ChatGPT

Ensuring MCP Client Implementation Meets the Latest Spec

Your MCP client (the **consumer** side) should strictly adhere to the latest **Model Context Protocol (MCP)** specification. The current stable MCP spec revision is **2025-06-18** [1], which represents what most developers expect. This means your implementation must include all standard features defined in that spec – and nothing beyond (no experimental extensions) – to ensure compatibility and easy testing. Below we break down the key aspects you should verify:

MCP Specification Version and Scope

- **Protocol Version:** Use the latest protocol version string in the handshake (as of now, `"2025-06-18"`) unless negotiated otherwise. The spec explicitly notes this revision as the **latest** [1], so make sure your client advertises/supports it during initialization.
- **No Experimental Features:** The MCP spec defines an `experimental` capability for any non-standard extensions [2]. To stay within **standard** usage, avoid relying on any features not in the official spec. For example, do not assume support for unofficial transports or prototype fields unless they're part of the 2025-06-18 spec. In practice, your client's capability negotiation should **not** request or require anything under the `experimental` flag (those can be added later if needed). Other developers will expect only the standard behaviors.
- **JSON-RPC Compliance (No Batching):** MCP is built on JSON-RPC 2.0 for message formatting, so your client should format requests and handle responses accordingly. Importantly, the latest spec **removed support for JSON-RPC batching** (sending multiple calls in one request) [3]. Ensure your client sends one request per JSON-RPC message (with a unique `id` each time) and does not attempt to batch multiple calls in a single payload – since batching is no longer supported in the current spec.
- **Ease of Testing:** Keep the implementation straightforward and modular so it's easy to invoke in test cases (e.g. using Node's built-in `node:test` or similar frameworks). This typically means clean separation of concerns (protocol handling vs. business logic) and providing simple methods to connect/disconnect, send a request, and await a response. A well-structured client will make it trivial to write tests that call into it and verify behavior.

Initialization and Capability Negotiation

Before any tool usage, an MCP client must perform the **initialization handshake** with the server as per spec:

- **Initialize Request:** Your client should start by sending an `initialize` JSON-RPC request. This includes the protocol version it supports, the client's own capabilities, and some client info (name, version, etc.) [4] [5]. For example: `"method": "initialize", "params": { "protocolVersion": "2025-06-18", "capabilities": { ... } }`. Make sure the `protocolVersion` you send is the latest you support (e.g. `"2025-06-18"`).

- **Server Response:** The server will reply with its supported protocol version (possibly adjusting if it doesn't support the one requested) and a set of server capabilities [6]. Critically, to use tools, the server's capabilities **must** include a `tools` object. According to spec, a server that supports tools will advertise something like: `"tools": { "listChanged": true }` in its capabilities [7]. The `listChanged` flag indicates whether the server can notify the client if the available tools change at runtime (your client should handle or ignore such notifications appropriately). The presence of the `"tools"` capability signals that the `tools/list` and `tools/call` operations are available. Ensure your client checks the server's capabilities and only proceeds with tool usage if supported (most servers will include it if they are meant to expose tools).
- **Finalize Handshake:** After receiving the server's capabilities and agreeing on a version, your client should send an `initialized` notification to complete the handshake [8]. Only after this can normal operations (like listing or calling tools) proceed. Adhering to this sequence (initialize → response → initialized) is part of being 100% spec-compliant. Other developers' clients/servers will expect this exact flow, so your implementation should already handle it correctly.
- **Version Negotiation:** During initialization, if the server responds with a different `protocolVersion` than requested, your client must decide if it supports that version. The spec advises using the latest version supported by each side and disconnecting if there's no overlap [9]. Since you want to be up-to-date, ideally your client supports at least 2025-06-18. After negotiation, if you're using HTTP transport, remember that the client **must include** an `MCP-Protocol-Version: 2025-06-18` header on all subsequent HTTP requests to confirm the agreed version [10]. (Double-check that your HTTP requests include this header once the handshake is done.)

By ensuring the above initialization logic is in place, you make your client fully compliant from the start of the session.

