Model Context Protocol (MCP) – Comprehensive Spec & Implementation Guide

Quick Implementation Brief

- **Purpose & Architecture:** The Model Context Protocol (MCP) is an open standard for connecting AI applications (LLMs) with external data sources and tools[1]. It follows a **host–client–server** model: a host AI app spawns one or more MCP **clients** (connectors) that establish JSON-RPC sessions with external **servers** (data/tools providers)[2][3]. This separation means servers focus on specific capabilities (e.g. file data, APIs) while the host coordinates model integration and ensures isolation (servers cannot see the entire conversation or each other's data)[4]. Think of MCP as a "**USB-C port for AI** " – a universal way to plug in contextual data and functions[5].
- **Protocol Basics:** MCP messages use **JSON-RPC 2.0** format over a persistent, stateful connection[2]. Each message is a single JSON object with jsonrpc: "2.0", an id for requests/responses, a method name (for requests/notifications), and params or result fields as appropriate. **Batching of JSON-RPC calls is not supported** in the latest spec[6] – send one request per message. Standard JSON-RPC error codes are used for failures (e.g. -32601 for "Method not found")[7].
- **Transports:** Two transport mechanisms are defined[8], and your implementation can support one or both:
- **STDIO (Standard I/O):** The client launches the server process as a subprocess and communicates via its stdin/stdout streams[9]. Each JSON-RPC message is sent as a **single line** (UTF-8 text) terminated by a newline. **No embedded newline characters are allowed in the JSON payload**[10] – ensure you escape or remove any literal newlines in JSON strings to avoid breaking the message framing. The server should not write anything to stdout except valid JSON-RPC messages (use stderr for logs)[11]. This transport is simple and ideal for local plugins.
- **Streamable HTTP + SSE:** The server runs as an independent service with a single HTTP endpoint (e.g. /mcp), handling multiple clients[12]. Clients send **HTTP POST** requests to transmit JSON-RPC requests (one per POST)[13], and the server replies either with a single JSON response or by **streaming responses/events** via **Server-Sent Events (SSE)** [14]. SSE allows the server to push multiple messages (events) over a long-lived HTTP response. Clients can also open a **GET request** on the endpoint to listen for server-initiated messages (notifications or requests) via SSE[15]. This transport supports richer, bidirectional communication (server can send async notifications). **Be sure to implement SSE properly** : each event data is a JSON message prefixed with data: lines, and if using SSE, advertise Accept: text/event-stream on GET/POST requests[16][17]. The server may intermix JSON-RPC requests/notifications on the SSE stream before the final response to a POST, enabling streaming results[18]. After sending the JSON-RPC

response, the server should close the stream[19] (unless keeping it open for session events).

- **Transport Security & Gotchas:** For HTTP transports, **validate the Origin header on incoming requests** to prevent malicious web pages from connecting to local MCP servers (DNS rebinding attacks)[20]. If your server is meant for local use, bind to 127.0.0.1 rather than all interfaces[21]. Implement authentication for remote servers. Handle **multiple SSE connections** cautiously: a client may open several SSE streams in parallel – your server **MUST NOT broadcast the same message on all streams** (no duplicate events)[22]. Instead, send each message on one stream only; use SSE's built-in **id field** on events to support reconnection (if a stream breaks, the client can send Last-Event-ID on a new GET to resume)[23]. Implementing event IDs and replay on reconnect is recommended to avoid message loss[24]. Also, if your server assigns a **session ID** (via Mcp-Session-Id header in the Initialize response), clients will include it in all requests; enforce this to reject requests missing the correct session ID (HTTP 400) and terminate sessions on your side with 404 when done[25][26]. Clients indicate protocol version via an MCP-Protocol-Version HTTP header on every request after init – make sure to check this and respond 400 if an unsupported version is sent[27]. (By default, if no header, assume an older version like 2025-03-26 for compatibility)[28].
- **Lifecycle & Handshake:** MCP connections have a defined lifecycle[29]:
- **Initialization:** The client **must start by sending an initialize request**[30]. This includes the protocol version (a date string like "2025-06-18" for latest spec) the client supports, a dictionary of the client's capabilities, and some client info (name, version, etc.)[31][32]. The server **must reply** with an InitializeResult containing the negotiated protocol version and the server's capabilities and info[33][34]. **Version negotiation** : if the client's proposed version isn't supported, the server can respond with a different version it supports[35] – the client should disconnect if it can't handle that version. Both sides should end up agreeing on one version; all subsequent messages will conform to that spec. **Capability negotiation** : the client and server each declare which optional features they support (more on capabilities below). Only features enabled by both sides' capabilities can be used during the session[36][37]. For example, if the server offers a tools feature but the client didn't announce support for tools, tool calls should not be used.
- **Post-Init Sync:** After a successful init handshake, the client sends an initialized **notification** (method: notifications/initialized, no id) to signal it's ready[38]. This marks the start of normal operations. **Important:** Until this point, neither side should send other requests except possibly benign pings or log messages[39]. The client should not invoke any server feature methods before receiving the initialize response, and the server should not send requests (aside from optional logging or ping notifications) before the initialized notification arrives[39]. Skipping this synchronization can lead to race conditions (a common pitfall).
- **Operation Phase:** Once initialized, the connection enters normal operation. The client and server exchange JSON-RPC requests and notifications according to the capabilities negotiated[40]. The client will typically call **server-provided methods** (for "server features"), and the server may at times call **client-provided methods** (for "client

features" or utilities). All messages must follow JSON-RPC format and the agreed protocol version rules. Multiple requests can be in flight concurrently (JSON-RPC allows async handling as long as IDs are tracked).

- **Shutdown:** Either side (usually the client) can terminate the session. There is no special MCP "disconnect" message; it's done via the transport: e.g. client closes the STDIO pipes or the HTTP connection(s)[41][42]. For STDIO, the client should close the server's stdin, wait for exit, and if necessary send a SIGTERM/SIGKILL if the server hangs[43]. For HTTP, the client can simply stop sending requests and close any open SSE channels (or send an HTTP DELETE to explicitly end a session if supported)[44]. Design your server to handle sudden disconnects gracefully (e.g. free resources on EOF or client gone).
- **Capabilities Negotiation:** Capabilities are key-value objects sent in the initialize request/response that **advertise optional features** each side supports[36]. Your implementation should correctly declare and interpret these, as they gate what methods can be used. Important capabilities include:
- **Server-side capabilities:**
    - prompts – server can provide prompt templates (with optional listChanged notifications when available prompts change)[45].
    - resources – server provides contextual data (files, documents, etc.), possibly with subscribe (supports subscriptions for updates) and listChanged (notifications when resource list changes)[45][46].
    - tools – server offers executable tools/functions; can have listChanged for tool list updates[34][47].
    - logging – server can send structured log messages to the client[34].
    - completions – server supports autocompletion requests (used for assisting with arguments, e.g. completing a resource template or prompt argument)[48].
    - experimental – a flag for any non-standard extension features (both client and server can advertise an experimental object if they implement unofficial extensions).
- **Client-side capabilities:**
    - roots – client can provide filesystem/project root directories to the server (so the server knows the boundaries of accessible files)[49].
    - sampling – client allows the server to request **LLM completions** (the server can ask the client's AI model to generate text or other content, under user's approval)[50].
    - elicitation – client allows the server to ask the **user for additional info** via a structured prompt (e.g. server needs user input mid-workflow)[50][51].
    - (Clients also implicitly support core JSON-RPC and possibly progress notifications and cancellation – these are usually baseline or utility features.)
- Only if both sides advertise a given capability can the associated feature be used. For example, if a server requires the client to support sampling but the client did not declare it, the server should refrain from calling sampling/createMessage (or consider failing initialization). It's good practice for servers to **fail early** (return an error in initialize) if a

required capability is missing (to avoid runtime errors). For instance, a server heavily dependent on being able to call the model (sampling) might reject a client that doesn't support it. Conversely, clients should treat unknown server capabilities as "feature not available" and continue without them.

