
{
  "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 }
}
```

This tells the client "I (the tool) have something at this URI." The client can then decide to auto-fetch it or present it to the user as an available resource. The annotations might indicate that this link is mainly for the assistant (LLM) to use [226] [227]. The spec notes that resource links returned by tools might not show up in a normal `resources/list` – they could be ephemeral or ad-hoc [228]. The client should treat them as valid

resources for reading or subscribing nonetheless. - **Embedded Resource Content**: In some cases, a tool could return a content block of type `"resource"` with an actual resource object embedded (URI plus data) [229] [230]. This is like the tool directly giving the data as if it were a resource read. The example in spec shows embedding a file's content in the result [231]. The server should have `resources` capability if it uses this (since it's essentially the same format) [229] [232]. - **Structured Content**: If the tool has a defined `outputSchema`, the server can return a machine-readable result in addition to (or instead of) textual content. The spec provides a `structuredContent` field for this purpose [233]. For backward compatibility, it's recommended to also include the equivalent as text in the `content` array [233] [234] (so older clients or the model's context can still see something). - For example, if `get_weather` had an output schema specifying temperature, conditions, humidity as fields, the server might return:

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

And indeed the spec example shows exactly this for a weather tool [235] [236]. The text version is a JSON string, and `structuredContent` is the parsed object. - The client can use `structuredContent` to, for instance, present a nicely formatted table or trigger some UI action (since it has actual data types). Or a programming-oriented client could feed it directly into a function. - If the client (or the model) only understands text, the textual JSON in content ensures nothing is lost.

- **Tool Output Schema**: Declaring an `outputSchema` in the tool definition is optional but recommended if the tool returns structured results [237] [238]. If provided:
- The **server** must ensure its `structuredContent` adheres to that schema [239] [240].
- The **client** should validate the `structuredContent` against the schema when it receives it [239] [241]. This way, the client can detect if the server or tool produced an unexpected format.
- Schemas help with **self-documentation** and robust integration – for instance, a UI could automatically generate forms or result displays from the schema definitions.

Providing an output schema can significantly help an AI or a client know what to expect, especially in typed scenarios (numbers vs strings, etc.) [242] [243].

- **Dynamic Tools and Updates**: If `listChanged: true`, the server can add or remove tools at runtime and notify via `notifications/tools/list_changed` [244] [245]. For example, the server might detect the project is a Python project and register a `run_tests` tool dynamically, then send `tools/list_changed`. The client, upon receiving that, calls `tools/list` again to get the new set. VS Code's agent mode supports such dynamic discovery [246] – e.g., enabling tools based on context. The `list_changed` notification has no params (just a signal) [247] [248].

- **Tool Annotations**: Tools can have an `annotations` field in their definition to convey extra metadata about how/when to use them [213]. The spec leaves this open-ended as an object. One known annotation (used by VS Code) is:

- `readOnlyHint`: a boolean indicating the tool does not alter state (read-only). If true, VS Code does **not** ask for user confirmation to run that tool [249] [250], because it's considered safe (for example, a `get_weather` tool might be read-only). This improves UX by not nagging the user for harmless queries.
- Other possible annotations could be imagined (like `dangerous` or `costly` flags), but as of this spec the main one highlighted is readOnly. The `title` was also mentioned as an annotation in VS Code docs [251], but that might be a documentation quirk since title is also a top-level field. It's likely just clarifying the title is used in UI.

The spec explicitly warns that clients should treat tool annotations as **untrusted** unless from a trusted server [252]. This is because a malicious server could mislabel a dangerous tool as `readOnly`. So clients should ideally have their own allow-lists or user settings rather than blindly trusting the annotation. In practice, a well-known server can be trusted, but caution is advised.

- **Executing Tools Safely**: On the server side, implementing a tool means writing code that will run triggered by external input (the model's requests). Therefore, **servers must validate all tool inputs** and guard against misuse [253]. The spec's security guidelines for tools include:
- Checking that input parameters meet expected formats and ranges (despite the client presumably validating against schema, double-check on server).
- Enforcing access controls – e.g., if a tool can read a file, ensure the path is within allowed roots and not something sensitive.
- Rate limiting calls if needed (to avoid abuse or infinite loops) [253].
- Sanitizing outputs (so that returning content doesn't inject malicious sequences into the model's context inadvertently).
- Possibly sandboxing code execution if the tool runs code.