Listing Available Tools

Once initialized, the client should retrieve the list of tools the server offers. This is done via the `tools/list` RPC call, which is part of the standard spec:

- **Request Format:** The client sends a JSON-RPC request with method `"tools/list"`. In JSON form it looks like:

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

The `cursor` param is optional and used for pagination if the server has many tools; you can send `null` or omit it to get the first page [11]. Typically for testing or most use cases, the initial call will return the full list (or first page) of tools. Ensure your client constructs this request exactly as specified (correct method

name and JSON structure).
- **Response Format:** The server responds with a list of tool definitions in the result. According to the spec, the response will have a structure like:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "tools": [ ... ],
    "nextCursor": "..."
  }
}
```

The `tools` array contains objects describing each tool. For example, a tool entry might look like this (from the spec example):

```
{
  "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": ... (if provided)
}
```

[12]. Each tool entry includes:
- `name` – the unique identifier used to call the tool.
- `title` – a human-friendly name (optional, for display).
- `description` – a description of what the tool does.
- `inputSchema` – a JSON Schema object defining what arguments this tool expects. In the above example, the tool requires a `"location"` parameter of type string [13]. The schema specifies parameter names, types, and which are required. Your client should parse or at least **respect** this schema so it knows how to format the arguments when calling the tool. For instance, if a tool expects a string `location`, your client must provide a string for that field.
- `outputSchema` – (optional) a JSON Schema for the tool's output, if the server provided one. Not all tools will have this, but if present, it tells the expected structure of the result (particularly for structured outputs). Your client **can** use this to validate or interpret the response, but at minimum it should be aware it exists.

- `annotations` – (optional) additional metadata or hints about the tool. These are mainly advisory and for trust/safety. The spec says clients should treat annotations as untrusted unless from a trusted server [14], so you don't need to do much with them in a generic client beyond maybe logging or ignoring them.

    - **Pagination:** The response also contains `nextCursor` if there are more tools than fit in one response [15]. For completeness, your client should handle this (i.e. if `nextCursor` is provided, you could call `tools/list` again with that cursor to get more tools). However, in many cases (and likely in testing), the server will have a manageable number of tools and you get them in one go. It's good to implement or at least stub this logic so you remain spec-compliant.

Testing tip: To verify your implementation, you can call `tools/list` after initialization and check that you receive a JSON-RPC **result** containing a `tools` array. For example, in a Node test you might assert that the response has `result.tools` and that one of the entries has the expected fields (name, description, etc.). This ensures your client's request was understood and that it can parse the server's reply. The actual tool details will depend on the server (in testing you might use a known test server or a mock), but the presence of a well-formed tools list is what matters.

Calling a Tool

After obtaining the tool list, your client should be able to **invoke a specific tool** using the `tools/call` method. Ensuring this works for both HTTP and WebSocket (if applicable) is key for testing. According to the spec:

- **Request Format:** The client sends a JSON-RPC request with method `"tools/call"` and parameters specifying which tool and what arguments. For example:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "tools/call",
  "params": {
    "name": "get_weather",
    "arguments": {
      "location": "New York"
    }
  }
}
```

[16]. Here, `"name"` is the tool's identifier (as provided in the list), and `"arguments"` is an object whose structure must conform to the tool's `inputSchema`. **This is where using the correct parameter types is crucial** – e.g., if the schema says `location` is a string, pass a string (as shown with `"New York"`). If the tool required, say, a number or an object, you'd need to provide that. Your implementation should be prepared to format the arguments exactly as expected by the schema. (In some cases, you might even

validate the arguments against the schema before sending, to catch errors early, though simply trusting your inputs and sending them is also fine as long as they match the spec.)

- **Response Format:** The server will execute the tool and return a JSON-RPC response. On success, the response's `result` will contain the tool's output. The MCP spec supports two styles of output:

- **Unstructured content:** The result includes a `content` field, which is an array of content items (text, images, etc.), plus an `isError: false` flag indicating the tool ran successfully [17]. For example, a successful response might look like:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "content": [
      {
        "type": "text",
        "text": "Current weather in New York:\nTemperature:
72°F\nConditions: Partly cloudy"
      }
    ],
    "isError": false
  }
}
```

[17]. In this example, the tool returned a text message with the weather info. The `content` array could include other types too (images, audio, etc.) each with a `type` and associated data, but for many simple tools it will just be a block of text. Your client should handle this by reading the `result.content` array. For basic testing, you might not need to do much with the content except recognize that it's there. The key part is that `isError` is false, which tells you the call succeeded.
- **Structured content:** In newer MCP versions, tools can alternatively return a structured JSON object in a `structuredContent` field (especially if they have an `outputSchema`). In such cases, the `result` might look like: `{ "structuredContent": { ... }, "isError": false }`. The spec says that for backwards compatibility, servers *should also* include a serialized version of that structured data in the `content` array as a text item [18]. Whether your client specifically handles `structuredContent` or not, it should at least not break if it encounters it. You might choose to parse it if an `outputSchema` was provided. However, since the question is focusing on standard support, just be aware of it. Most tools in basic usage will probably return simple text content unless they're designed to supply JSON data back.

- **Error Handling:** If something goes wrong (e.g. invalid arguments or tool internal error), the server would return a JSON-RPC error response instead of a result. This would include an `"error"` field with code/message. Your client should handle that by recognizing an error response versus a normal result. For now, assuming **"nothing besides standard spec"** and a correct call, you expect no error. Just ensure your implementation can catch a JSON-RPC error and surface it (perhaps via a rejected

promise or callback error). In testing, you might not intentionally trigger an error, but it's good to have this path covered for completeness.

Testing tip: To test tool invocation in Node, you can do the following: after retrieving the tool list, pick one tool (for example, the first tool in the list or a specific known tool if you know the server's offerings) and call it with dummy but valid arguments. Then assert that you get back a JSON-RPC **result** with `isError === false`. You don't need to validate the content of the result beyond that for this connectivity test. In other words, the test should confirm that the **call was successful and got a 200-OK style response**. For HTTP, a "200-like" response means the HTTP status was 200 and the JSON-RPC `result` is present. For a WebSocket (if used), it means you received a JSON message with the corresponding `id` and a `result`. The actual content (e.g. the weather text) can be ignored in the test – the presence of a well-formed result is enough. This ensures your client can invoke a tool end-to-end.

(If you want extra rigor, you could also test that passing a wrong type to `arguments` yields a JSON-RPC error from the server, but that's beyond the core "happy path" test and depends on server validation. The main goal is verifying the spec-compliant call flow.)

Transport Modes: HTTP vs WebSocket

According to the MCP spec, there are two official transport mechanisms and the possibility of custom transports:

- **Standard Transports:** The current spec defines **stdio** and **streamable HTTP** as the official transports [19]. In practice:
- **Stdio:** The client can spawn the MCP server as a subprocess and communicate via its stdin/stdout. This is common for local tools (the client starts a tool server on the fly). If your use case includes local servers, ensure your client can launch a process and pipe JSON-RPC messages to it. (Testing stdio might involve running a lightweight test server binary and seeing that your client can initialize it and exchange messages.)

- **HTTP + SSE:** For remote or long-running servers, the client connects over HTTP. The spec replaced older HTTP+SSE designs with a streamlined **HTTP POST + optional Server-Sent Events (SSE)** model [20]. Your client should:

    - POST every JSON-RPC request to the server's endpoint (e.g. `http://host:port/mcp` or similar). The HTTP request should include `Content-Type: application/json` and an `Accept` header that includes `text/event-stream` (since the server might use SSE to stream results) [21].
    - Handle the response: If the server responds immediately with `Content-Type: application/json`, that means it's returning a single JSON-RPC response in the HTTP body (typical for non-streaming results). This comes with an HTTP 200 OK status and the JSON content – your client just reads the body and parses the JSON. If the server instead responds with `Content-Type: text/event-stream`, it's initiating an SSE stream [22]. In that case, your client should listen to the SSE stream and read events until the final JSON-RPC response arrives (the spec says the server will eventually send the JSON-RPC response as one of the SSE events, then close the stream) [23].