- **MCP Server Features (what your server can provide):** Once connected, the following are the main categories of functionality your **server** can implement and expose to the client/host. You can choose to implement any subset, but you **must declare them in capabilities** during init, and follow the MCP's JSON-RPC method conventions for each:
- **Prompts:** Pre-defined prompt templates or conversation snippets that a user can invoke to guide the AI[52]. These are *user-triggered* (e.g. user picks a "Summarize document" prompt from a menu, which the client then fetches from the server). If supported, implement:
    - prompts/list: list available prompt templates (supports pagination via cursor) [53][54].
    - prompts/get: retrieve a specific prompt's content by name (the server returns the prompt's text/messages, possibly filling in user-provided arguments)[55][56].
    - Prompts are identified by a name and include optional metadata like a human-friendly title, description, and a list of arguments (parameters that can customize the prompt)[57][58]. The actual prompt content is typically a sequence of message(s) with roles (user/assistant) and content (text or even images)[59][60].
    - If listChanged capability is true, your server should send notifications/prompts/list_changed whenever the set of available prompts changes (e.g. new template added)[61].
    - *Edge cases:* Ensure prompt content is properly escaped (no stray newlines if using STDIO). Prompt messages can include embedded resources (like an image or file reference) – those would use the same content schema as tools or resources (with type and possibly base64 data or URI)[62][63]. Additionally, the **completion API** (if your server supports completions) can be used to help fill prompt arguments or templates (this allows AI-assisted suggestion of argument values)[64] – implementing completions is advanced (it involves methods for autocompletion) but be aware of the integration.
    - *Gotcha:* Prompts are meant for **user control**[65]. Your server should not attempt to force a prompt without user action (the host UI decides when to call prompts/get). So, focus on making prompts discoverable (clear titles/descriptions) rather than auto-invoking them.

- **Resources:** Read-only context data that your server exposes (files, documents, knowledge base entries, etc.)[66][67]. Resources are typically *attached by the client application* to the LLM's context as needed. Implementations:
    - resources/list: list available resources, each identified by a **URI** (e.g. file:///path/to/file.txt or a custom scheme) and accompanied by metadata like name, optional title (for display), description, MIME type, size, etc.[68][69]. If there are

many resources, support pagination (cursor in params and nextCursor in results)[70][71].

- resources/read: retrieve the content of a specified resource by URI[72][73]. The result includes a contents array (one or more content blocks). Text content is returned with a text field, binary content with a base64 blob field, along with the same resource metadata (URI, name, mimeType, etc.)[74][75]. The server may chunk a large resource into multiple content blocks in the array if needed.
- Optionally, resources/templates/list: list any **parameterized resource templates** (URIs with placeholders) if your server can dynamically fetch resources based on parameters[76][77]. For example, a template might be git://repo/{commit} or db://{query}. These templates allow the client to prompt the user for parameters (possibly via auto-complete) and then form a concrete URI to read.
- If listChanged is supported, send notifications/resources/list_changed when the set of resources (or templates) changes[78] (e.g. new file available).
- If subscribe is supported, implement a way for the client to subscribe to updates on a particular resource: the client will call resources/subscribe {uri} [79], and your server should then push notifications/resources/updated when that resource's content changes[80]. The updated notification typically includes the uri (and optionally a new title or info if changed) to tell the client that resource should be refreshed[81]. Only send updated notifs for clients that subscribed.
- **Annotations:** Resources (and even prompt messages or tool results) can include an annotations object with metadata like audience (who this is meant for: "user" vs "assistant"), priority (0.0–1.0 importance for inclusion), and lastModified timestamp[82][83]. These help the client decide which resources to include in the model's context (e.g. prioritize high-priority ones)[84]. Use these hints appropriately (e.g. mark sensitive data audience as user-only).
- **URI schemes:** You can use standard schemes (file://, https://, git://) or define custom ones for your domain[85][86]. Just ensure they are valid URIs and document any custom scheme's usage. The file:// scheme is common for local files (you might also represent directories with MIME type inode/directory)[86].
- *Pitfalls:* Enforce resource access boundaries – your server should **not expose files outside intended roots**. Typically, the host client will tell you what root directories (via the roots feature) you are allowed to operate in[87][88]. Validate path inputs to prevent traversal attacks[89]. Also handle cases where a file is missing or inaccessible – return an JSON-RPC error (-32002 Resource not found, as recommended)[90][91]. If your server can't support a method (like no resources at all), respond with -32601 Method not found to any resources/* calls[7].
- Keep resource lists up-to-date: if files can change or new ones appear, use list_changed notifications (if enabled) to alert the client, so the UI can refresh. But avoid spamming – batch changes if possible.

    - *Memory consideration:* If resources are large (e.g. big files), be mindful of memory/transport. SSE streaming could be used to send large contents in chunks by sending multiple resources/read partial responses, but the spec typically expects one response per read. You might use progress notifications for long reads or just make the client request smaller pieces. Always respect the **timeouts** the client might impose (a client will cancel if no response within some time)[92].

- **Tools:** Functions or operations that the AI assistant (model) can call through your server to perform actions or fetch information[93][67]. Tools are *model-triggered* (the AI decides to invoke them when it "thinks" calling the tool will help, but the user should usually authorize it). To implement tools:
    - **Tool Definition:** Each tool has a unique name (identifier used to invoke it), plus metadata: a human-readable title and description, an inputSchema (JSON Schema defining what arguments it accepts), and optionally an outputSchema (JSON Schema for the structure of its result)[94][95]. These schemas inform both the user and the AI what the tool expects and returns. You may also include annotations to describe behavior or restrictions (though clients treat these as untrusted unless your server is trusted)[96].
    - tools/list: the client may call this to get the list of available tools and their definitions[97][98]. Support pagination if listing is large. Only include tools that the current user/session should have access to (respect permissions).
    - tools/call: this request is sent by the client when the AI model decides to use a tool. The params will include the name of the tool and an arguments object with keys/values as defined by your inputSchema[99][100]. Your server should execute the corresponding function or action, then return a result.
    - The result JSON should contain either a **successful outcome** or an **error indication**. On success, you return a result with either **unstructured output** or **structured output** :
    - **Unstructured output:** Return a content array of output blocks (similar to resource content). For example, a simple success might be:
    - "result": {
      "content": [
      { "type": "text", "text": "Operation completed successfully." }
      ],
      "isError": false
      }
    - The content can include multiple items and different types: you can return text (plain text)[101], image (with base64 data and a mimeType)[102], audio, etc., depending on what the tool produces. Each content item can have annotations (e.g. mark an image result as for the user or assistant)[103].
    - **Structured output:** If your tool naturally produces JSON data (e.g. an API response object), you can use the structuredContent field. For example, if the tool is "get_weather_data" and outputs a JSON with temperature, humidity,

etc., you would put that JSON object under structuredContent, and **also** include a text representation in the content array (for backward compatibility and for the AI to possibly read as text)[104][105]. The spec encourages including both when using structured output[104][106]. If you provided an outputSchema for the tool, ensure your structuredContent conforms to it (the client may validate it)[107].

- **Resource links:** A special case of output content is type: "resource_link" – your tool can return references to resources (perhaps ones that were not listed initially) by providing a URI and metadata[62][108]. For example, a code search tool might return a resource_link to a file that matches the query. The client can then decide to fetch or display that resource. These links might not appear in resources/list results (they can be ad-hoc)[109]. Make sure the client has permission to access whatever URI you return. If you utilize this, it's good to also implement the resources feature so the client can actually read the link via resources/read if needed[63].
- **Errors from tools:** There are two levels of error handling for tools[110]: 1. **Protocol-level errors:** If the tool name is not found, or arguments are invalid (schema violation), or an internal server error occurs *before* execution, you should respond with a JSON-RPC error response (with an appropriate code like -32602 Invalid params or -32603 Server error) similar to any JSON-RPC method error[111][112]. This tells the client the call itself failed. 2. **Execution errors:** If the tool runs but encounters a runtime issue (e.g. an API call within the tool returned an error, or the result is a logical failure), then return a *success* JSON-RPC response but indicate the failure in the result: include "isError": true and perhaps an error message in the content. For example:
- "result": {
  "content": [{ "type": "text", "text": "Weather API timeout" }],
  "isError": true
  }
- This way the AI model knows the tool call returned an error state (and can act accordingly, e.g. inform the user or try something else). Use isError: true for *expected* errors that are part of normal operation (like "user not found"), whereas protocol errors are for more fundamental issues.
- If your server set tools.listChanged: true in capabilities, send notifications/tools/list_changed when tools become available or unavailable dynamically[113] (this is relatively rare – e.g. a new plugin installed at runtime).
- **Human oversight:** Tools give the AI the ability to take actions (write files, make HTTP calls, etc.). **It's strongly recommended that the host (client app) requires user approval for each tool invocation**[114]. From the server side, you should assume the user may decline an unsafe tool call. Also document clearly what each tool does in its description. A big **no-no** is executing anything on the user's machine without their consent – the host is expected to enforce this, but design your tools with transparency and safety.

