ChatGPT

Reference Implementation of a Spec-Compliant MCP Server (Protocol 2025-06-18)

Introduction and Protocol Overview

The Model Context Protocol (MCP) is a standardized JSON-RPC 2.0 based framework that connects AI models (clients) with external services (servers) [1]. A fully spec-compliant MCP server (as of the **2025-06-18** MCP specification) must implement the complete protocol lifecycle and all core capabilities, excluding any experimental features. The server acts as a context provider and tool executor for the client's language model, exposing **Prompts**, **Resources**, **Tools**, **Models**, and **Metadata** functionality as defined by the MCP spec. All communications use JSON-RPC 2.0 message format [2] [3] with strict adherence to protocol rules (no batched requests, unique IDs, proper error objects, etc.). This document describes the design of a reference MCP server for testing clients under fully compliant conditions, including supported transport modes (STDIO, HTTP, WebSocket) and detailed behaviors for each capability.

Key protocol requirements: The server **must** support the MCP handshake sequence (`initialize` → capabilities negotiation → `initialized` notification) [4] [5], honor the negotiated protocol version, and only use features agreed upon during initialization [6]. It must expose standard server capabilities – **tools**, **resources**, **prompts**, **models**, **metadata** – through defined JSON-RPC methods. (The *experimental* capability is excluded in compliance with spec guidelines.) The implementation should follow JSON-RPC 2.0 conventions exactly, including the **removal of request batching** [7] and proper error handling per the spec. In the following sections, we detail the server's initialization process, transport-specific considerations, each capability's API and expected behavior, and example messages that strictly conform to the 2025-06-18 MCP specification.

Initialization Handshake and Capability Negotiation

Handshake Sequence: Upon a new connection, the client initiates an MCP session by sending an `initialize` request. This request includes the client's supported protocol version and advertised capabilities [8] [9]. The MCP server must respond with an `InitializeResult` that confirms a mutually supported protocol version and declares the server's own capabilities [10] [11]. For example, a client might send:

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

The server's response should echo or adjust the protocol version and enumerate all capabilities it supports, for instance:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "protocolVersion": "2025-06-18",
    "capabilities": {
      "prompts": { "listChanged": true },
      "resources": { "subscribe": true, "listChanged": true },
      "tools": { "listChanged": true },
      "models": {},
      "metadata": {},
      "logging": {},
      "completions": {}
    },
    "serverInfo": {
      "name": "RefMCP-TestServer",
      "version": "1.0.0"
    },
    "instructions": "..."
  }
}
```

Here the server declares standard capabilities (`prompts`, `resources`, `tools`, `models`, `metadata`) with any sub-capabilities (like `listChanged` or `subscribe`) set as supported [12] [13]. After the server's successful response, the client finalizes initialization by sending a notification `notifications/initialized` (no ID) to signal it is ready [14]. The server should not send any requests (other than optional pings or logging messages) before receiving this `initialized` notification [15]. Likewise, the client should refrain from other requests until the `initialize` handshake is complete. This ordered handshake ensures both sides agree on the **protocol version** and which features are enabled for the session.

Version Negotiation: During `initialize`, the client proposes a protocol version (typically the latest it supports). The server must either accept that version or respond with an alternate version it supports [16]. In the above example, both sides use `2025-06-18`, the latest stable spec. If no common version is found,

the client should disconnect. Once a version is agreed, all subsequent messages must conform to that spec revision. Notably, for HTTP transports the server expects the client to include an `MCP-Protocol-Version` header on every request matching the negotiated version [17] (discussed more under transports below).

Capability Negotiation: As shown, the `capabilities` fields in the initialize request and response advertise optional features each side can handle [18] [19]. The reference server must list **all standard server capabilities** it implements:
- `prompts` – indicates support for prompt templates [20]
- `resources` – indicates support for shareable context resources [20]
- `tools` – indicates support for model-invokable tools/actions [21]
- `models` – indicates support for multiple model endpoints or model selection (if applicable)
- `metadata` – indicates server metadata can be retrieved (and that the server meets metadata requirements, e.g. OAuth metadata)

Each capability may include sub-capabilities booleans. For example, `prompts` and `tools` use a `listChanged` flag to tell if the server will send notifications when the list of available prompts/tools changes [22] [23]. The `resources` capability includes `subscribe` (if the server supports client subscriptions to resource updates) and `listChanged` (notifications on resource list changes) [24] [25]. In the handshake example above, the server set `subscribe:true` and `listChanged:true` for resources, meaning it can handle resource subscriptions and will notify if new resources become available. The server also advertised `logging` and `completions` capabilities (which are part of the spec for structured logging and argument auto-completion) [19] [26] – these are optional but included here for completeness. The **experimental** capability must be omitted or set empty, as experimental features are not allowed in a strict compliance scenario.

Finally, the server includes a `serverInfo` object with its name and version [27]. This is part of the standard handshake and should be filled with identification details for the client's benefit. After receiving the `initialized` notification from the client, the server and client enter the normal **operation phase** where either can send requests or notifications according to the negotiated capabilities [6]. The server should be prepared to handle all standard methods associated with its declared capabilities, as described in the next sections.

JSON-RPC 2.0 Compliance and Error Handling

All MCP messages use JSON-RPC 2.0 format and semantics [2] [28]. The server must enforce the following guidelines in its implementation:

- **Message Structure:** Every request from client or server is a JSON object with `"jsonrpc": "2.0"`, a unique `"id"` (number or string, not null), a `"method"` name, and (if needed) `"params"` [29]. Responses must carry the same `"id"` and either a `"result"` on success or an `"error"` on failure [3]. Notifications have no `id` and no expected response [30]. The server should reject any message that deviates from this structure (e.g. missing `jsonrpc` field or using an unsupported JSON-RPC version).

- **No Batching:** The MCP spec **removes support for JSON-RPC batching**. The server must not accept an array of request objects in a single message [7]. Each JSON-RPC message (request or notification)

should be processed and responded to individually. If a client erroneously sends a batch (JSON array), the server should respond with a parse or invalid request error for the whole message. In our reference server, we strictly handle one request at a time, as required by the 2025 spec update which explicitly forbids batched requests [7].

- **Unique IDs:** The server should expect that request IDs are never re-used by the sender within the session [31]. It must echo back the same `id` in the corresponding response. If an `id` is missing (which indicates a notification) [32], no response is sent. If an `id` is null or duplicate due to a client bug, the server should treat it as an error (likely `Invalid Request` or `Invalid id`). The spec also disallows `id: null` specifically [31].

- **Error Responses:** When a request fails (due to invalid parameters, an internal exception, unknown method, etc.), the server returns a JSON-RPC error object with an integer `code`, a `message`, and optionally a `data` field providing more details [33]. The MCP spec doesn't define a new error format; it uses standard JSON-RPC 2.0 conventions [3]. However, it does recommend certain codes for common situations. For example, if a client requests a resource or tool that doesn't exist, the server should respond with a *custom application error* code (usually in the -32000 to -32099 range) and a descriptive message. The spec suggests code **-32002** for "Resource not found" errors in resource access [34]. Similarly, `-32601` ("Method not found") would be appropriate if an unknown method is called (e.g. client calls a capability the server didn't advertise). The server must ensure to return **no result field** when sending an error – only the `error` object with code and message, per JSON-RPC rules. For instance, an error replying to an invalid `resources/read` might look like:

```
{
  "jsonrpc": "2.0",
  "id": 5,
  "error": {
    "code": -32002,
    "message": "Resource not found",
    "data": { "uri": "file:///nonexistent.txt" }
  }
}
```

[35]. In this reference design, we use JSON-RPC's standard error codes where applicable (`-32602` for invalid params, `-32600` for malformed JSON, etc.) and meaningful custom codes for domain-specific issues (using the recommended -32000 range). Proper error handling extends to transport-level issues as well (e.g. returning HTTP 400 if a POST body isn't valid JSON as noted later).

By following these JSON-RPC guidelines, the server ensures robust and spec-compliant interactions. Next, we outline how the server operates over each supported transport channel while maintaining these semantics.

Transport Modes and Session Management

The MCP server must support three standard transport modes for client communication: **STDIO**, **HTTP (streamable)**, and **WebSocket**. Each transport encapsulates the same JSON-RPC message flow, but with different framing and connection handling. The server design keeps the JSON-RPC processing consistent across transports, with variations only in how messages are sent/received.

STDIO Transport (Local Subprocess)

In STDIO mode, the MCP server runs as a subprocess of the client (or host application) and communicates via standard input/output streams [36]. The protocol rules for STDIO are:

- The server reads JSON-RPC request objects (and notifications) from **STDIN**, one per line [37]. Each message must be a single-line UTF-8 encoded JSON string terminated by a newline (the spec mandates messages *MUST NOT contain embedded newlines* internally) [38]. The client is responsible for writing well-formed JSON without extra newlines.

- The server writes JSON-RPC responses and any server-initiated requests/notifications to **STDOUT**, each as a single line ending with a newline [39]. The client listens on STDOUT and parses each line as a JSON message. The server must not write any non-JSON output to STDOUT (to avoid breaking the protocol) [40].

- For logging or diagnostics, the server may write to STDERR. According to spec, writing UTF-8 logs to `stderr` is allowed and clients may capture or ignore it [41]. But nothing critical to protocol operation should go to stderr.

- The client launches the server process and typically closes stdin/stdout to terminate. Our server should handle an EOF on stdin as a signal to shut down gracefully.

STDIO transport is simple and ideal for local integrations (e.g. a plugin launching a helper tool). It assumes a 1:1 client-server process relationship. The reference server should implement a loop reading from stdin, processing JSON-RPC messages, and writing results to stdout. This mode inherently supports only one client session per process (no concurrent connections), but it is fully synchronous and stateful.

Streamable HTTP Transport

For remote or multi-client scenarios, the server runs as an independent HTTP service. MCP's **Streamable HTTP** transport (introduced in spec version 2024-11-05 and refined in 2025-06-18) uses a combination of HTTP **POST** and **Server-Sent Events (SSE)** over **GET** [42] [43]. The server must expose a single HTTP endpoint (e.g. `/mcp`) that handles all JSON-RPC traffic. Key requirements for HTTP mode:

- **HTTP POST for Requests:** The client sends each JSON-RPC request or notification as a separate HTTP POST to the server's MCP endpoint [43]. The POST body is the JSON payload (one request or one response per POST). The client includes an `Accept` header indicating it can handle `application/json` and `text/event-stream` content types [44]. For example, a client might do `POST /mcp`

`HTTP/1.1` with body `{"jsonrpc":"2.0","id":2,"method":"tools/list","params":{...}}`. The server then **must** decide how to respond:

- If the request can be answered immediately and in one message, the server **may reply directly** with a regular HTTP JSON response (`Content-Type: application/json`) containing the JSON-RPC response object [45]. In this case, the HTTP status is 200 (OK) if a result is returned, or an appropriate error status (400/500) if the body is an error object. The body itself should be the JSON of the result or error.

- **Streaming via SSE:** If the server needs to send multiple messages (e.g. updates, partial results, or any server-initiated requests) before the final response, it should upgrade the POST response to an SSE stream [46]. To do this, the server responds with `Content-Type: text/event-stream` and HTTP status 200, and keeps the connection open. It can then send a series of SSE events. Each SSE event's data is a JSON-RPC message (encoded as a JSON string). The spec says that if an SSE stream is opened for a request, it **should** eventually include the final JSON-RPC response for that request [47], after any intermediate messages. Once the final response is sent, the server should close the stream. This mechanism allows the server to push notifications or follow-up requests to the client in the context of handling a client's request. For example, when the client calls a long-running tool, the server might stream progress notifications and then the result.

- **Notifications/Responses via POST:** If the client POSTs a JSON-RPC **notification** (no id) or a **response** to a server-initiated request, the server should not reply with a JSON body. According to the spec, the server should return HTTP **202 Accepted** with an empty body on successfully receiving a notification/response [48]. If the input is malformed or unacceptable, the server can return an HTTP 4xx with an error JSON in the body (though no `id` since the original was a notification) [49].

- **HTTP GET for Server Push:** The client may also establish a long-lived receive channel by sending an HTTP GET to the MCP endpoint with `Accept: text/event-stream` [50]. This is essentially creating a persistent SSE connection that the server can use to push **notifications or requests** asynchronously, without being prompted by a specific client action. If the server supports asynchronous server-initiated messages (like resource update notifications, logging messages, etc.), it should encourage the client to open this SSE channel. When the server receives a GET for SSE, it should respond with `Content-Type: text/event-stream` and keep the connection open. All JSON-RPC notifications or requests from server to client can then be sent as SSE events on that channel. The spec allows one or multiple concurrent SSE streams; typically one is enough per session. The client will read events continuously until the stream is closed. The server should send periodic keep-alive comments or pings over SSE as needed to keep the connection alive (optional).

- **Session Identification:** HTTP being stateless, the MCP spec uses headers to maintain session context. After initialization, the server will include an `Mcp-Session-Id` header in its initialize response (or in an `endpoint` SSE event for older transport) to identify the session [51]. The client must include this `Mcp-Session-Id` header in all subsequent HTTP requests to bind them to the session state [51]. Our server should generate a unique session ID (e.g., a UUID) on a new session and accept only matching IDs on that session's requests. If a request comes with a wrong or missing session ID, the server returns 404 or 401 to indicate the client should re-initialize. If the server restarts or loses the session state, it can respond with 404 to any request with an unknown session, prompting the client to start a new session (per spec, client SHOULD re-init on 404 in such cases) [51].

The server may also implement an **explicit session termination**: if the client sends an HTTP `DELETE` to the MCP endpoint with the `Mcp-Session-Id` header, the server can end that session [52]. The spec allows the server to return 405 if it doesn't support client-initiated termination [53]. Our reference design will support DELETE to clean up sessions.

- **Protocol Version Header:** As mentioned, every HTTP request after handshake must include `MCP-Protocol-Version: 2025-06-18` (or the negotiated version) in the header [54]. The server should verify this header. If a request comes without it (and the server cannot infer version from context), the server should default to the previous spec (2025-03-26) for compatibility [55], or respond with error 400 if that's not acceptable. Since our test server is strictly 2025-06-18 compliant, we will require the header and return **400 Bad Request** if the version header is missing or incorrect [55].