On the client side: - Always confirm with the user before running a tool (unless it's in some safe auto-mode as configured by user) [202]. - Show the exact inputs the model provided, so the user can spot if something is off (like the model trying to pass `rm -rf /` to a shell tool, the user can catch it). - Allow user to cancel or modify the input before running if desired [204] [205] (some UIs let you edit the arguments in the confirm dialog). - After execution, possibly show the output to the user or attach it into the chat for the model (ensuring it doesn't directly execute code in user environment without showing results).

- **Error Handling for Tools**: There are two levels of errors:
- **Protocol-level errors**: If the tool name is invalid or input is wrong format, the server can respond with a JSON-RPC error (so no `result`). For example, unknown tool:

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

(from spec example) [254]. Or if required param missing: `-32602 Invalid params` with details. If the tools feature isn't supported at all, server might return `-32601 Method not found` to any `tools/...` call. These errors mean the call didn't even execute the tool logic.

- **Tool execution errors**: The tool ran but encountered a runtime issue (API returned error, the operation failed, etc.). In this case, the server should return a **normal result** with `isError: true` and perhaps an error message in the content [255] [256]. This keeps the JSON-RPC request successful (so the client isn't confused thinking the protocol failed), but marks to the client and model that the outcome was an error.
  - E.g., a weather API tool might return `isError:true` with content `"Failed to fetch weather data: API rate limit exceeded"` [256]. The client can present that message to the user or model.
  - The model might then respond accordingly ("I couldn't get the weather because the API limit was hit.").

Using `isError` vs. JSON-RPC error is a design choice: `isError` is for *business-logic errors* that are expected as part of normal operation, whereas JSON-RPC errors are for *structural or invocation errors*. This distinction allows the conversation with the AI to continue gracefully even if a tool fails – the failure becomes part of the assistant's reasoning rather than a broken protocol interaction.

The client should handle both. If it gets a JSON-RPC error from `tools/call`, it likely means something was fundamentally wrong (the client might log it or inform the user that the tool couldn't run at all). If it gets a result with `isError:true`, it can treat it as the tool's output (often the client will just feed that back to the model or show it to the user).

- **Security Considerations Recap**: The spec explicitly enumerates recommendations [257] [253]:
- Server side: validate inputs, enforce access (e.g., a file write tool should not allow writing outside allowed directories), maybe implement permission checks (some servers require an API key or user login to do certain tools, tying in with the Authorization spec).
- Server side: possibly log all tool invocations and results for audit.
- Client side: confirm with user, show inputs, apply timeouts to tool calls to prevent hanging [258], and treat any output with caution (especially if it could contain code or instructions).
- Both sides: assume the model might try malicious things (since the model's choice to call a tool could be influenced by an adversarial prompt). So design tools and their usage defensively. For instance, if there's a `execute_code` tool, have the client *always* show the code to the user for approval, rather than just running it.

In essence, the Tools feature transforms the AI from a static Q&A system into an interactive agent that can perform tasks – but it relies on a tight partnership between server (which implements the tasks) and client (which mediates those tasks with the user's oversight). MCP provides the schema and message framework to do this in a consistent way across different applications.

Additional Protocol Features and Notifications

Beyond the core "feature" methods above, MCP defines various **utility messages** that support a robust and user-friendly experience. These include **progress updates**, **cancellation**, **logging**, **pings**, and specialized notifications.

Progress Updates (Streaming Feedback)

For long-running operations (like a tool that takes several seconds or more), MCP allows the server to send **progress notifications** to keep the client (and user) informed [98] [100]. This is done via the `notifications/progress` message.

- A progress notification carries:
- An optional `message`: a text description of current status (e.g. "Indexing file 3 of 10...") [259] [260].
- A `progress` number: how much work has been done so far (a count or percentage) [261] [262].
- An optional `total`: the total work units, if known [263] [264].
- A `progressToken`: a token to identify which request this progress is about [265] [266].

Importantly, the progress mechanism is **opt-in and initiated by the client**. When the client issues a request that it wants progress on (say a `tools/call` that might be slow), it includes a `progressToken` in that request's `_meta` data [267] [268]. For example:

```
{
  "id": 55,
  "method": "tools/call",
  "params": { ... },
  "_meta": { "progressToken": 123 }
}
```

The actual value of the token can be any string or number chosen by the client to correlate messages [269] [270]. If the server sees this and supports progress, it may start sending `notifications/progress` with that same token.