    - *Pitfalls:* Make sure tool outputs that are large (images, audio) are handled properly (base64 encoding and not too large to send in one go – consider splitting or compressing if needed). Avoid nondeterministic JSON key ordering if the client or AI will parse structuredContent (follow standard JSON). **Do not trust user-provided arguments blindly** – validate them, as a malicious prompt could trick the AI into calling your tool with unexpected inputs. Also, treat any annotations or extra metadata your server didn't set with skepticism, and similarly clients will ignore or sandbox tool annotations (they are not guaranteed to be safe)[96].

- **Client-Initiated Features (server calling back to client):** If your server declares certain capabilities, it can **make requests to the client** as well (these are the inverse of above). As a server developer, you might invoke these to leverage functionality in the host environment:
- **Roots:** If the client supports roots, you can query the client's defined filesystem/project roots. Typically, this means the user has explicitly granted your server access to certain directories. Use roots/list (as a server, **you send this JSON-RPC request to the client** over the existing connection) to get an array of root URIs and names[115][116]. This could help scope your operations (e.g. know which folder to index). The client may also notify you if the roots change (notifications/roots/list_changed)[117] (e.g. user switched project). *Note:* The server should **never access files outside these roots** , even if it gets a request (enforce that in your implementation)[89][118].
- **Sampling (LLM requests):** If the client supports sampling, your server can ask the host to generate a completion (essentially calling the AI model for help). This is done via sampling/createMessage request, where you provide a prompt (as a list of messages with roles and content) and optionally some preferences[119][120]. The client will (after user approval) have the AI produce a completion and return it in the response[121][122]. This is powerful: it lets your server do things like "chain" model calls (e.g. if the server is a retrieval plugin, it might ask the model to summarize the retrieved info). *Important considerations:* The user must be able to review these prompts and responses (the spec emphasizes a human should vet any AI-to-AI queries)[123][124]. Your server should design these sampling prompts carefully and only request when necessary. The request parameters allow **model preference hints** – because the server might not know what models the client has, it can specify abstract priorities (e.g. prefer speed vs accuracy via speedPriority, intelligencePriority, etc.) and **hints** like model family names[125][126]. The client will map these to an actual model it has access to (for example, you might hint claude or gpt-4, but the client will choose the closest available model matching the priorities)[127][128]. This indirect model selection is a bit of a sharp edge – do not assume the client will use the exact model you hint; always handle the response generically. Also, the maxTokens, systemPrompt, etc., can be provided to guide the generation[120][129].
- **Elicitation (User input requests):** New in the latest spec, if the client supports elicitation, your server can request additional info from the user mid-interaction[130]. This is done with an elicitation/create request, where you provide a prompt message to

display to the user (e.g. *"Please provide your database password"* – but note, **don't request sensitive info!**[131]) and a **JSON Schema** (requestedSchema) describing the expected format of the user's answer[132][133]. The client will show a UI for the user to input the data (structured or simple text) and then return either an acceptance with the data (in a JSON object matching your schema)[134][135], a decline (user refused)[136], or cancel (user aborted)[137]. Your server should handle all three cases. This is useful for interactive workflows (e.g. a server tool needs an API key or wants the user to pick from options). *Gotchas:* The schema is limited to simple, flat structures – only basic types and no nested objects/arrays, to keep it easy for clients to implement[138]. Don't abuse elicitation to ask for passwords or personal data; clients are instructed to prevent that and it violates trust. Use it sparingly and only for necessary additional context.

- **Logging & Progress:** If you declared logging in server capabilities, you can send log messages to the client via the notifications/log or similar (the exact schema is defined in spec's schema reference). This can help with debugging or providing the user with insight into what the server is doing. Keep logs concise and sanitize any sensitive info. Also, the base protocol supports progress notifications – if a long-running request is in progress, you may send notifications/progress with an ID to indicate work is ongoing (and possibly a percentage or status). This can reset client-side timeouts[139]. Similarly, handle cancel notifications: if the client sends a notifications/cancel for an in-flight request (it provides the id of the request to cancel), your server should attempt to halt that operation and eventually not send a response (or send an error if already mid-way). Proper cancellation handling prevents resource waste.
- **Error Handling & Robustness:** Anticipate and handle edge cases gracefully:
- If protocol version negotiation fails (no common version), the server should return an error to initialize (e.g. error code -32602 with message "Unsupported protocol version") and then disconnect[111][112]. The spec's example shows including a list of supported versions in the error data[140].
- If required capabilities are missing, you may also fail initialize with an error (e.g. code -32602 with data explaining "capability X required") – the spec doesn't give a specific code for that, but it's acceptable to treat it as invalid params.
- Use appropriate JSON-RPC error codes for other issues (e.g., -32601 if a method is not implemented by your server, -32602 for bad params, -32603 for unexpected exceptions). You can also define custom error codes (prefer negative values to avoid conflict with JSON-RPC) for domain-specific errors like -32002 for resource not found[90].
- Implement **timeouts** on your side if calling out to external systems (APIs, databases) – this prevents hanging the client. The client will likely also have a timeout and send a cancellation if you take too long[139], but it's best to fail fast if backend is unresponsive.
- Test with out-of-order messages: JSON-RPC allows responses to be sent back asynchronously. For STDIO, ensure your reading/writing loop can handle receiving multiple requests and sending responses in parallel (especially if using async Rust/Python code). For HTTP, be prepared for multiple simultaneous POSTs from the client (the client might not always wait for one call to finish before sending another). Use proper locking or serialization for shared resources in your server.

- **Shutdown cleanup:** Make sure to release file handles, stop background threads, and generally clean up when the connection ends. In Rust, for example, handle SIGTERM or stdin closed events to exit gracefully. The client might abruptly kill the process if it doesn't exit on its own after stdin close[43], so ensure no infinite loops blocking shutdown.
- **Security & Trust Considerations:** The MCP spec places heavy emphasis on security, given that servers can have access to user data and execute code:
- **User Consent:** Always assume the user must **authorize any sensitive action**. The host application is responsible for getting user consent (e.g. allowing a tool call or sharing a file), but your server should still **expose clear info** so the client can inform the user. Provide accurate description for tools and resources, so the user knows what they're granting access to[114].
- **Data Privacy:** Do not transmit user data outside of the MCP channel unless explicitly expected. The host will ensure user data isn't sent to the server without permission[141]. From the server side, if you are pulling data from a user's account (say, their Google Drive via API), treat that data carefully and do not retain it longer than needed.
- **Tool Safety:** Treat tool execution like running untrusted code – because effectively the AI is deciding when to run it. Sandbox what you can, handle exceptions robustly, and **never execute a tool action without the client's go-ahead**[142]. For instance, even if your server has a "delete file" tool, the client UI should confirm with the user – but you should also double-check any irreversible action if possible.
- **OAuth Authorization (for remote servers):** If your MCP server provides access to protected data (e.g. enterprise databases or APIs), consider using the **authorization flow** defined in the spec (which aligns with OAuth 2.0). In the latest spec, an MCP server can act as an OAuth 2.1 **resource server** , and the client as an OAuth client, obtaining an access token for the server[143]. The server should advertise its OAuth authorization server in a resource metadata document (via WWW-Authenticate header on 401 responses, per RFC 8414/ RFC 9728)[144][145]. This is complex, but essentially it allows the user to grant your server access (via an OAuth flow in their browser) rather than sharing raw credentials. If implementing a production server that requires login, following this standard is **highly recommended** to avoid man-in-the-middle token theft[146]. For development or local use, you might skip this (or use env vars for credentials if using STDIO, as suggested[147]).
- **Isolation:** Remember the host's design principle: your server only sees what it needs. **Do not assume you'll get the full chat transcript.** For example, you might only receive a user query if the host decides to send it as part of a sampling request or as a parameter to a prompt/tool. Many servers won't ever see raw user questions. This is by design for privacy[4]. So, build your functionality around the data you explicitly request (resources, user inputs, etc.), not on snooping the conversation.
- **Future-Proofing:** MCP is evolving (versioned by date). The latest is **2025-06-18**[148]. Keep an eye on the spec updates (the **Key Changes** document highlights major revisions[149]). For instance, support for SSE was revamped in late 2024, and elicitation was added in 2025[51]. Design your implementation to be version-aware (you

negotiated a version in init). If you later support multiple versions, you may need conditional logic for older vs newer behavior. However, focusing on the latest spec is best for new implementations (you can include the agreed protocol version in logs or debugging to help).