- **HTTP Security Considerations:** The server must implement some basic security checks. The spec **requires validating the `Origin` header** on incoming connections to prevent malicious web pages from connecting via JavaScript (DNS rebinding attacks) [56]. Our server should allow only known or whitelisted origins (or at least block forged ones). Also, if the server is running on a user's localhost, it should bind to `127.0.0.1` by default rather than wildcard `0.0.0.0` [57] to reduce exposure [58]. Authentication is also crucial for protected servers (discussed under metadata), but at minimum, an MCP server should reject unauthorized connections or token-less requests if it manages sensitive data.

Example HTTP flow: A client might POST an `initialize` JSON to `https://example.com/mcp`. The server responds with `Content-Type: application/json` containing the capabilities. The server also sets `Mcp-Session-Id: abc123` in this response (e.g. in an HTTP header or as part of JSON—older versions did endpoint events, but we use header). The client then sends `initialized` as another POST (with the session header). If the client then wants to list tools, it POSTs `tools/list` to `/mcp` with the session header and `MCP-Protocol-Version: 2025-06-18`. The server, if it has only a small list, might directly respond with JSON containing the tools list. If the client calls a long-running tool like `tools/call` for a complex operation, the server might respond with `text/event-stream`: the HTTP response stays open and the client receives events, perhaps a `{"jsonrpc":"2.0","method":"notifications/progress",...}` first, then finally a `{"jsonrpc":"2.0","id":X,"result":{...}}` once done. After finishing, the server closes that stream. Meanwhile, the client can also do `GET /mcp` to keep an SSE channel for spontaneous server messages (like `notifications/resources/updated`). Our server should handle multiple POSTs sequentially and one SSE GET channel per session.

WebSocket Transport

WebSocket is not explicitly defined in the MCP spec as of 2025-06-18 (the spec focuses on stdio and HTTP/SSE) [59] [42]. However, many MCP implementations and clients use WebSockets as a custom transport for convenience in full-duplex communication [60] [61]. The reference server will support WebSocket transport in a manner consistent with JSON-RPC operation.

WebSocket behavior: The server would listen on a WebSocket endpoint (for example, `ws://localhost:3000/mcp` or `wss://server/mcp/ws`). When a client connects via WebSocket, it establishes a persistent

bidirectional channel. The JSON-RPC handshake and all subsequent messages are then exchanged as text frames over this socket. Key points for WebSocket mode:

- After the connection upgrade, the client should immediately send the `initialize` request as a JSON text frame. The server processes it and sends back the `initialize` response in another frame, just as it would over stdio or HTTP (no additional HTTP response codes since the handshake is at WebSocket protocol level now). The client then sends `initialized` as a notification frame. At this point, the session is established within that WebSocket connection.

- The WebSocket essentially carries a single MCP session. The server should track session state per WebSocket (assigning a session ID internally). There is no need for `Mcp-Session-Id` headers because the connection itself is the session context.

- The server can send JSON-RPC **requests** or **notifications** to the client at any time by simply pushing a text frame with the JSON content. This makes WebSockets very straightforward for features like server-initiated notifications (similar to what SSE provides in HTTP, but here it's built-in to the channel). The client likewise sends requests/notifications as frames to the server. Multiple outstanding requests in both directions are allowed (subject to JSON-RPC id matching).

- The server must ensure thread-safe or event-loop-safe handling of simultaneous messages. For example, if a client sends a new request while the server is still computing a prior one, the server may handle them concurrently or queue them as appropriate, but each response must eventually be sent with the correct ID. Ordering of independent requests is not strictly enforced by JSON-RPC, but our implementation will likely process them in arrival order unless there's a reason to prioritize.

- **WebSocket framing:** We will use text frames for all JSON content (binary frames are not necessary). Each JSON-RPC message is a complete frame. The server should buffer incomplete JSON text until a full message is received (most WebSocket libraries handle this automatically, delivering complete text messages). There's no need for newline delimiters here since WebSocket frame boundaries serve that purpose.

- **Connection closure:** If either side closes the WebSocket, that terminates the session (similar to a TCP connection for HTTP/1 or process termination for stdio). The server should attempt a graceful shutdown if it detects the socket closing (e.g., finish any current operations if possible, or at least not leave resources hanging). Clients are expected to re-connect and re-initialize if needed.

- **Error Handling:** Protocol errors (malformed JSON, etc.) can be handled by sending a JSON-RPC error response frame. But if there is a severe error (like invalid protocol version or authentication failure), the server might choose to close the socket after sending an error notification. The spec's rules about version negotiation apply similarly – if the client sent an unsupported version, the server can respond with an error and terminate the connection if they cannot agree on a version.

WebSockets essentially combine the benefits of stdio (persistent, low-latency, bidirectional) with network transparency. The reference server's WebSocket implementation will use the same request handlers as other transports; only the framing/read-write mechanism differs. This ensures consistency: a `tools/call` request will produce the same result whether it came via WebSocket or HTTP POST.

Session and State Considerations

Regardless of transport, the server should maintain a consistent **session state** for each client connection. This includes: the negotiated protocol version, the set of enabled capabilities, any open resource subscriptions, and any context (like which model is active, if the server manages models). In STDIO and WebSocket, the session is implicit per process or connection. In HTTP, session is tracked via the `Mcp-Session-Id`. The reference server will use a session object internally keyed by connection or session-id. This allows, for example, that a `resources/subscribe` call registers a subscription tied to that session, and notifications are only sent to that client's SSE or WebSocket.

Timeouts: The spec mentions timeouts and that disconnections should not be misinterpreted as cancellations [62]. In practice, the server might implement a timeout for inactivity or long-running calls. However, it should allow the client to explicitly cancel requests by sending a `notifications/cancelled` message with a matching *progress token* or request ID (the spec defines a cancellation notification mechanism). Our design can include support for `notifications/cancelled` to abort long tool executions if the client requests it. Properly handling cancellation is part of being spec-compliant (to avoid ghost operations consuming resources after client is gone).

In summary, the server supports: - STDIO for local usage, - HTTP+SSE for web-friendly usage, and - WebSocket for full-duplex persistent usage. It handles session continuity and ensures that in all modes the core JSON-RPC message flow and ordering requirements are met. With the transports covered, we now describe each **capability** in detail: what API methods the server must implement and how they behave.

Capabilities and Methods

Tools Capability (Executable Actions)

Capability Declaration: If the server offers tools that the model can invoke, it advertises the `"tools"` capability in the initialize handshake. In our server's `InitializeResult`, `"tools": { "listChanged": true }` indicates that tool usage is supported and the server may send *list changed* notifications when available tools change [23]. The presence of the tools capability means the server must implement the standard tool-discovery and tool-invocation methods described below.

Tool Listing (`tools/list`): This request returns the list of all tools the server makes available to the client model. Each tool is described by a JSON object with its name and metadata. The client calls this via JSON-RPC:

```
{
  "jsonrpc": "2.0",
  "id": 10,
  "method": "tools/list",
  "params": { "cursor": null }
}
```

The `params.cursor` is optional and used for pagination if there are many tools (the spec supports pagination cursors for list operations) [63]. In a simple case, the server returns all tools in one page with a structure like:

```
{
  "jsonrpc": "2.0",
  "id": 10,
  "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": {
          "type": "object",
          "properties": {
            "temperature": { "type": "number" },
            "conditions": { "type": "string" }
          }
        }
      },
      { ... other tools ... }
    ],
    "nextCursor": null
  }
}
```

This example shows one tool **get_weather** with a JSON schema for its input [64]. The `outputSchema` is optional; here we included it to illustrate structured output (the spec update 2025-06-18 introduced support for structured JSON outputs from tools [65]). A tool definition typically contains:

- `name`: a unique identifier for the tool (used to invoke it) [66]
- `title`: a short human-friendly name (optional) [67]
- `description`: a brief description of what the tool does [67]
- `inputSchema`: a JSON Schema object describing the expected arguments for the tool [68] [67]. This ensures the client knows how to format tool inputs.
- `outputSchema`: *optional* JSON Schema describing the structure of the tool's output [69] (if the tool returns structured data rather than just text).

- `annotations`: optional metadata about the tool's behavior or safety (out of scope for this spec discussion, but clients should treat annotations as untrusted unless from a trusted server) [70].

Our reference server will include a set of sample tools to allow thorough client testing. For instance, we might implement:
– **Echo tool**: `echo` – takes a string input and returns the same string (useful for verifying basic tool call mechanics).
– **Sum tool**: `sum` – accepts two numbers and returns their sum (tests structured input and numeric output).
– **Get Weather tool**: `get_weather` – as above, accepts a location and returns a fake weather report (tests external API style interaction and multiline text output).

The `tools/list` response will list these tools. If the list is very long, the server could provide a `nextCursor` to paginate further results [71]; otherwise `nextCursor` is null or omitted. The client is expected to call `tools/list` at startup to discover available functionality.

Tool Invocation (`tools/call`): This method allows the client (or more precisely, the AI model via the client) to execute a specific tool. The request parameters must include the `name` of the tool and an `arguments` object with the inputs conforming to the tool's inputSchema [72]. Example request:

```
{
  "jsonrpc": "2.0",
  "id": 11,
  "method": "tools/call",
  "params": {
    "name": "get_weather",
    "arguments": { "location": "New York" }
  }
}
```

Here the client is invoking the `get_weather` tool with `"location": "New York"` [73]. The server will look up the tool by name, validate the arguments against the schema, and execute the corresponding function. The response should contain either a result or an error. On success, the result is typically given in a `content` field (or structured fields if using structured output). For example, a non-error response might be:

```
{
  "jsonrpc": "2.0",
  "id": 11,
  "result": {
    "content": [
      {
        "type": "text",
        "text": "Current weather in New York:\nTemperature: 72°F\nConditions:
Partly cloudy"
```

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

[74]. In this example (drawn from the spec), the tool returned a text result describing the weather [75]. The spec represents tool outputs as an array of **ContentBlock** objects (here one with `type: "text"`). This design allows returning rich media as well, e.g. an `image` content with a base64 or URI, or `structuredContent`. If our `get_weather` had an outputSchema and we chose to return structured data, we could use a `structuredContent` type (for instance, return a JSON object with temperature and conditions). In the simple case above, we just returned a text block. The `isError: false` indicates the tool ran successfully. If the tool itself encountered an application error (like an API failure), the server might still return a `result` with `isError: true` and the content detailing the issue, *or* alternatively it could produce a JSON-RPC error response. The spec suggests using the result with `isError` for tool-level errors that are not protocol errors [75]. Our server will follow this: runtime errors in tool execution will be reported in the `result.isError = true` along with any error message in the content, whereas invalid tool name or bad arguments produce a JSON-RPC error (e.g. code -32602 for invalid params or -32601 if tool name not found).

Tool List Change Notifications: If the server's available tools can change at runtime (for example, a user enables a new plugin or an admin hot-loads a new tool), it should notify the client. Since our server declared `"listChanged": true` for tools, it promises to send a `notifications/tools/list_changed` notification whenever the set of tools is added to or removed [76]. The notification has no params; the client, upon receiving it, is expected to call `tools/list` again to get the updated list. For example:

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

[77]. Our server will emit this if, say, a new tool module is loaded dynamically. For a testing reference server, tools may be mostly static, but we include this for completeness.

Tool Execution Details: Each tool in the server is implemented as a function or command. The server should execute them sandboxed from the model (the AI model only sees the results). For potentially dangerous actions, the spec strongly advises a "human in the loop." While our automated test server may not have a human UI, we should simulate safe behavior. For instance, we might design the server such that it doesn't actually perform file writes or external calls unless configured, and instead returns dummy outputs. The spec suggests UIs should require user approval for tool calls [78], but in our test scenario, we'll assume the client has already handled any necessary approval before sending the `tools/call` to us.

We will, however, abide by safety in implementation: e.g., if a tool is meant to run a shell command, we restrict it to a safe subset or a mock. This is out-of-scope for spec compliance (which focuses on protocol), but worth noting for a real server. In terms of protocol, our server should also handle tool calls that take

time: it can send interim `notifications/progress` events if needed (especially over SSE or WebSocket) to update the client. The MCP spec defines a `ProgressNotification` that can be used to report ongoing status, keyed by a `progressToken` associated with the original request [62]. We could implement that for long-running tools. For brevity, we won't detail the progress mechanism here, but it's part of being fully compliant for lengthy operations.

Example Tools: To ground this, imagine our server's `tools/list` returns three tools: - `"echo"` (title "Echo", description "Echoes input text", inputSchema expects `{ "message": "string" }`). - `"sum"` (title "Sum Two Numbers", description "Calculates sum of two numbers", inputSchema expects `{ "a": "number", "b": "number" }`, and maybe an outputSchema with a single number result). - `"get_weather"` as shown.

A client could invoke them by name. For `echo`, the result might be a simple text content repeating the message. For `sum`, we might return a structured result like `{"total": 15}` or a text `"The sum is 15"`. These examples will be clearly marked as **developer-defined tools** in documentation; they are not part of the MCP spec itself but serve to validate client interactions. The important thing is that the **format** of requests and responses (including the schemas and content encoding) strictly follows the MCP specification, which our design ensures by using the same patterns as the spec's examples.

Resources Capability (Contextual Data Access)

Capability Declaration: The server declares `"resources"` in its capabilities if it provides contextual data to the client (files, documents, database entries, etc. that the model can read). In the handshake, the server can specify two flags under `resources`: `"subscribe"` and `"listChanged"` [24]. For full compliance, our server will support both: `"subscribe": true` (allowing clients to subscribe to resource changes) and `"listChanged": true` (notifying clients when the set of available resources changes) [25]. If a server had neither, it would list an empty object for resources [79], but we aim to implement all features.

Listing Resources (`resources/list`): This request retrieves a directory of resource items that the server makes available for reading. The server might expose files in a project, knowledge base articles, or any structured data identified by URIs. The client sends a request:

```
{
  "jsonrpc": "2.0",
  "id": 20,
  "method": "resources/list",
  "params": { "cursor": null }
}
```

The response includes a `resources` array, each entry describing one resource, plus an optional `nextCursor` for pagination [80] [81]. For example:

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

```
  "id": 20,
  "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
  }
}
```

[82]. Each resource has:
- `uri`: A unique identifier in URI form (e.g. file://, https://, git://, etc.) [83]. The URI scheme indicates how the client might treat it. Our server might use **file:///** URIs for local files, possibly custom schemes for other types.
- `name`: A short name or filename for display [83]. In the example, "main.rs" is the file name.
- `title`: Human-friendly title (optional) [84]. Here it's a more descriptive title of the file.
- `description`: Longer description (optional) of the resource content or purpose [84].
- `mimeType`: MIME type if known [85]. E.g. `text/x-rust` for a Rust source file, `text/markdown` for a README, `application/json` for a JSON data file, etc. This helps the client decide how to handle or display it.
- (Optionally, `size` in bytes if known, especially for binary resources) [86].

The server should list all relevant resources. In our test server, we might, for instance, expose a small virtual file system: e.g., `file:///notes/todo.txt` or `file:///data/custom.json`. The listing returns the top-level resources or a subset based on context. If there are too many resources, pagination via `cursor` is used (the server gives a `nextCursor` token to fetch the next page).

Reading a Resource (`resources/read`): Once the client knows a resource URI, it can request the server to fetch the contents. The request requires a `uri` parameter:

```
{
  "jsonrpc": "2.0",
  "id": 21,
  "method": "resources/read",
  "params": { "uri": "file:///project/src/main.rs" }
}
```

The server looks up the resource by URI, reads its content from wherever it's stored (filesystem, database, etc.), and returns it in the response. The result includes a `contents` array of content blocks [87] [88]. For a simple file, typically one content block is returned, either as text or binary:

```
{
  "jsonrpc": "2.0",
  "id": 21,
  "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}"
      }
    ]
  }
}
```

[89]. In this example, the server returned the contents of a Rust source file as a text block (note the `text` field holding the file content) [90]. The content object echoes some metadata (uri, name, title, mimeType) for convenience and then includes either a `text` field (for text resources) or a `blob` field (for binary data) with the actual content [91] [92]. The spec distinguishes: - **Text Content:** If the resource is textual, use a `text` field (UTF-8 string).
- **Binary Content:** If not text (e.g., an image or PDF), use `blob` field containing base64-encoded data, and set the `mimeType` accordingly (e.g. image/png) [93].

Our server will automatically choose based on MIME type or file type: if it's a recognized text type, we send text; otherwise base64 blob. In either case, the client will thus receive content in-line in the JSON result. (For extremely large resources, a server might choose to break them into multiple content blocks or require streaming, but the spec usually expects `resources/read` to return the whole content in one go. Since this is a test reference, we can assume reasonably sized resources.)

If the requested URI is not found or not available, the server should return an error. As noted earlier, a suitable error code is **-32002** with message "Resource not found" [35] (including the URI in error data). Our implementation will do that for invalid URIs.

Resource Change Notifications: With `"listChanged": true` declared, the server should notify the client when the set of resources changes (e.g., a new file is added or an existing one removed from context). The notification is `notifications/resources/list_changed` with no params:

```
{
  "jsonrpc": "2.0",
  "method": "notifications/resources/list_changed"
}
```

[94]. After this, the client knows to call `resources/list` again to get the updated listing. Our server will issue this if, say, a new resource becomes available during the session (for example, if the server monitors a directory and a new file appears).

Additionally, because we support `"subscribe": true`, the server allows the client to subscribe to specific resource updates. The client would send `resources/subscribe` with a target URI to watch:

```
{
  "jsonrpc": "2.0",
  "id": 22,
  "method": "resources/subscribe",
  "params": { "uri": "file:///project/src/main.rs" }
}
```

If accepted, the server will respond (usually just an empty result `{}` or some acknowledgment), and then in the future, if that resource's content changes, the server will push a `notifications/resources/updated` message. An update notification typically includes the `uri` and possibly updated metadata like a new title or modification time:

```
{
  "jsonrpc": "2.0",
  "method": "notifications/resources/updated",
  "params": {
    "uri": "file:///project/src/main.rs",
    "title": "Rust Software Application Main File"
  }
}
```

[95]. (The inclusion of `title` here is just an example from the spec – it shows that some metadata can be included, perhaps the title changed or it's re-sent for convenience.) On receiving this, the client could automatically call `resources/read` again for that URI to get fresh content. Our server will implement `resources/subscribe` and track subscribers per session. For testing, we might simulate an update (for example, on a timer or via an external trigger, change the content of a file and send out `updated` notifications).

Resource Data Types and Constraints: We already covered the content blocks. The spec also defines **Annotations** for resources, which our server can utilize [96]. Annotations include: - `audience`: who the resource is intended for (e.g., `["assistant"]` if only for the AI to see, or `["user"]` if meant for user consumption) [97]. - `priority`: a number 0.0 to 1.0 indicating importance (1.0 = crucial context, 0.0 = purely optional) [98]. - `lastModified`: timestamp of last change [99].

Our test server can provide these in the resource definitions to see if clients honor them. For example, a resource might be annotated with `audience: ["assistant"]` to hint that it's background data the user

might not need to see, and `priority: 0.8` to indicate it's fairly important. Clients might use that to auto-include certain resources in prompts.

Security for Resources: The server should ensure that only authorized resources are exposed. If our server was dealing with a real filesystem, it must not allow arbitrary file paths outside allowed scope. Since this is a reference server, we'll assume a controlled set of resources. The spec's security notes say: validate all resource URIs, enforce access controls for sensitive data, and ensure binary data is properly encoded [100] [101]. We will do so by whitelisting known URIs and encoding binary as base64 in `blob`.

Example: Suppose our server makes available two text files (`notes.txt` and `plans.md`) and one image file (`diagram.png`). `resources/list` would list them with URIs like `file:///resources/notes.txt`, etc., and appropriate MIME types (`text/plain`, `text/markdown`, `image/png`). The client could then `resources/read` any of them. If the client subscribes to `notes.txt` and the server (perhaps via some internal logic) updates that file, it will send a `resources/updated` notification for that URI. This sequence tests that the client can handle dynamic context changes.

In summary, the **resources capability** allows the client to retrieve additional context data from the server in a structured way. Our server will implement listing, reading, subscription, and notifications to fully exercise the spec-defined behavior for resources.

Prompts Capability (Predefined Prompt Templates)

Capability Declaration: The server indicates support for prompt templates by including `"prompts"` in its capabilities. In the handshake, it can set `listChanged: true` under `prompts` if it will notify prompt list changes [22]. We will enable this flag in our server to cover that functionality. The prompts capability means the server provides predefined conversational templates or messages that a user (or client) can insert into the model's context to guide it.

Listing Prompts (`prompts/list`): The client invokes this method to discover what prompt templates are available. Request format:

```
{
  "jsonrpc": "2.0",
  "id": 30,
  "method": "prompts/list",
  "params": { "cursor": null }
}
```

The server responds with a list of prompt definitions. For example:

```
{
  "jsonrpc": "2.0",
  "id": 30,
  "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
  }
}
```

[102] [103]. In this snippet, there is one prompt template named "code_review". The fields of each prompt definition include:
- `name`: Unique identifier for the prompt (used to request it) [104].
- `title`: A short human-readable name (optional but recommended) [105]. If not given, clients might display the `name`.
- `description`: A description of what the prompt is or does (for the user's understanding) [103].
- `arguments`: An array defining what arguments (placeholders) the prompt can accept [103]. Each argument can have a `name`, a `description`, and whether it's `required` [106]. (The spec also allows an optional `title` for arguments, similar to tools, but in the example above none is shown explicitly beyond name/description/required.)

In our example, the "Request Code Review" prompt requires one argument: the code snippet. The prompt is essentially a template that will incorporate that code. The client's UI might allow a user to choose this prompt and fill in the "code" argument, then send a request to actually get the prompt content via `prompts/get`.

Our reference server will define a set of sample prompts. For instance, besides "code_review", we might have:
– A prompt `"bug_report"` with a template for filing a bug report given steps to reproduce, etc.
– A prompt `"refactor"` that asks the model to refactor provided code.
We will advertise them in `prompts/list`. If the list is long, again, `nextCursor` is used for pagination (but we can assume a small set, returning all at once with `nextCursor: null`).

Getting a Prompt (`prompts/get`): After discovering a prompt by name, the client can retrieve the full prompt content (which often includes a series of message roles or a formatted text) by calling `prompts/get`. The request requires the `name` of the prompt and an `arguments` object mapping any required arguments:

```
{
  "jsonrpc": "2.0",
  "id": 31,
  "method": "prompts/get",
  "params": {
    "name": "code_review",
    "arguments": {
      "code": "def hello():\n    print('world')"
    }
  }
}
```

[107] [108]. The server will locate the template for "code_review", substitute the provided arguments (in this case inject the code snippet), and return the composed prompt content. The response structure:

```
{
  "jsonrpc": "2.0",
  "id": 31,
  "result": {
    "description": "Code review prompt",
    "messages": [
      {
        "role": "user",
        "content": {
          "type": "text",
          "text": "Please review this Python code:\n def hello():\n
print('world')"
        }
      }
    ]
  }
}
```

[109] [110]. Here, the server returned a `description` (which can reiterate or refine what the prompt is for) and a list of `messages`. Each message in the prompt is typically a role + content pair (like messages in OpenAI chat completions). In this example, the prompt consists of a single user message with some text content asking the assistant to review code (the code provided was inserted into the text) [111]. In general, a prompt template could include multiple messages, e.g., a system message with instructions and a user message with the user's part. The content can be multi-modal: not only text, but potentially image or audio content blocks if a prompt included those (the spec allows prompts to include images or references to resources as part of content) [112]. Our server's prompts will likely just use text content for simplicity.

The `prompts/get` result essentially gives the client a ready-made conversation snippet to prepend or otherwise include in the model's context. The client might directly feed the returned `messages` into the model's context window. The model will then continue the conversation from there.

One thing to note: the spec allows prompt arguments to be **auto-completed** via the completion API [107] (reference to "the completion API" suggests if an argument is not provided, the server might try to fill it using its own AI capability). Our server likely will not implement auto-fill of prompt arguments (since that involves the server itself calling an LLM to complete an argument). Instead, we'll assume arguments are provided fully by the client/user. If we wanted full compliance, we would also implement the optional `completion/complete` method (part of the `completions` capability) to assist with argument suggestion [26] – but since `completions` is an extra and the prompt spec explicitly mentions it as optional, we can omit that for now. We will simply enforce that required arguments must be present, otherwise return an error (code -32602 for missing params, for example).

Prompt List Change Notification: With `listChanged: true`, the server should notify if prompts availability changes. The notification is `notifications/prompts/list_changed` with no parameters [113], analogous to the tools and resources cases. In practice, prompt sets might change if, say, the server loads a new prompt pack or the user creates a custom prompt. Our test server might not dynamically add prompts at runtime, but to be thorough, we could simulate it (e.g., after some time, send a `prompts/list_changed` to indicate a new prompt was added). The client then knows to call `prompts/list` again.

Usage and Interaction: Prompts are **user-controlled** by design [114]. Typically, a UI might list the available prompts as shortcuts (for example, a slash-command menu in a chat UI) [115]. The user explicitly triggers one, causing the client to call `prompts/get` and insert the returned messages into the conversation. From the server perspective, it simply provides these templates. Our reference server will store prompt templates (likely hard-coded or loaded from files) and serve them on request.

Example prompt template: To illustrate, consider a `"code_review"` prompt template file that looks like:

- System role: "You are a senior engineer. The user will provide code, and you will review it for potential improvements."
- User role: "Please review this ${language} code:\n```${language}\n${code}\n```"

And it has arguments `language` (optional, default to say "Python") and `code` (required). If the user selects this prompt and provides code (and maybe the language), the server on `prompts/get` would fill in those slots and return the two messages properly formatted (with the code enclosed in markdown, etc.). The example we gave above was simplified to a single user message for brevity, but it's easy to imagine multi-message templates.

Our server's prompt definitions will include any such placeholders and how to substitute them. We'll ensure the JSON structure of `PromptMessage` (role + content) is exactly as in spec. The roles should be "user", "assistant", or potentially "system" (the spec likely allows "system" as well though it's not explicitly shown in the snippet we cited; typically system messages are allowed to set context). For compliance, we restrict roles to those defined in MCP's schema for messages (commonly user/assistant, and possibly system or function depending on spec's alignment with chat standards). Given the spec snippet only mentions "user" or "assistant" for prompt messages [112], we'll use those two to be safe.

In summary, the **prompts capability** in our server allows testing clients to fetch pre-made conversation starters. We'll implement the listing and fetching exactly as per MCP spec, and support notifications for any prompt list changes.

Models Capability (Model Selection and Management)

Capability Declaration: The `"models"` capability is included to indicate the server can interface with or manage multiple underlying AI models. (While the core MCP spec focuses on context, many real servers connect to model APIs; thus providing a way to list or select models is considered a standard feature in practice.) We declare `"models": {}` in the server's capabilities if we want to expose model information to the client. The absence of sub-fields suggests no specific sub-capabilities (the models capability is generally straightforward: listing and possibly selecting models).

Listing Available Models: The server can implement a method `models/list` to enumerate which AI models or model endpoints it can use for generating responses. For example, if the server is a proxy to multiple models (GPT-4, GPT-3.5, Claude, etc.), the client might query this to allow the user or the system to pick a model. The request would be:

```
{
  "jsonrpc": "2.0",
  "id": 40,
  "method": "models/list"
}
```

(We can allow a params object for future expansion such as filtering, but none is needed for a basic list.) The server's response might look like:

```
{
  "jsonrpc": "2.0",
  "id": 40,
  "result": {
    "models": [
      {
        "name": "gpt-4",
        "description": "OpenAI GPT-4 (2025-06-01 snapshot)",
        "contextLength": 8192
      },
      {
        "name": "gpt-3.5-turbo",
        "description": "OpenAI GPT-3.5 Turbo (2024-09 version)",
        "contextLength": 4096
      },
      {
        "name": "claude-v2",
        "description": "Anthropic Claude v2.0",
        "contextLength": 100000
      }
    ]
```

```
  }
}
```

This is an illustrative example. Each model entry could contain various info: a `name` (identifier to use when referring to the model), a human-friendly `description`, maybe technical details like `contextLength` (max tokens) or `provider`. The exact schema for model info isn't fixed by MCP spec, but including such fields is useful. Some implementations treat model listing as a core feature; for instance, one open-source server provides a REST endpoint `GET /models` to list models it supports [116]. Our reference will mimic that functionality in JSON-RPC form.

We will suppose that our test server might be connected to multiple dummy model backends. However, since the MCP client in a typical scenario already has an LLM (the client is often the one doing the actual completion calls using provided context), the *models capability* might be less commonly used. It is more relevant if the MCP server itself acts as an LLM service (like an inference service). Still, to be comprehensive, we include it.

Selecting/Activating a Model: If the server can actually switch which model it uses for tasks like prompt completion or tool output, there could be a method like `models/select` or an analogous concept. Some servers implement an activation by a separate call (e.g., the profullstack server has `POST /model/:modelId/activate` [117]). In our JSON-RPC interface, we could define `models/use` or `models/set` (not part of official spec, but a logical extension) to let the client tell the server "use this model for future operations." For testing, it might suffice to have the listing and assume the client chooses one at initialization via some config. We will, however, describe how the server could handle a request to switch models if it came.

For example, the client might send:

```
{
  "jsonrpc": "2.0",
  "id": 41,
  "method": "models/select",
  "params": { "name": "gpt-4" }
}
```

The server would then set its active model to GPT-4 (assuming it's available and maybe requires an API key which the server has). Response could be an acknowledgment:

```
{
  "jsonrpc": "2.0",
  "id": 41,
  "result": { "active": "gpt-4" }
}
```

Alternatively, if the server keeps multiple models available simultaneously (not just one active), the concept of "select" might not be needed. The spec doesn't define this, so we treat it as an implementation detail. For our purposes, we can assume the server has one primary model or role (like for autocompletion tasks in `elicitation` or `completion` capabilities) and listing models is mainly informational.

Model Metadata: The models capability can also tie into preferences. The MCP spec includes a notion of `ModelPreferences` that a server can send to the client when requesting a completion [118] [119]. For example, if our server needs the client's LLM to respond and has preferences (like "prefer a faster model over a more accurate one"), it can specify numeric priorities or hints. This is more relevant when the server calls `sampling/createMessage` (the server telling client to have the model generate something). In our context, if the server itself doesn't produce LLM completions, `ModelPreferences` might not come into play.

However, if we imagine our server might do something like a *Completion Tool* (e.g., a tool that itself calls an LLM—like how some MCP servers wrap an LLM API as a "tool"), then having model selection is critical. For full compliance, we simply ensure the server can list models and acknowledge a selection. The actual use of multiple models internally will depend on the server's purpose.

Example use case: Consider a scenario where the MCP client (host) is ChatGPT and the MCP server provides a custom LLM service. The client might show the user a list of models offered by that server (via the models capability). The user could pick one, and then when the server is later asked to do something (like complete a prompt or run a tool that involves text generation), it uses the chosen model. Our test server might not truly have different models, but we can simulate by naming a couple and ensuring the mechanism is in place.

For testing a client, verifying it can parse the `models/list` output and maybe send a `models/select` is the main goal. We should clearly mark that actual model switching is a no-op or simulated in this test server (unless we integrate some dummy completions for each model).

In summary, the **models capability** lets the server inform the client about AI model options. We implement `models/list` returning model info. If needed, we accept a `models/select` (or we could design `models/activate`) call. This covers the expectation of exposing model-related controls as part of a "fully spec-compliant" server, even though the core spec doesn't standardize it – it's considered a standard extension by convention in many MCP servers [116].

Metadata and Server Information Capability

Overview: The server's **metadata** refers to general information about the server and its requirements (especially authentication/authorization data). In the MCP spec, certain metadata is exchanged and discovered outside of the JSON-RPC calls, whereas other metadata can be retrieved via defined mechanisms. We ensure our server exposes all necessary metadata in a spec-compliant way.

Server Metadata in Handshake: In the `initialize/initialized` handshake, the server already provides some metadata: the `serverInfo` object with its `name`, optional `title`, and `version` [27]. Our server did this in the handshake example (name "RefMCP-TestServer", version "1.0.0"). This identifies the server implementation to the client. We also included an `instructions` field in the initialize result

(which can convey human-readable instructions to display to the user about the server usage, as per spec optional field) [120]. Those are part of metadata about the server's identity and usage.

Protected Resource Metadata (OAuth2): A critical part of server metadata, as of MCP 2025-06-18, is how a server advertises its **authorization requirements**. MCP servers are now considered OAuth 2.0 Resource Servers [121] [122]. This means if a server requires authorization (e.g. needs an access token from an OAuth issuer), it must expose a standard OAuth **Protected Resource Metadata** document. According to RFC 8728 (OAuth 2.0 Protected Resource Metadata) and the MCP spec's adoption of it, the server should host a `.well-known` JSON document (usually at `/.well-known/oauth-protected-resource`) that includes fields like `authorization_server` URLs, supported scopes, etc. [123] [124]. Our server, being a test reference, can either operate in an open mode (no auth required) or a protected mode. For completeness, we will implement a metadata endpoint that conforms to RFC 8728 when running in protected mode.

Concretely, if our server requires OAuth tokens, it will serve (likely over HTTP) at path `/.well-known/oauth-authorization-server` or `/.well-known/oauth-protected-resource` the JSON that tells clients where the auth server is. The spec says the MCP server must include an `authorization_servers` field in the protected resource metadata pointing to its Authorization Server(s) [125]. For example, the content might be:

```
{
  "authorization_servers": ["https://auth.example.com/"],
  "resource": "https://api.example.com/mcp",
  "scopes_supported": ["read", "write"]
}
```

(This is illustrative; exact fields follow RFC8414 and RFC8728). Additionally, if a client connects without a token or with an invalid token, the server should respond `401 Unauthorized` and include a `WWW-Authenticate` header that indicates the URL of the resource metadata [126] [127]. Our server will do this: e.g.,

```
WWW-Authenticate: Resource=\"https://api.example.com/.well-known/oauth-
protected-resource\"
```

So that the client can discover where to get auth. This mechanism is part of the *metadata discovery* process that MCP mandates for secure servers [128].

Metadata via JSON-RPC: The question references a `metadata/get` method. While the core spec does not define a specific JSON-RPC method named `metadata/get`, some servers or tools may implement one to query server metadata or settings. For instance, a particular MCP server might have a custom method to fetch its configuration or status. In our design, we ensure that at minimum all needed metadata is available

through the official channels (handshake, `.well-known` endpoints). However, for testing convenience, we could implement a simple JSON-RPC endpoint as well:

- `metadata/get`: returns a summary of server info and capabilities at runtime. This could include the same info from handshake (server name, version) as well as perhaps the list of capabilities enabled and any relevant URLs. This is not mandated by spec, but a client might call it as a sanity check. One existing server's API lists a `GET /api/mcp/metadata` endpoint to get "metadata about the instance and available capabilities" [129] [130]. We can mirror that idea in JSON-RPC form.

If we implement `metadata/get`, the response might be:

```
{
  "jsonrpc": "2.0",
  "id": 50,
  "result": {
    "name": "RefMCP-TestServer",
    "version": "1.0.0",
    "capabilities": {
       "prompts": { "listChanged": true },
       "resources": { "subscribe": true, "listChanged": true },
       "tools": { "listChanged": true },
       "models": {},
       "metadata": {},
       "logging": {},
       "completions": {}
    },
    "authRequired": false
  }
}
```

This is largely redundant with information from `initialize`, but it can be useful if, say, a client connects and wants to double-check the server's self-description, or if the server's capabilities changed mid-session (though capabilities are negotiated at start and cannot spontaneously change, except experimental enabling which is not allowed here).

We included `authRequired` as an example field indicating whether the server expects OAuth tokens. If `authRequired: true`, the client would know to obtain a token. Alternatively, we could list an `authorization_server` field here for convenience, but since OAuth metadata is supposed to be fetched via the well-known route, it's better not to duplicate it in JSON-RPC.

Server Metadata Fields: In addition to OAuth info, server metadata might include things like: - Allowed network origins (if using CORS for HTTP), - The purpose or domain of the server (e.g. "this server gives access to your GitHub data"), which could be in `instructions` or some metadata field. - Any static resource links the server wants to share (some servers provide a `metadata` resource or a static about page).

However, these go beyond the spec's strict requirements. For being spec-compliant: the main thing is supporting the OAuth metadata discovery process and providing correct info in handshake. Our server will do both. If a client requests a well-known metadata, we serve it. If a client fails to auth, we respond with 401 and `WWW-Authenticate` directing them to that metadata [126].

Developer Mode / Unauthenticated Testing: For testing purposes, we might run the server in a no-auth mode (no token needed). In that case, we can set `authRequired: false` in the hypothetical `metadata/get` and either not host the well-known metadata or host one indicating no auth necessary. The spec says authorization is OPTIONAL and not used for stdio at all [131]. So, if our server is just on localhost via stdio or WebSocket, we likely skip auth. If via HTTP, we could simulate an OAuth protection if we want to test a client's handling of auth flows.

Metadata Change Notifications: There is no explicit notification like "metadata changed" defined, because server metadata (like auth endpoints or own version) generally doesn't change during a session. So we need not implement any `metadata/list_changed`. The capability "metadata" as listed in the question likely refers to supporting the metadata retrieval and compliance aspects discussed, rather than a specific notification flow.

Conclusion on Metadata: Our server will: - provide identification and capability metadata in the initialize response, - expose OAuth2 metadata at a well-known endpoint if configured for auth (the presence of `"metadata"` capability could implicitly mean "this server follows the OAuth metadata spec"), - optionally implement a `metadata/get` JSON-RPC method for completeness, returning static info about the server's state.

This ensures that a client or tester can always obtain information about what the server is and what it expects. It aligns with the official spec's emphasis on OAuth resource metadata and general transparency of capabilities.

Example Usage and Compliance Verification

Bringing it all together, we can outline a typical interaction with the reference MCP server to demonstrate full compliance:

1. **Initialization:** A client connects (via one of the transports) and sends `initialize`. The server responds as described, advertising `tools`, `resources`, `prompts`, `models`, `metadata` and more [132] [133]. Both sides agree on version 2025-06-18. The client sends `initialized` [14]. At this point, the client knows from capabilities that it can proceed to use all the standard features.

2. **Discovering Capabilities:** The client might first call `metadata/get` (if available) or simply rely on the known capabilities. Suppose it calls `tools/list` to see what it can do. The server returns the list of sample tools (e.g. echo, sum, get_weather) with proper JSON schema definitions [64] [67]. The client could display these to a user or just note that it can use them in reasoning.

3. **Using a Tool:** The client (or the AI agent running in the client) decides to invoke a tool. For example, the AI wants to get the weather in Phoenix, so it sends `tools/call` with name "get_weather" and argument `"location": "Phoenix"`. The server executes the dummy get_weather tool and

streams back a result. If using HTTP SSE or WebSocket, perhaps the server first sends a `notifications/message` log (if logging was enabled) or some `progress` (not strictly necessary for a quick call), and then the final result with content "Sunny and 85°F" or similar. The client receives the JSON result and gives it to the AI model which integrates that into its conversation.

4. **Accessing a Resource:** The user via the client decides to load some context, e.g. asks "Please summarize my TODO list." The AI sees there's a resource "todo.txt" in the `resources/list`. The client calls `resources/read` for `file:///resources/todo.txt`. The server returns the text content of the file [90]. The client then includes that text (or a trimmed version) in the prompt for the AI model. If the file is updated on disk and the server notices, it will send `resources/updated` so the client knows the content changed (if user had the file open, for example).

5. **Using a Prompt Template:** The client has a UI where the user selects a saved prompt. Say the user chooses "Request Code Review". The client calls `prompts/get` with name "code_review" and provides the code snippet argument. The server returns the structured prompt messages [109]. The client inserts those messages at the beginning of the chat conversation (e.g., one system or user message that frames the code review task). The AI model then proceeds with generating a response following that prompt. The result is that the user gets a nicely structured output guided by the template.

6. **Model Selection:** If the server had multiple models and maybe it was performing completions, the client might call `models/list`. The server returns a list containing e.g. GPT-4, GPT-3.5, etc., or in another scenario, different knowledge-base versions. If the user or system wants to change model, the client calls `models/select` (or this might have been done out-of-band via config). The server will then either route future requests to the selected model or remember the preference. In our test, we might simulate this by just acknowledging and maybe printing a log that model changed.

7. **Authentication (if applicable):** If the server required an OAuth token, initially the client's first attempt to do something (other than `initialize` which might be allowed unauthenticated to get the auth info) would result in a `401 Unauthorized` HTTP error with `WWW-Authenticate` header pointing to the metadata URL [127]. The client would fetch `/.well-known/oauth-...` (outside of MCP JSON-RPC, via plain HTTP GET) [124], find the `authorization_server` info, then go obtain a token (the flow might involve user login, etc., not in scope here). Once the client has an access token, it would include it in an `Authorization: Bearer <token>` header on all subsequent HTTP requests [134] [135]. Our server would validate the token for each request (ensuring token's audience matches our server's URI as required [135]). If token is good, the request proceeds. This entire flow would be tested by ensuring the server's metadata and 401 responses are correct. For simplicity, our reference might run in an open mode, but it's designed to be easily switched to secure mode to test a client's OAuth handling. (Notably, STDIO and WebSocket might use API keys or other auth – spec says for STDIO, use environment or none [131], but we won't complicate that here.)

Through these interactions, we validate that **every message and behavior strictly conforms to the MCP spec**. The JSON schemas we return match the official schema (e.g., our `tools/list` and `tools/call` results align with the spec's defined `ListToolsResult` and `CallToolResult` structures [136] [74], our `PromptMessage` objects follow the content format with `role` and `content` [137], etc.). We also ensure that we handle edge cases like unknown methods (return proper JSON-RPC error), maintain the lifecycle

state (don't process other requests before initialization complete), and don't use any deprecated features (like no JSON-RPC batch arrays, no old HTTP SSE endpoint separate from the unified one, etc.).

By implementing all these capabilities – tools, resources, prompts, models, and metadata – and following JSON-RPC 2.0 and MCP 2025-06-18 spec requirements, this reference server provides a robust testbed. It can be used to validate MCP client behavior (e.g., a ChatGPT plugin connector or another AI assistant) under fully spec-compliant conditions, guaranteeing that if the client works with this server, it should work with any proper MCP server implementing the same spec.

Sources: The behaviors and examples above are derived from the official MCP specification (revision 2025-06-18) and related documentation, ensuring accuracy. For instance, JSON message formats follow the spec's schema definitions [2] [3], and all capability methods use the protocols as documented in the spec (e.g. `tools/list` [136], `tools/call` [74], `resources/list` [81], `resources/read` [89], `prompts/list` [102], `prompts/get` [109]). Error handling and the no-batch rule follow the latest spec updates [7]. By adhering to these references, we ensure the server is fully compliant with MCP v2025-06-18.

---

[1] [121] [122] [123] [124] MCP 2025-06-18 Spec Update: AI Security, Structured Output, and User Elicitation for LLMs | Forge Code
https://forgecode.dev/blog/mcp-spec-updates/

[2] [3] [28] [29] [30] [31] [32] [33] Overview - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/basic

[4] [5] [6] [8] [9] [10] [11] [12] [13] [14] [15] [16] [18] [19] [20] [21] [27] [120] [132] [133] Lifecycle - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/basic/lifecycle

[7] [26] [65] Key Changes - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/changelog

[17] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [62] Transports - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/basic/transports

[22] [102] [103] [104] [106] [107] [108] [109] [110] [111] [112] [113] [114] [115] [137] Prompts - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/server/prompts

[23] [63] [64] [66] [67] [68] [69] [70] [71] [72] [73] [74] [75] [76] [77] [78] [136] Tools - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/server/tools

[24] [25] [34] [35] [79] [80] [81] [82] [83] [84] [85] [86] [87] [88] [89] [90] [91] [92] [93] [94] [95] [96] [97] [98] [99] [100] [101] Resources - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/server/resources

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

[61] WebSocket MCP Bridge MCP Server by yonaka - PulseMCP
https://www.pulsemcp.com/servers/yonaka-websocket-mcp-bridge

[105] [118] [119] Schema Reference - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/schema

116 117 GitHub - profullstack/mcp-server: A generic, modular server for implementing the Model Context Protocol (MCP).
https://github.com/profullstack/mcp-server

125 126 127 128 131 134 135 Authorization - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/basic/authorization

129 130 GitHub - FalkorDB/FalkorDB-MCPServer: FalkorDB-MCPServer is an MCP (Model Context Protocol) server that connects LLMs to FalkorDB
https://github.com/FalkorDB/FalkorDB-MCPServer