Each `notifications/progress` the server sends will look like:

```
{
  "jsonrpc": "2.0",
  "method": "notifications/progress",
  "params": {
    "progressToken": 123,
    "progress": 2,
    "total": 5,
    "message": "Processed 2 out of 5 files"
  }
}
```

This indicates 2/5 files done, with a message [271] [272]. The client matches `progressToken: 123` to the original request ID 55. It can then display a progress bar or status to the user. If multiple requests with progress are running, the tokens differentiate them.

The spec notes: - The `progress` value should be monotonically increasing (never go backwards) [262]. - If `total` is unknown, the server can omit it or send a best-guess, and just increment `progress` arbitrarily (e.g., 0.4, 0.6 if 100% unknown). If known, providing both gives a clear percentage. - The optional `message` is for human consumption (so it might be shown in a status line or console) [273]. - Example: A tool "train_model" could send progress 0..100 with total 100 as percentage, or it could send progress=some bytes processed out of total bytes.

Progress notifications are sent as JSON-RPC **notifications** (one-way messages) from server to client. They can be delivered over SSE if HTTP transport, or over the persistent channel if stdio/WS.

Effect on timeouts: As mentioned earlier, receiving a progress notification is a hint to the client that the request is still active. The client might reset a request's timeout timer upon each progress update [100], so that as long as progress is coming, it doesn't prematurely cancel. However, a maximum cap should still apply in case progress notifications themselves get stuck or something goes wrong [101].

Not all servers or operations will implement progress. But for heavy tasks, it greatly improves UX – the user sees that something is happening rather than the application hanging silently.

Cancellation Mechanism

MCP defines a way to cancel in-flight requests cooperatively. Either side can send a `notifications/cancelled` message to indicate it will no longer wait for or process a certain request [274] [275].

- The cancellation notification contains:
- `requestId`: the ID of the request to cancel [276] [277].
- `reason`: optional human-readable reason for cancellation [278] [279].

For example, if a user hits a "Stop" button to halt a long tool call with ID 52, the client could send:

```
{
  "jsonrpc": "2.0",
  "method": "notifications/cancelled",
  "params": { "requestId": 52, "reason": "User aborted" }
}
```

The server, upon receiving this, should attempt to stop work on that request. Since JSON-RPC has no built-in cancel concept, this out-of-band notification is the signal.

Key points: - **Timing**: The cancellation is advisory. Due to latency, the server might have already finished or even sent the response by the time it gets the cancel. Or the server might receive it and then shortly after finish the task. The spec notes cancellation *should* happen while the request is in-flight, but it might arrive too late in which case it's essentially a no-op (the result might still come, but the client will likely ignore it) [275] [280]. - **Effect**: When a side receives a cancel for a request it is handling, it should stop processing and not send a response (or if already sent, fine). If not yet sent, it can consider the result "unused" so it can abandon heavy computation [280] [281]. If partial SSE events were being streamed, the server might close the stream early upon cancel. - **No Cancel for Initialize**: The spec forbids the client from cancelling an

`initialize` request [281] (that handshake is too fundamental; if it's stuck, better to drop connection than send cancel). - **Either direction**: The server could also cancel a request it made to the client. For example, if the server requested `sampling/createMessage` and then no longer needs it (maybe user input arrived or some other change), it could send cancel. Or if a long elicitation request is no longer needed (user took too long or gave up), server might cancel it. These scenarios are less common but possible. - **Client behavior on cancel**: If the client cancels a request it sent, it will ignore any response that might still come (since it told the server to stop, but maybe the server was about to reply). If the server cancels a request it sent (to client), the client should not bother to send a reply for it.

This mechanism aligns with how LSP and others handle cancellation ($/cancelRequest in LSP). It is purely cooperative – no guarantee – but typically effective for long tasks.

Logging Notifications

MCP servers can emit **log messages** to share diagnostic or debug info with the client (and ultimately, possibly the user or developer). The server declares `logging` capability if it supports this [8]. Logging works via: - A notification `notifications/message` with params: - `data`: the log content (could be a string message or any JSON-serializable object) [282] [283]. - `level`: the severity level (e.g. "INFO", "WARN", "ERROR", etc.) [284] [285]. The spec likely defines an enum for LoggingLevel (commonly levels might be "trace", "debug", "info", "warn", "error", "fatal"). - `logger`: optional name of the logger/source [284] [286] (could indicate which component or module of server is producing the message).