- **Testing Edge Cases:** Test your server with mis-ordered sequences (client sends something before init – you should ideally ignore or error until init is done), invalid JSON (ensure you handle parse errors and respond with e.g. -32700 Parse error if you get malformed JSON), and multiple simultaneous actions. Also test recovery: e.g., SSE reconnect scenarios (simulate a dropped connection and ensure your server continues the session when the client reconnects with Last-Event-ID).
- **Consent UIs and Footguns:** Ultimately, many "sharp edges" are mitigated by the host UI. But as a server developer, **be clear in what your server is doing** so that the host can inform the user. One classic footgun would be not specifying what data a tool will send externally – if your tool calls an external API with user data, mention that in description. Another is returning content meant for the assistant that includes instructions (could be misused by the model) – if something is only for the user, mark its audience as user so the client might not feed it into the model[83]. These details help maintain alignment and safety.

Modular Deep Dive Documentation

Below, we provide structured documentation for each aspect of MCP, for those who need a deeper understanding while implementing in Rust, Python, or other languages. You can treat each section as a reference module.

1. Base Protocol and Connection Lifecycle

- **JSON-RPC 2.0 Communication:** All MCP messages are JSON-RPC 2.0 objects in UTF-8 text[2]. Familiarize yourself with JSON-RPC if you haven't – it's a simple RPC format with method, params, id for requests, and result or error for responses. Both client and server can send requests or notifications at any time after init, making the connection *bidirectional*. There is no built-in schema enforcement by JSON-RPC, but MCP defines what methods and params are valid. Notably, **JSON-RPC batching (sending an array of requests in one payload) is disallowed in MCP**[6], so process one message at a time. Also, preserve message ordering where it matters (e.g. the init handshake).
- **Connection Lifecycle Phases: Initialization → Operation → Shutdown**[29]:
- *Initialization Handshake:* The client opens the connection (spawns subprocess or opens HTTP) and immediately sends an initialize request with its supported protocol version and capabilities[150][49]. The server responds with matching version (or a fallback it supports) and its capabilities[35][33]. Both share some metadata (like clientInfo and serverInfo objects containing name, version, etc., purely informational)[151][152]. If versions don't match and no common ground, the server should respond with an error listing supported versions[140]. If capabilities are incompatible (e.g. server requires

something), this is the time to fail or warn (perhaps via an error or the instructions field in InitializeResult – which can contain a message to the client UI)[152].

- *Post-init Notification:* The client, after receiving a successful InitializeResult, sends a notifications/initialized (note: no id because it's a JSON-RPC notification)[38]. This tells the server that the client is ready to proceed. According to the spec, the server should wait for this before sending any requests that aren't just trivial pings or logs[39].
- *Operational Phase:* Now the main interaction happens. The client will typically call server methods (depending on what the user does in the UI), and the server might call client methods (depending on its logic). This continues until one side decides to end. Both sides must adhere to using only the features that were agreed upon (e.g., if resources wasn't in capabilities, the client shouldn't call resources/list, and if tools wasn't agreed, the server shouldn't expect any tool calls)[153].
- *Graceful Shutdown:* Usually the client (host) initiates shutdown when the user is done or disconnects the server. For STDIO, closing the server's stdin is the signal; the server should then exit on its own[43]. For HTTP, simply closing connections or not issuing new requests suffices; the client may send an HTTP DELETE to the endpoint with the session header to indicate termination if the server supports that[44]. It's good for the server to clean up session state and perhaps log out if it had authenticated to other services.
- *Lifecycle Edge Cases:* If the server crashes or disconnects unexpectedly, the client should handle that (it might alert the user or attempt reconnect depending on context). As a server, you might not need to do much except ensure any child threads/processes also terminate. If the client disconnects (e.g., user killed the app), your server might get an EOF or no further HTTP calls – you should then shut down after a timeout or immediately if possible.
- **Message Ordering & Concurrency:** MCP doesn't enforce strict ordering beyond the init phase. Multiple requests can be processed in parallel. For example, a client could issue resources/list and tools/list back-to-back; you can handle them concurrently in separate threads or async tasks. Just ensure that when sending responses, you include the correct id that matches the request. JSON-RPC allows responses to be sent in any order; a common pattern is to respond as soon as each is ready. If your server has to maintain sequence for some reason (e.g., to avoid race conditions in your own data), you may queue internally. Also, note that server->client requests (like sampling/createMessage) might be triggered by a prior client->server call or by some internal event. The client is built to handle incoming requests via SSE/STDIO at any time, but again, best practice is to do so within an expected flow (e.g., server might call sampling/createMessage *during* handling of a tools/call if it needs model help, rather than randomly).
- **Error Responses:** Use JSON-RPC error objects with code, message, and (optional) data. The spec provides examples like "Unsupported protocol version" with code -32602 (Invalid params)[140]. For feature-specific errors, consider using JSON-RPC's reserved range -32000 to -32099 for custom server errors (the spec uses -32002 for resource not found as an example[91]). Always include a descriptive message, and if useful, a data object with extra info (like which URI wasn't found, which param was wrong, etc.). This helps in debugging and may be shown to the user or developer.

- **Cancellation:** The MCP defines a notifications/cancel that either side can send to indicate it wishes to cancel a request that's in progress (it will include the id of the request to cancel, in params)[139]. As a server, be prepared to receive a cancel for a long-running operation (especially one you initiated, like a sampling request). The protocol's rule is that after sending cancel, the client will stop waiting for a response[154]. It's up to you to actually halt the work if possible. If you manage to cancel before completing, you simply shouldn't send a response (the client won't expect it). If a response does race to the client despite cancellation, the client may ignore it. It's not an error if a cancellation comes too late – just do your best. Also, if *you* as a server want to cancel something you requested from the client (like an elicitation or sampling), you can similarly send a cancel notification (though typically, for sampling, the user would cancel it via UI which then triggers the client to send you a cancel).
- **Keepalive/Ping:** In long-lived sessions, either side might want to ensure the other is still responsive (especially in HTTP where inactivity might close connections). The spec mentions a ping method (likely mcp/ping or similar) – though not fully detailed in what we saw, it's common to implement a simple notification or request that does nothing (or returns pong). This can also be used during initialization if needed (the spec allowed pings before init completion). Use it sparingly; a better approach is to rely on TCP keepalive or HTTP heartbeat if possible.

2. Transport Details and Implementation Options

- **STDIO Implementation:** If you implement MCP in Rust or Python for local use (e.g. an MCP server that runs on the user's machine), STDIO is often the simplest integration path. The host application (like Claude Desktop or an editor) will launch your program and communicate via pipes.
- *Rust:* You can use standard library stdin()/stdout() in a loop, or a library like serde_json to parse JSON from a buffered reader. **Important:** Read input stream by **lines**. Each incoming JSON-RPC message is terminated by \n. Do not attempt to parse stream of bytes without delimiting – you'll either need to manually buffer until a newline is seen or use something like BufRead::read_line. Be careful if the JSON content might contain \n within strings; however, the spec disallows that unescaped[10]. In practice, the JSON will likely come compact (one line), but it's wise to handle the case of an escaped newline (\\n in the JSON becomes a real newline in the string if naive splitting). Usually, ensuring the sender never outputs literal newlines in JSON (only \\n) avoids ambiguity. After reading a line, trim any whitespace and ensure it's not empty (some implementations might send a newline keepalive).
- *Python:* Use the sys.stdin iterator or readline() in a loop, then json.loads() for each line. Write to sys.stdout with json.dumps(response) + "\n", and flush immediately (to ensure the client doesn't wait). Avoid printing anything else (no stray print() calls!). Direct logging to stdout will break the protocol, so direct logs to stderr or a file.
- Since your server is a subprocess, you might parse command-line args or env vars for configuration (like which port for HTTP if you optionally support both, or debug flags). The host might set env vars for auth tokens if using STDIO (since STDIO can't do

interactive OAuth easily, the spec suggests using environment credentials for STDIO transports)[147].