Example:

```
{
  "jsonrpc": "2.0",
  "method": "notifications/message",
  "params": {
    "data": "Fetching API data...",
    "level": "INFO",
    "logger": "WeatherTool"
  }
}
```

This might be sent by the server to inform the client that it's doing something. The client could display it in a console or ignore it if not needed.

- **Controlling log level**: The client can send a `logging/setLevel` request to tell the server what minimum level to log at [287] [288]. For instance, `{"method":"logging/setLevel","params":{"level":"WARN"}}` to only get warnings and errors. If the client never sets a level, the server decides what to send (maybe errors only, or everything if it's a debug build) [289] [290].
- **Use cases**: Logging is mainly for developers or advanced users to see what's happening under the hood (especially when building custom MCP servers). It can also be useful for auditing actions (the server might log tool calls, etc., though those might also be conveyed through normal responses).

- **Note**: Log messages are separate from the standard JSON-RPC errors. They're more verbose runtime info rather than error responses. They also can be sent anytime (even during initialization, the spec allowed logging notifications early) [84] .

The client may surface logs in a dedicated output panel. In VS Code, likely these logs could appear in an Output channel for the MCP server or in the developer console if enabled.

Heartbeat Ping

MCP defines a simple `ping` request that either side can use to test connectivity. The schema shows `ping` (probably returning an `EmptyResult` or similar) [291] .

- The client might send periodic `ping` requests if no other traffic has occurred, just to keep the connection alive or measure latency.
- The server can respond with a trivial success (maybe an empty object). The spec doesn't detail ping's response in the text we have, but presumably it's just:

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

  or possibly it might echo some timestamp.
- Pings are allowed during initialization wait (the client or server can send a ping to ensure the other side is still there) [292] .

It's mostly a utility and not always needed if there is frequent traffic or if the transport has its own keepalive.

Session Management and Authentication Notifications

We covered session management via headers earlier. There isn't a specific notification for session end or anything; it's handled by HTTP codes or disconnections.

However, related is **Authentication** under the Authorization extension: - If a server is protected (requires OAuth tokens), and a client tries to use it without a token or with an expired token, the server will respond with HTTP 401 and a `WWW-Authenticate` header indicating where to get the token [293] [294] . - The client then goes through an OAuth flow (outside MCP messages) to obtain a token, then retries requests with `Authorization` header containing the token (or maybe includes it in the MCP session handshake if specified). - The spec's Authorization section is quite detailed, effectively aligning MCP with OAuth 2.1 flows (with the server as Resource Server and an external Auth Server) [295] [296] . It is beyond the core MCP, but key points: - The client can discover the auth server from the MCP server (via metadata and headers) [297] [294] . - Clients can register dynamically if needed (public vs confidential clients). - Once an access token is obtained, the client includes it with requests to authorized endpoints (likely via `Authorization: Bearer <token>` HTTP header for HTTP transport). - If using stdio, auth is out-of-band (maybe environment variables or a prompt to user to login and then pass token via some config message – the spec says STDIO should *not* use the HTTP auth spec, but could use environment credentials) [298] . - If a token is revoked or expired, the server returns 401 and the process repeats.

Visual Studio Code's MCP integration explicitly supports OAuth with GitHub/Microsoft accounts and provides UI for users to grant and manage trust for MCP servers [299] [300] . In VS Code, there's a "Manage

Trusted MCP Servers" action where a user can see which MCP servers have access to their account tokens [301] [300]. This is an example of how client integration should give users control over which servers can act on their behalf.

"Roots" (Workspace Roots) – Client-to-Server Feature

We should mention **Roots** briefly, as it often ties in with Tools/Resources. The Roots feature (client capability) allows the client to inform the server about the boundaries of the user's workspace or project directories [302] [303]. If the client supports it: - The client declares `roots` capability (with possibly `listChanged`) [304] [305]. - The server can then send `roots/list` request to get a list of root URIs that the client provides (e.g. the path of the open folder in an IDE) [306] [307]. - The client responds with an array of Root objects (each with a file URI and an optional name/label) [308] [309]. - If at any time the user adds/removes a workspace folder or the set of roots changes, the client sends `notifications/roots/list_changed` [310]. The server, on receiving that, should call `roots/list` again to update its internal notion of accessible directories [311] [312]. - This mechanism effectively sandboxes the server's operations to certain directories. For example, a tool that reads files should ideally only allow file paths under those roots. It's a security measure (prevent server from wandering the filesystem beyond what's authorized) and also a context hint (server knows where to search for relevant files).

In an integration like VS Code, when you open a folder, that folder is the root. VS Code's extension will support roots, so it will send that information to the MCP server [313]. In the tools example screenshot above, one of the tools was `showRoots` – likely a tool to list the workspace roots, demonstrating how the server is aware of them.

Elicitation and Sampling – Server-initiated AI interactions

While not asked explicitly in the user's request, for completeness we note: - **Sampling**: If the server declares `completions` or specifically uses `sampling`, it can ask the client to have the model generate a message (via `sampling/createMessage` request) [314] [315]. The server provides a prompt (which might include context it has) and the client's AI produces a response that is sent back. This is like the server invoking the model for a sub-task. Use cases include the server doing a multi-step reasoning where it queries the model for intermediate steps. *User control:* The spec says sampling must be explicitly approved by user (since it uses possibly their API credits or might reveal prompt info) [316] [317]. So the client should likely prompt the user "Server X wants the AI to generate text with prompt P – allow?". - **Elicitation**: If the client supports `elicitation`, the server can send `elicitation/create` requests to ask the user for more info during a workflow [318] [319]. The server provides a message to show the user (e.g. *"Please provide your GitHub username"*) and a JSON schema for the expected answer [319] [320]. The client then should pop up a form to the user. The user's answer comes back as the response to that request. Elicitation is a way for the server (and thus the AI's chain of logic) to get clarification or additional data from the user in the middle of a process. Clients must make it clear which server is asking and allow the user to decline or cancel [321]. The result from an elicitation includes an `action` field (accept/decline/cancel) and possibly `content` with the provided data [322] [323].

These two features turn the interaction into a three-way: user, AI, and server all in a loop. They are advanced and likely used for complex agent workflows (for example, if a tool needs user's API key, the server could elicit it through a prompt).

Integration with IDEs (Visual Studio Code) and Best Practices

MCP is designed to integrate into developer tools and other AI-assisted applications. A prime example is **Visual Studio Code's** implementation of MCP, which as of 2025 is in preview [324] [325]. Integrating MCP into an IDE or app involves handling the technical protocol and also presenting the features to the user in an intuitive, safe manner. Here we summarize how an integration can work, using VS Code as a model for best practices:

- **Running/Connecting to Servers**: The client (IDE) needs to start or connect to the MCP server. In VS Code, if an extension includes an MCP server (perhaps packaged as a binary or a Python script), it can be launched via stdio. If the server is remote, the user might configure an endpoint URL and the client connects via HTTP+SSE. VS Code supports both local and remote transports [326]. The user might be asked to trust the server's source, especially if it's remote.

- **Handshake and Capability Use**: The client performs the initialization handshake as per spec. In VS Code, the extension would negotiate all capabilities it supports (which are quite extensive: tools, prompts, resources, elicitation, sampling, roots, authentication, etc.) [327] [114]. VS Code's documentation explicitly lists those, confirming it implements basically the full MCP feature set.

- **Discovering Tools/Prompts/Resources**: After init, the client obtains lists of tools, prompts, resources:

- It may populate a **Tools menu or palette**. VS Code has a "Tools picker" for agent mode where it lists all tools from all MCP servers, grouped by server [203]. The user can check which ones the AI is allowed to use. The tool's `title` and `description` are shown so the user understands what they do [203]. In the screenshot above【37】, we see tools `add`, `getGreeting`, `showRoots` under a server. Users can toggle them on/off quickly.
- Tools are also referenced in the chat UI. When the AI attempts to use one, a confirmation pops up (with the tool's title, description, and the input args) [204] [205], as shown in screenshot 【38】. The user can approve or cancel. If approved, the extension sends `tools/call` to the server. If cancelled, it sends a `cancelled` notification to abort the call.
- The client should implement logic to automatically deny or allow some tools based on policy. For example, if a tool has `readOnlyHint` true, VS Code does not show the confirmation (auto-runs it) [249] [250], to streamline things like simple getters.

- Tools might also be invoked directly by the user via UI (maybe a context menu "Run tool X now"), but typically it's the AI that decides.