- *Concurrency:* STDIO is inherently synchronous in delivery, but you can still process requests concurrently by spawning threads/tasks for long operations. Just be careful to avoid interleaving responses at the byte level – if using one thread per request, have a thread-safe queue to send complete response strings to the output in order, or use a mutex around stdout writes. Alternatively, handle one request at a time if that simplicity is acceptable (but that might underutilize parallelism).
- Handle process signals: as noted, if the client wants to kill you, it might close stdin (EOF) or send SIGTERM. Watch for EOF on stdin (read returning empty) to break your read loop and exit gracefully.
- **HTTP + SSE Implementation:** This is more involved but necessary for remote or multi-client servers:
- Essentially, implement an HTTP server with one endpoint (e.g. using frameworks like **Hyper or Axum in Rust** , or **FastAPI/Starlette or Flask in Python** with an SSE support library). This endpoint must accept:
    - **POST** : with a JSON-RPC message in the body. The client will set Accept: application/json, text/event-stream (meaning it can handle either JSON or SSE response)[16]. Your server should read the request body as JSON. Determine if it's a **request** (has id and method), a **notification** (has method but no id), or a **response** (has id and either result or error).
    - If it's a **client- >server request**: you need to process it and respond. You have two ways to respond: 1. **Single JSON response** : just respond with Content-Type: application/json and the JSON object of the response. Do this for simple, quick replies (typical for most requests that return one result). 2. **SSE stream** : respond with Content-Type: text/event-stream and begin sending SSE events. Use this if you plan to send multiple messages or keep the connection open for a while (e.g. a tools/call that streams chunks of data or a long sampling/createMessage that streams partial output and then the final result). According to spec, even when streaming, you should eventually send the JSON-RPC response as one of the SSE events (so the client knows the call is complete)[155][156], then you can terminate the stream. Make sure to include the **proper SSE framing** (data: lines and a \n\n to end each event). Many web frameworks have SSE helpers where you can yield or flush events easily.
    - If it's a **client- >server notification** (no response needed): you should still return an HTTP status. Spec says if server "accepts" it, return **202 Accepted with empty body**[157]. If it's malformed or you reject it, return an HTTP 4xx (maybe 400) and you *may* include a JSON-RPC error object in the body (though with no id, since it was a notification)[14].
    - If it's a **client- >server response** (the client answering a request you sent): you typically don't have to send anything meaningful back — just 202 Accepted if you got it. The spec says treat it like notifications: return 202 if okay[157]. You'd

match this to your pending request by id internally and wake up whatever is waiting.

  - **GET** : this is used by the client to initiate a **receive-only SSE stream**[17]. The client sets Accept: text/event-stream. If your server supports server-initiated messages, you should respond with Content-Type: text/event-stream and keep the connection open, sending events whenever needed. If your server does **not** support a persistent SSE push channel, you can reply with 405 Method Not Allowed to GET[158], and the client will know not to expect push messages (it could then rely on responses to POSTs only). If you do support it, this SSE channel is basically a way to send notifications or requests to the client at any time, without waiting for a POST from the client. For example, you might push a notifications/resources/list_changed event when a resource update happens on the backend.
  - Implementation detail: SSE requires sending the initial headers and potentially a first event. In the older MCP spec, a first event called endpoint was sent, but in the new unified endpoint it's not needed except for backward compatibility. In the **back-compat mode** (if you detect a legacy client), you might need to handle an initial GET that returns an endpoint event (with URL of SSE endpoint), but with the new spec (2025-06-18) the single endpoint handles both POST and SSE.
  - You should handle **Last-Event-ID** : when the client reconnects, you'll get a Last-Event-ID header with the last event ID it saw[24]. If you kept track of event IDs (which should be unique per stream)[159], you can replay any missed events on the new connection so the client catches up. This is especially relevant for notifications or a long streaming response that got cut off. Decide how you store unsent events (could be in memory, or require re-fetching the state).

- *Rust specifics:* Libraries like **Warp or Axum** can handle SSE by providing an sse::reply stream. You may need to spawn a task per client to push events. Ensure thread safety of shared data (each SSE connection likely corresponds to one MCP session, identified by session ID if you use them).
- *Python specifics:* Frameworks like **FastAPI** support event streams (via StreamingResponse). Or you might use **Flask with Flask-SSE** or **Quart** for async. Be cautious with WSGI servers – SSE requires not buffering the response. An async server (Uvicorn, Hypercorn) is often better.
- **Session Management:** For a stateless protocol like HTTP, maintaining a session is crucial. The spec's approach is the server can issue a session ID on the Initialize response (HTTP) and then require the client to include Mcp-Session-Id header on all future requests[25]. Implement this if your server needs to track state across requests (most do). Use a secure random GUID or token. Check every incoming POST/GET for the correct session header (except the first Initialize). If it's missing or wrong, respond 400 or 401. Also provide a way to expire sessions (a timeout or if you get an explicit DELETE). If you see a DELETE with a session, you can free that session's resources and invalidate the

token[44]. Subsequent requests with that session should be 404 Not Found (session terminated)[160].

- **Web Security:** If running as a web service, enable TLS (HTTPS) and require proper auth (like API keys, OAuth tokens as discussed, or even simple auth for now). The protocol itself is transport-agnostic about auth, but real deployments must not accept arbitrary connections without auth. OAuth token should be sent in Authorization: Bearer header by clients once obtained; your server should verify it. The spec aligns with OAuth 2.1 Resource Indicators so that clients include a resource param during auth to avoid token confusion[146]. That's advanced, but if using a library, configure audience checks on tokens (so a token for service A can't be used on service B).
- **Performance:** HTTP and SSE can handle many clients, but be mindful of thread per connection model. Use non-blocking async or a threadpool to handle requests. In Rust, asynchronous frameworks can manage many SSE connections efficiently. In Python, an async framework or a multi-process setup might be needed for scale (since long SSE responses could block a worker thread).
- **Backward Compatibility:** The spec's *Backwards Compatibility* notes[161][162] outline how to deal with older clients/servers (pre-2025 HTTP transport which used separate endpoints for SSE and POST). In summary: if you want to support older MCP clients, you might maintain the old /events SSE endpoint and the old POST behavior (which expected first SSE event to give an alternate POST URL). This complicates your server, so you may decide to skip legacy support unless needed. Likewise, a new client connecting to an old server will try a POST and on 405/404, do a GET expecting an endpoint event to switch to legacy mode[162][163]. This is mostly relevant if you aim to integrate with clients that haven't updated. For a new implementation, you could choose to only support the latest unified transport and document that requirement.

3. Capability Negotiation and Feature Advertising

- **Why Capabilities:** MCP is designed to be extensible and to let servers/clients progressively implement features[164]. Capabilities prevent miscoordination: each side knows what the other can do. It's essentially a feature flag exchange.
- **Structure:** In JSON, the capabilities field of Initialize params/result is an object with keys for each feature category. If a feature key is absent, it means not supported. If present but with an empty object, supported (with default sub-capabilities). If present with sub-fields (like {"subscribe": true}), it's supported with those optional behaviors. For example, {"resources": {"subscribe": true, "listChanged": true}} means the server supports resources and will allow subscriptions and send list change notifications[165][166]. The client might respond with {"resources": {}} if it doesn't need any special sub-feature (client's perspective on resources might not have subfields to declare).
- **Matching Capabilities:** Some capabilities only make sense on one side (e.g. tools, prompts, resources are server-side; roots, sampling, elicitation are client-side). But others like logging or experimental could be on either. The initialize exchange is essentially two one-way declarations, not a negotiation per feature bit. However, **both sides must interpret the combination**. For instance, if server declares tools and client

didn’t , the client will likely ignore tool-related messages. In practice, most client implementations will only call methods for features they know about and see declared by server, and servers will only call client methods they see supported. If there’s a mismatch (server declares something client didn’t or vice versa), the non-supporting side should treat any related method calls as “method not found” errors. To avoid runtime issues, it’s smart to align on these: e.g., if the server sees the client didn’t advertise sampling but the server really needs it, the server can error out or disable that part of functionality.