- For **Resources**, the client likely creates a UI to browse them. VS Code allows users to open a list of resources (by running "MCP: Browse Resources" command) [149]. It shows resource names (and possibly descriptions). Resource templates are shown as well, often distinguished by needing input (in the screenshot 【39】, we see a list of "Repository Content for specific branch/commit..." which are templates from presumably a GitHub MCP server). When the user picks a resource template, VS Code prompts for the parameters (and if completions are defined for that parameter, it provides suggestions) [328] [329].

- When a resource is selected, if it's text, the client can open it in an editor tab. If it's binary (image/audio), the client might preview it or save it. If the user wants to share it with the AI, the client can insert it as context: VS Code has "Add Context" to attach a resource to the chat prompt [149] [330], meaning it will include the content in the question to the AI.

- If resources update (via notifications), the client updates the open editors or notifies the user. VS Code supports resource updates such that an open resource document updates in real-time if the server sends an update (e.g., log file tailing) [331].

- For **Prompts**, the client likely merges them into the chat's slash commands or a similar UI. In VS Code, prompts become slash commands prefixed with the server name, like `/mcp.my-mcp-server.promptname` [115]. The user can trigger them easily. If a prompt has arguments, VS Code will show a dialog for input, with auto-completion if configured [134] [133], as shown in screenshot 【40】. After the user fills the argument(s), VS Code inserts the resulting prompt messages into the chat and sends them to the model.

- Prompts thus act as "canned expert actions" the user can use without remembering or typing a complex query.

- **Session and Auth**: If a server requires authentication (say it needs to access user's GitHub), the client should handle that gracefully:

- VS Code's integration with OAuth is such that if the server advertises an auth server (via the Authorization spec metadata), VS Code will use its built-in auth support to obtain a token from GitHub or Entra ID [299] [300]. The user likely goes through a familiar OAuth consent page, then VS Code stores the token.
- VS Code then associates that token with the MCP server (in its accounts management). The user can revoke trust via the Accounts menu [301] [300], which likely stops sending the token or disconnects from the server.
- The integration should ensure tokens are scoped properly and not exposed beyond the server's requests.

- For local servers that don't need auth, nothing special is needed. For remote, the client might also verify TLS certificates or allow user to accept self-signed if in enterprise context.

- **Multi-server handling**: An IDE can connect to multiple MCP servers at once (for example, one might provide coding tools, another provides access to a database, etc.). The client should keep them separate and combine their offerings sensibly:

- Tools from multiple servers can be aggregated in the UI (with server grouping as shown).
- Prompts can be namespaced (prefix by server).
- Resources might be listed per server or in different categories (perhaps the Quick Pick shows them grouped by server or by type).
- Each server runs its own JSON-RPC connection (likely a separate process or endpoint). The client must manage each handshake individually.
- When the AI (model) is composing a reply, if multiple servers are available, the client might allow the model to call any tool from any server. This is powerful but also tricky: the model might have to

specify which server's tool. Possibly the naming convention `serverName.toolName` or the client enforces unique names.

- VS Code's prompt naming suggests they use `mcp.<servername>.<prompt>` format in the UI, which implies similar for tools behind the scenes.

- **Error reporting to user**: If something goes wrong (server crashes, version mismatch, etc.), the client should inform the user. E.g., if `initialize` fails with version error, the client might show "The MCP server is not compatible (it supports version X, we require Y)". If a tool call returns a JSON-RPC error, maybe log it or show a notification.

- VS Code likely shows a toast or an output error if an MCP server disconnects unexpectedly or if a request fails.

- **Consent and Safety**: The integration must implement the security principles in UI:

- Make sure the user explicitly **enables** an MCP server (maybe via installing an extension or configuration).
- Possibly ask the user on first use: "Allow AI to use tool X which can modify files?" – so user is aware of capabilities.
- Provide settings to permanently allow certain safe operations without prompt, and always prompt for riskier ones.
- Keep a clear record of what actions were taken (logging to output each tool invocation and result can help debugging and trust).

- The user should always be able to stop or pause an AI agent if it's going out of control with tool usage.

- **Performance**: The client should handle concurrency gracefully. For example, if the AI triggers multiple tool calls in rapid succession (or parallel in different threads of conversation), the client might queue or allow some concurrency depending on server capabilities. Also, streaming results via SSE should be handled asynchronously so that partial outputs (like a tool that streams a large result) don't block the UI.