- **Extensibility:** The experimental capability can be used to try out features not yet in the standard. It could be an object like {“experimental”: {“fancyFeature”: 1}} on both sides. There’s also a provision for new standardized capabilities in future spec versions (like how elicitation was added). When you implement, consider that unknown keys might appear – don’t crash on unknown capability keys; just ignore those you don’t know. Similarly, if you want to use an experimental feature, be prepared that the other side may not acknowledge it.
- **Capability Example:** A Rust server might have a config like:

-
  ```
  let mut server_caps = json!({
  “logging”: {},
  “resources”: { “subscribe”: true, “listChanged”: true }
  });
  if tools_enabled {
  server_caps[“tools”] = json!({ “listChanged”: true });
  }
  if prompt_templates_available {
  server_caps[“prompts”] = json!({ “listChanged”: false });
  }
  ```
- And include that in the Initialize response. The client, if it supports resources and tools too, might have sent an init with e.g. {“roots”: {“listChanged”: true}, “sampling”: {}}. Those two sets don’t conflict; they just indicate what each side can do. After init, the server knows it **can** call roots/list or sampling/createMessage on the client (because client had roots and sampling), and the client knows it can call resources/, _tools/_, prompts/* on the server.
- **Versioning and Capabilities:** The protocol version itself (e.g. “2025-06-18”) implies a certain set of available methods and JSON schema. If you claim that version, you should implement at least the mandatory parts of that spec. Capabilities then toggle optional parts. For example, “structured tool output” was introduced in 2025-06-18; if you support that version and advertise tools, you are implicitly expected to handle the structuredContent field in tool results[167]. If you implemented an older version, some fields wouldn’t exist. Thus, ensure your data structures align with the spec version.

4. Server Feature Details

This section breaks down specifics of **Prompts** , **Resources** , and **Tools** features from the server's perspective with additional insights.

- **Prompts Feature:**
- *Use Case:* Great for providing canned queries or multi-turn preset dialogues. For instance, a server for a coding assistant might offer a "Explain Code" prompt that, when invoked, provides a system message and user message template to the LLM to analyze code.
- *Data Model:* Each prompt is basically a snippet of conversation. The prompts/list result gives metadata. The prompts/get result gives the actual content: typically a list of PromptMessage objects (role + content). Content can include text or even images or file content (if your prompt references an example image, you might embed it as base64 with type: image). You can also include an arguments list in each prompt's metadata to tell the client what inputs it can supply[168]. For example, a prompt "Translate" might have an argument { name: "language", description: "Target language", required: true }. The client UI could prompt the user for that argument when they select the prompt.
- *Dynamic Prompts:* If your available prompts can change (maybe user created new ones or they depend on context), implement list_changed notifications. If not, you can set listChanged: false (or omit it) meaning the set is static for the session.
- *Sharp Edge:* If prompts have large content (like a long system message), just be mindful of JSON size. But since it's one-off on request, it's fine. More importantly, ensure that any placeholder in prompts are properly represented as arguments rather than leaving brackets in the text – because the AI might see those literally. The spec doesn't define templating language (some use e.g. {{var}} in the messages, but better is to let the client or user fill in via arguments).
- *Implementing in Rust/Python:* Likely just store prompt definitions in a list or dictionary keyed by name. On list, return the list (possibly truncated for pagination – maybe not needed if few prompts). On get, lookup by name, combine with any provided arguments (the client might send them). You might need to inject the argument values into the prompt text. E.g. if your prompt content has a placeholder, you could do a simple string replace or use a template engine. Document how arguments are inserted so the client knows (the spec hints that autocompletion can help provide argument values, so perhaps they expect arguments to be independent fields, not embedded).
- **Resources Feature:**
- *Use Case:* Provide documents, files, or any contextual data. Many servers will implement this to give the LLM knowledge it otherwise wouldn't have.
- *Listing Pagination:* The resources/list and resources/templates/list methods accept an optional cursor and return nextCursor if not all items fit[169][71]. Decide on a sensible page size (maybe 50 or 100 items). The cursor could be as simple as an index or token. If you have relatively few resources, you can ignore pagination (return everything and no nextCursor).
- *Resource URIs:* They need to be globally unique within your server's context. file:// URIs should ideally map to actual files if you expose a filesystem. You might also use query-

like URIs, e.g. myapi://user/123. The client might treat unknown schemes opaquely and always fetch via you (with resources/read). If using https:// as a URI, note that the spec expects the client could fetch those directly *without* involving your server[170]. Only use https:// if the resource is truly publicly accessible and you want the client to go fetch it itself (like linking to an external website). If the data needs auth or processing, better use a custom scheme or file:///git://.

- *Read Responses:* We showed text vs binary content. If you have extremely large text, consider splitting into multiple content entries (the spec allows returning an array of content blocks in contents field)[73]. Each block could represent a chunk of a file or a separate file if you allowed multi-read (though normally resources/read is one URI -> one content entry).
- *Subscriptions:* This is a powerful feature – e.g. your server monitoring a file for changes (tailing logs) and notifying the client. Implement resources/subscribe to add a subscription (store the requesting client or session and the URI). The spec doesn't have an explicit unsubscribe; presumably if the session ends or maybe by another call. You could add an unsubscribe method (not sure if spec defines one). On change, send notifications/resources/updated. The client upon receiving that might call resources/read again to get new content. You should include at least the uri in the notification, and you can include updated title or other fields if they changed (the example shows title being sent again)[80].
- *Edge Cases:* If a resource becomes unavailable (file deleted, access revoked), you might send a resources/updated or resources/list_changed (if the list of resources shrinks). If the client then tries to read and you return not found, that's expected. Always handle resources/read of unknown URI with an error (ideally code -32002)[91].
- *Concurrency:* If two read requests come for the same large file, ensure your file reading logic can handle it (maybe lock or just concurrent read if OS allows). Also if a read is in progress and the file updates and triggers a notification, that's fine – the client might get an update soon after reading.
- **Tools Feature:**
- *Use Case:* Expand the LLM's capabilities by letting it call external functions (via your server). Examples: database queries, web searches, math computations, file modifications, etc. Tools effectively turn the LLM into an agent that can act, with the host as a gatekeeper.
- *Tool Design:* When implementing a tool, think of it like designing an API endpoint:
    - Define clearly what it does (for the description).
    - Define the input parameters (JSON schema) as strictly as possible to guide the model/user. Use types, required fields, enums if applicable.
    - If possible, define an output schema so the result can be validated/formalized[171]. This is optional but highly useful for the client and even the LLM (some advanced AI orchestration might use the schema to verify the output).
    - For instance, a "Create Jira Ticket" tool might take title (string) and description (string) and perhaps projectKey (string enum of project codes). The output

might be a JSON with ticketId and url. With an output schema, the client could even directly present a UI based on the returned JSON.

- *Tool Execution:* Depending on your environment, calling external services or performing actions might require async operations or external libraries. In Rust, heavy compute or I/O can be spawned to threads or use async I/O. In Python, use asyncio for concurrent calls if needed (especially if waiting on network).
    - Always catch exceptions inside tool implementation so you can return a proper error in the MCP format rather than crashing the server.
    - If a tool might take a long time, you can start streaming partial output. For example, a web search tool could stream intermediate results as text messages (SSE events with data: lines each containing a JSON with content like "Found result 1…"), and then final result. However, designing the AI's usage of such streaming is tricky – often it's easier to accumulate and return once.
- *Tool Call Flow:* The model in the host, upon seeing the tool list, might decide to call one by name (some advanced LLMs can formulate a JSON-RPC call internally, or the host might do function calling). The client then does tools/call. Your server executes and returns a result. The host then typically inserts the tool's result into the conversation (e.g. "Assistant (tool): ") for the model to continue. If isError:true, the model knows the tool failed and might try something else or apologize.
- *Stateful Tools:* If a tool has side effects (like "write_file"), remember that the server might maintain state. E.g., a "write_file" tool could add or modify a resource. In that case, you should send a resources/list_changed or resources/updated notification accordingly so the client knows the file list or content changed thanks to the tool's action.
- *Multiple Tools and Namespacing:* Ensure tool names are unique. There's no built-in namespacing mechanism in MCP (like all tools share one flat namespace per server). So avoid generic names that might collide if multiple servers' tools are combined (though typically each server is separate). Names like "search" or "open_file" might be fine if only your server provides them. If you foresee name conflicts, use prefixes (e.g. "db_search").
- *Security:* Tools are the biggest risk if misused. For example, a tool that executes shell commands could be dangerous if the AI finds a prompt that convinces it to do something harmful. As an implementer, **limit what your tools can do**. If possible, constrain inputs (use allow-lists, e.g. only allow certain commands or domains to fetch). The host is supposed to confirm tool usage, but you should not assume the AI is always benign. Log tool usage (so users can audit what was done).
- *Edge Case – Structured vs Unstructured output:* With the new spec, structured output allows direct consumption of results, but always include a text version as well[104][172]. The reason is the AI might only read the text if it's not explicitly coded to parse structuredContent. By providing both, you cover both the AI (which might read the text) and a smart client (which might parse the JSON to present nicely to the user).
- *Return as soon as possible:* If your tool can produce some output quickly but more later, consider streaming. Or at least provide progress indication via content messages. The

SSE mechanism allows intermixing messages – just ensure the final one is the actual JSON-RPC response concluding the call.