- **Extensibility**: As MCP evolves (new capabilities or changes in future versions), the client should be built to negotiate versions and degrade gracefully. The VS Code implementation supporting multiple versions via the `MCP-Protocol-Version` header ensures it can talk to older servers.

- **Testing and Debugging**: Tools and resources open up a lot of possibilities but also complexity. A good integration provides ways for developers to test their MCP servers (maybe a command to simulate a tool call, etc.) and for users to see what's happening under the hood (the logging feature, as well as maybe a verbose mode to print all JSON-RPC traffic for debugging if needed).

To illustrate integration, consider a user story in VS Code:

The user installs an extension that provides an MCP server for a cloud database. They open VS Code and connect to their database (perhaps the extension starts the server). VS Code initiates MCP handshake, obtains that the

server offers: a prompt "Generate SQL Query", resources listing database tables, and a tool "execute_query". The user sees a new "MCP Tools" section where they enable the `execute_query` *tool (which is read-only in effect as it just fetches data). They ask the AI in chat: "What are the names of all customers who spent more than $1000 last month?" The AI decides to use the* `execute_query` *tool. VS Code intercepts: shows a dialog "Run execute_query – SQL Query Tool with input: SELECT name FROM customers WHERE spend > 1000 AND month = 'last'" (which the AI formulated). The user clicks Continue. The server runs the query, streams a progress (because it's a large query, progress:50%, etc.) and then returns a result resource link to a CSV file. The VS Code extension gets a resource_link, it automatically fetches it via* `resources/read` *(or perhaps the AI explicitly asks to read it). The results are displayed to the user or summarized by the AI. Throughout, the user had control to cancel (maybe a Stop button on the progress notification).*

This scenario shows many pieces working together: tools, prompts (if the server had any), resources, progress, cancellation.

In conclusion, integrating MCP means implementing the **protocol plumbing** (JSON-RPC messages over a chosen transport) and the **UX** that surfaces these AI capabilities in a user-friendly, safe manner. The latest spec (2025-06-18) provides a comprehensive blueprint for both: from handshake negotiation to session management, from how to structure a tool's JSON schema to how to handle errors and streaming. Following this spec, developers can create AI extensions that are interoperable with multiple clients, and clients can support a growing ecosystem of AI tools and data sources in a standard way. The result is a richer, more powerful AI assistance experience – one where the AI is not a black-box, but an agent that can work with your tools and data under your supervision.

Sources:

- Official MCP Spec (2025-06-18) 332 333 77 21 334 96 118 130 152 335 207 218 235 254 256
- Visual Studio Code MCP Integration Guide 203 205 115 133 301

---

1 2 3 4 5 6 9 10 332 Specification - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/index

7 8 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 98 99 100 101 102 103 104 105 106 292 333 Lifecycle - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/basic/lifecycle

11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 96 334 Transports - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/basic/transports

59 How to build and deploy a Model Context Protocol (MCP) server | Blog
https://northflank.com/blog/how-to-build-and-deploy-a-model-context-protocol-mcp-server

60 Model Context Protocol | AI MCP Server Consulting Experts
https://www.advisorlabs.com/services/model-context-protocol

61 virajsharma2000/mcp-websocket: This server implements ... - GitHub
https://github.com/virajsharma2000/mcp-websocket

62 Enhanced LLM Communication via WebSockets - MCP Market
https://mcpmarket.com/server/websocket

97 137 138 178 179 180 181 182 185 186 247 248 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 311 312 Schema Reference - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/schema

107 Overview - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/server

108 109 110 111 112 113 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 135 136 Prompts - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/server/prompts

114 115 132 133 134 148 149 203 204 205 206 215 246 249 250 251 299 300 301 313 324 325 326 327 328 329 330 331 MCP developer guide | Visual Studio Code Extension API
https://code.visualstudio.com/api/extension-guides/ai/mcp

139 140 141 142 143 144 145 146 147 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 183 184 187 188 189 190 191 192 193 194 335 Resources - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/server/resources

195 196 302 303 304 305 306 307 308 309 310 Roots - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/client/roots

197 198 199 200 201 202 207 208 209 210 211 212 213 214 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 252 253 254 255 256 257 258 Tools - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/server/tools

293 294 295 296 297 298 Authorization - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/basic/authorization

314 315 316 317 Sampling - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/client/sampling

318 319 320 321 322 323 Elicitation - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/client/elicitation