- **Other Server Features – Logging and Completions:**
- **Logging:** If you advertised logging, your server can send notifications/log (likely the method name is something like that – check schema). The spec indicates "structured log messages"[173]. This could be used for debug info or usage telemetry to the host. E.g., log events might have levels, messages, maybe categories. Keep it lightweight. Too much logging traffic could slow things (especially over SSE).
- **Completions (Autocompletion):** This server capability is a bit different – it likely provides an interface for the client to ask the server for suggestions when filling arguments or other fields. The resource templates and prompt get methods mention autocompletion integration[174][64]. For example, if a prompt argument is a filename, the client might call a completion/complete method on the server to get a list of possible file paths (like IntelliSense). The spec reference to **Completion API** suggests a method where the client provides a partial input and the server returns possible completions (like how language servers do). Implementing this is optional, but if your server has obvious auto-fill (like recent queries, common values), it could enhance UX.
- If implementing, define when to trigger: perhaps on specific arguments (e.g. an argument in prompt definition or resource template that is free-form but completions available). Then handle a method like completion/complete with parameters indicating context. Without the full spec text for it, one can model it similar to how LSP does completion.

5. Client Feature Details (for completeness)

(As a server implementer, you primarily need to know how to respond when using client features, but we document them for completeness or if you also implement a client.)

- **Roots (Client feature):** We covered this mostly – the server can call roots/list to get the accessible root URIs[175][116]. If your server doesn't need it, you can ignore it. If you call it but the client doesn't support it, you'll get a JSON-RPC error (-32601). Always check capability: don't call roots/list unless the client's capabilities in init had a roots object[49]. If you do use it: e.g. a file-indexing server might call it at startup (after init) to know where to index files. The root objects have uri and optional name[176]. They will typically be file:// URIs. You might then restrict your resources to those paths. If multiple roots, you have to handle potentially separate trees.
- The client may update roots (user changes project). If roots.listChanged:true was set by client, be prepared for notifications/roots/list_changed at any time[117]. When you get that, you should probably call roots/list again to get the new set. Or the client might include the updated list in the notification params (not shown in spec snippet – likely not, since the notification had no params in example).
- **Sampling (Client feature):** We described how to call sampling/createMessage[119]. Additional points:

- The content of messages supports text, image, audio just like other places[177][178] (so you could actually have the AI generate an image via some multi-modal model if supported). But text is most common.
- The response from the client will have role (usually "assistant") and the message content, plus model (the model that produced it, e.g. an identifier)[179], and a stopReason (why the generation stopped – could be "maxTokens", "endOfTurn", etc.). Your server might not need to use those, except maybe log which model was used.
- If you plan to call sampling often, design your conversation flow carefully. E.g., if your server is doing an agent that might loop (think ReAct style: observe -> tool -> observe -> tool …), make sure to detect when to stop asking for more sampling. Possibly incorporate a limit or rely on the AI's stopReason. Also ensure the user can intervene if this looping goes out of hand (the host likely enforces that anyway).
- **Elicitation (Client feature):** Use elicitation/create to ask user questions. We covered the basics: one important detail is that the message you send should be user-facing (the client will likely display it directly), so localize or phrase it clearly. The requestedSchema for simple cases can just be an object with one property if you just need one value, or multiple for a small form. The user's response comes as content object if action: "accept". If action: "decline" or "cancel", you might choose to either stop what you needed or proceed with a default assumption.
- E.g., if user declined to give their name, maybe your server continues without that info or uses a placeholder. Or if it was critical, maybe your server returns an error for whatever operation was pending.
- As this feature is new, anticipate changes. If you use it heavily, watch future spec versions.
- **One more client feature – "Roots" advanced usage:** A note, sometimes servers might want to manipulate files via the client (like open a file to edit it, or create new file). MCP does not define server->client methods to write files (for safety, they probably want tools on server to handle file writes with user permission). The roots just define scope. So any file modifications should likely be implemented as Tools on the server (so the server itself does the writing after confirming with user).

6. Implementation Tips (Rust & Python specifics, Pitfalls)

- **Rust:** Leverage strong typing for JSON-RPC messages. You might define Rust structs/enums for each method's params and results, according to the MCP schema (the official schema is in TypeScript on GitHub[180], but you can translate it). Consider using serde_json::Value if you want flexibility, but a structured approach will reduce errors. For example, define enums for method names so you don't handle strings everywhere. Use crates like tokio or async-std for async handling (especially for HTTP server or concurrently reading STDIN).
- Use serde to easily parse JSON. E.g., have a struct for InitializeParams with fields for protocolVersion, capabilities, clientInfo, etc.
- Testing: Write unit tests for parsing a sample initialize request and your response, for handling a tools/call input and producing output, etc. Also test error serialization.

- Logging: Possibly integrate with tracing crate to log debug info, but remember to not send internal debug info to stdout if using STDIO transport.
- Pitfall: **Integer vs float vs string** – JSON-RPC IDs can be number or string. Some clients might use “1” vs 1. Plan to accept both (e.g., use Value or a custom enum for ID in your struct). Also, large IDs might not fit in 32-bit. E.g. OpenAI functions sometimes use UUID strings as IDs.
- Pitfall: If using multi-thread, watch out for shared mutable state (like a list of resources that can be updated by a tool call – protect it with a Mutex/RwLock).
- **Python:** Consider using existing JSON-RPC libraries (like jsonrpcserver or aiorpcX) to handle the protocol framing, but given MCP’s custom bits, you might just do manual JSON handling.
- AsyncIO can simplify SSE: frameworks can push events asynchronously.
- Use Pydantic or similar to define schemas for your data if desired (or at least to validate inputs).
- Memory: Python might struggle with extremely high throughput compared to Rust, but for moderate use it’s fine. Just be mindful of not blocking event loops with long tasks (use await for I/O or offload CPU-bound tasks with threads).
- Pitfall: Floating point precision – JSON might have numbers for e.g. priority 0.9. Python’s float is fine for that. Just avoid accidental conversion issues (always use json module which handles floats).
- Pitfall: Exception handling – do not let exceptions propagate and crash the server; wrap calls in try/except and return JSON-RPC errors accordingly. Uncaught exceptions will likely make your server stop responding (for STDIO, it might just hang; for HTTP, could return 500 without a proper JSON error).
- **Common Pitfalls & “Foot Guns”:** Here’s a recap of critical gotchas to avoid:
- **Sending Non-JSON or Logging to the Wrong Stream:** As mentioned, nothing other than JSON-RPC messages should go over the designated channel (stdout for STDIO, HTTP response body for HTTP). Any stray print or HTML error page will confuse the client. Keep logs to stderr or out-of-band.
- **Not Handling Newlines in STDIO:** Ensure JSON messages are one-liners[10]. If you pretty-print JSON with newlines, the client will think the message ended early. Use compact encoding. If your content strings contain newline characters, they will appear as \n within the JSON string (which is fine), not as actual newline delimiters.
- **Ignoring Security Headers:** If using HTTP, implement Origin check for local servers[20]. The typical implementation: if you receive a request with an Origin header and it’s not from an allowed origin (e.g. not from null or localhost for local servers), reject it. This prevents malicious web pages from doing fetch(“http://localhost:port/mcp”, {method:POST, …}) on the user’s browser to control your server. Also, for local servers, binding to localhost only is a good default[21].
- **Forgetting MCP-Protocol-Version (HTTP):** After negotiating version in JSON, the client **must** include MCP-Protocol-Version header on each request[27]. If you implement a server, you should verify this header (and the value) on each request. If a client doesn’t send it (maybe an older client), you might allow it by falling back to a default (the spec

says assume 2025-03-26 if not provided[28]). But better to enforce it for consistency, or at least log a warning if missing.

- **Multiple SSE streams issues:** If your server supports SSE, design how to handle multiple connections. The spec requires not duplicating messages across streams[181]. So if client opens two SSE connections (perhaps one as a backup), you could choose one as primary to send events. Alternatively, only allow one active stream at a time (you could close older ones if a new one connects, or ignore new connections with a message). However, the spec explicitly allows multiple and suggests using event IDs to mitigate loss[182]. A robust strategy: send on the first stream; if it disconnects and the client reconnects (you'll see Last-Event-ID), resume on the new one.
- **Not Implementing Resend on Reconnect:** This is advanced, but if you claim to support resumable SSE (by using event IDs), then you should buffer or be able to regenerate events that occurred in the downtime. At least ensure that critical notifications (like final tool results) are not lost if the SSE drops. If implementing minimal, you might choose not to support resume – then don't include event IDs at all, and the client will assume it can't recover missing pieces beyond perhaps re-requesting data manually.
- **Mishandling IDs and Responses:** Each JSON-RPC request has an id. Make sure to match response to the same id. If you ever send a request to the client (like sampling), generate a unique id that doesn't collide with any the client has used. Typically, clients use numeric IDs for their requests; you could use a separate number range or strings for yours to avoid collision. E.g., some implementations use negative or prefix like "srv-1234" for server-originated requests.
- **Ignoring Cancellations/Timeouts:** If you don't respect cancels or timeouts, you may cause resource leakage. For example, if the client times out waiting for your response and assumes the request dead, but your server later responds, the client might ignore it but you wasted effort. It's good to have a global timeout on each request (the spec says clients will enforce a max timeout, even if progress notifications come)[183]. Decide on a reasonable default (maybe 30s or 60s). For any lengthy operations, send periodic progress notifications to reassure the client (if you declare support for that).
- **Not Cleaning Up on Session End:** If your server holds any user-specific state (like authentication tokens, open file handles, cached data), free them when session ends. If you assigned a session ID, and you detect the client is gone (or a DELETE comes), purge that session's data. Otherwise, memory or file descriptors can leak on a long-running server handling many sessions over time.
- **Trusting Input Too Much:** Validate everything the client (and by extension the user or AI) sends. For instance, if a tool expects an integer in params, ensure the JSON value is actually an integer type. If you get a string for an int, you might choose to accept by parsing or return an error. Also, watch out for extremely large inputs (the user could paste a huge file content as a prompt argument perhaps). You might enforce size limits.
- **Tool Misuse:** As a final emphasis, a "foot-gun" is giving the AI a dangerous tool. If you implement something like a shell command tool or network access tool, consider adding safeguards (like only allow specific commands or domains). The AI might not always make wise decisions, and you don't want your server to be the vector of an exploit. Keep the user in the loop for such tools.

- **Building a Server Template:** Since you mention creating a server template (likely a reusable skeleton in Rust/Python for MCP servers), aim for a modular design:
- Separate the **protocol handling** (JSON-RPC parsing, transport read/write, message dispatch) from the **feature logic** (what each method does). For example, have a core that reads messages and uses a routing table (method name -> handler function).
- Provide abstractions for features: e.g. a ResourceProvider trait/class that has list_resources(), read_resource(uri), etc., and your main server class calls those when a resources method comes in. Similarly, a Tool interface with an invoke(arguments) method could be used to register tools easily.
- This allows developers to plug in their own logic without rewriting the protocol part. It's much like how Language Server Protocol (LSP) libraries work, since MCP was inspired by LSP[184].
- Include default handlers for things like initialize (the template could auto-respond with configured capabilities), shutdown (clean exit), and maybe a basic logging implementation.
- Make the transport layer swappable: one could run the same server either as a subprocess (reading/writing stdio) or as an HTTP server. For Rust, you might achieve this by feature flags or separate binaries. For Python, maybe a flag to choose between launching a Flask app or stdio loop. In any case, the JSON-RPC dispatch logic is common; only reading/writing differs.
- Provide clear extension points so that implementers in Rust/Python can define new tools, resources, etc., and just hook them in. This will help avoid everyone reinventing wheel (and reduce chance of implementing spec incorrectly).

7. Security Best Practices & Conclusion

Implementing MCP in a robust way requires keeping security and reliability in mind at each step. The spec outlines principles like **user consent, data privacy, tool safety, and limited model visibility**[185][142]. In summary: - Always let the user have control over what data is shared and what actions are taken. - Do not expose more data than necessary; the host will mediate but be mindful if you're, say, reading the entire filesystem vs only allowed paths. - Treat all inputs (from user, AI, or external systems) as potentially unsafe – validate and sanitize. - Keep the protocol version and capabilities in sync to avoid subtle bugs (mismatched expectations). - Use the spec's guidance (like OAuth for auth) to avoid rolling your own insecure solutions. For instance, never hard-code tokens in JSON or send secrets in the clear; use the proper channels.

By following the MCP spec closely and noting the edge cases described above, developers in Rust, Python, or any language can implement reliable MCP servers that greatly enhance what AI applications can do, while maintaining a high standard of safety and interoperability. The end result is an ecosystem of **pluggable AI extensions** that work across different AI clients, fulfilling MCP's promise of being the standard "port" for model context and capabilities[5].

[1] [2] [141] [142] [148] [180] [184] [185] Specification - Model Context Protocol

https://modelcontextprotocol.io/specification/2025-06-18

[3] [4] [164] Architecture - Model Context Protocol

https://modelcontextprotocol.io/specification/2025-06-18/architecture

[5] Model Context Protocol (MCP) - Anthropic

https://docs.anthropic.com/en/docs/mcp

[6] [51] [146] [149] [167] Key Changes - Model Context Protocol

https://modelcontextprotocol.io/specification/2025-06-18/changelog

[7] [87] [88] [89] [115] [116] [117] [118] [175] [176] Roots - Model Context Protocol

https://modelcontextprotocol.io/specification/2025-06-18/client/roots

[8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [44] [155] [156] [157] [158] [159] [160] [161] [162] [163] [181] [182] Transports - Model Context Protocol

https://modelcontextprotocol.io/specification/2025-06-18/basic/transports

[29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [45] [48] [49] [50] [92] [111] [112] [139] [140] [150] [151] [152] [153] [154] [173] [183] Lifecycle - Model Context Protocol

https://modelcontextprotocol.io/specification/2025-06-18/basic/lifecycle

[46] [68] [69] [70] [71] [72] [73] [74] [75] [76] [77] [78] [79] [80] [81] [82] [83] [84] [85] [86] [90] [91] [165] [166] [169] [170] [174] Resources - Model Context Protocol

https://modelcontextprotocol.io/specification/2025-06-18/server/resources

[47] [62] [63] [94] [95] [96] [97] [98] [99] [100] [101] [102] [103] [104] [105] [106] [107] [108] [109] [110] [113] [114] [171] [172] Tools - Model Context Protocol

https://modelcontextprotocol.io/specification/2025-06-18/server/tools

[52] [65] [66] [67] [93] Overview - Model Context Protocol

https://modelcontextprotocol.io/specification/2025-06-18/server

[53] [54] [55] [56] [57] [58] [59] [60] [61] [64] [168] Prompts - Model Context Protocol

https://modelcontextprotocol.io/specification/2025-06-18/server/prompts

[119] [120] [121] [122] [123] [124] [125] [126] [127] [128] [129] [177] [178] [179] Sampling - Model Context Protocol

https://modelcontextprotocol.io/specification/2025-06-18/client/sampling

[130] [131] [132] [133] [134] [135] [136] [137] [138] Elicitation - Model Context Protocol

https://modelcontextprotocol.io/specification/2025-06-18/client/elicitation

[143] [144] [145] [147] Authorization - Model Context Protocol

https://modelcontextprotocol.io/specification/2025-06-18/basic/authorization