    - For testing basic functionality, you might use a server that sends a normal single response (many tool calls that complete quickly will do that). Just verify your client can handle both cases: immediate JSON vs. SSE. If you haven't implemented SSE yet, note that **clients must support both** event-stream and direct responses per spec [22]. It's an important part of being up-to-date. If your client currently only handles one, consider adding at least basic SSE handling (even if just using an EventSource or streaming fetch API under the hood).

- **WebSocket (Custom Transport):** You mentioned ensuring tests pass for WebSocket as well. **WebSocket is not officially part of the MCP 2025-06 spec**, but many developers (and hosting providers) have been experimenting with it as a custom transport. In fact, the community has discussed standardizing WebSocket support for MCP because of practical benefits in certain deployments (e.g. avoiding sticky HTTP sessions) [24]. Some projects have implemented WebSocket transport on their own (for example, a patch adding WebSocket support was merged into an MCP gateway project, and Anthropic's Python SDK has provisions for it [25]). All this to say: **WebSocket is considered a custom/experimental transport at this time**, but it's a recognized need. If your implementation already includes WebSocket support (or if you plan to add it), treat it as an **optional extension** that rides on top of MCP's JSON-RPC message format. The client logic for WebSockets would be:

- Open a WebSocket connection to the MCP server (some servers might offer a `/ws` endpoint or similar for this).
- Once connected, send the same JSON-RPC messages over the socket instead of HTTP POST. The initialization, tools/list, tools/call, etc., all use identical JSON payloads.
- Receive messages from the server over the socket (which could be responses or server-initiated notifications/requests). Your client needs to dispatch these just as it would over HTTP. Essentially, WebSocket is just a transport pipe; the MCP message sequences remain the same.
- For testing, ensure that if you connect via WebSocket, you can perform the initialization handshake and then do a `tools/list` and `tools/call` successfully, getting the expected JSON-RPC results. The outcome should mirror the HTTP case (e.g., you send `tools/call` and you receive a response with `result.content` etc., just delivered over the WebSocket channel).

Since WebSocket isn't an official part of the spec yet, make sure any WebSocket-related code in your client is **separated** or toggled by configuration so that it doesn't interfere with standard HTTP/SSE behavior. Other developers might not have WebSocket available, but if your client cleanly supports both, that's fine. It sounds like *"custom support can come later"* – which implies focus on HTTP now, and add WebSocket as a bonus when ready. As long as the HTTP path is solid, you meet the spec; WebSocket can be viewed as a custom extension for environments where it's needed.

Conclusion and Verification Checklist

By aligning your MCP client with the **2025-06-18 spec** and focusing on standard features, you ensure it's compatible with what other developers expect from an MCP integration. Here's a quick checklist to verify against your implementation:

- **Protocol Version:** Using latest version in initialize; no deprecated behaviors (e.g. no JSON-RPC batch) [3].

- **Initialization:** Implements the full handshake (initialize → gets capabilities → sends initialized) and negotiates capabilities including `tools` [7]. Does not assume any experimental extensions by default [2].
- **Tools Listing:** Supports sending `tools/list` and parsing the returned tool list (name, description, inputSchema, etc.) [12]. Prepares to handle pagination if `nextCursor` is set (even if just a basic loop or ability to fetch more).
- **Tools Calling:** Can send a `tools/call` with proper arguments matching the input schema [16]. Handles the response by checking for `result` vs `error`. If `result` with `content` is present, recognizes it as success (and maybe can retrieve the content text if needed). If `structuredContent` is present, optionally handle it or at least not break. Marks the call as successful (for tests, a 200/OK) when `isError` is false [17].
- **Error Cases:** Gracefully handles JSON-RPC error responses (logs or returns an error to the caller, without crashing). Not strictly part of a "happy path" spec compliance, but good practice.
- **HTTP Transport:** Implements the POST logic with appropriate headers, handles both immediate JSON responses and SSE streams [22]. Ensure your test environment can simulate or connect to an MCP server over HTTP; you might use a known MCP test server or a dummy server that echoes responses. Your client should consistently get a 200 HTTP status for valid requests and parse the JSON.
- **WebSocket Transport:** (If implemented) Able to perform the same actions over a persistent socket. If not implemented yet, that's acceptable (since it's not standard) – but plan for adding it if your environment requires it. For now, you might mark any WebSocket tests as experimental or to-be-done, unless you already have it working.
- **Testability:** Your client's functions (e.g. `listTools()` and `callTool(name, args)`) should be easy to invoke in a test script and should return Promises or values that the test can assert on. For instance, `listTools()` might return an array of tool info objects (which the test can verify length or contents of), and `callTool()` might return the tool result or a success indicator. Designing the interface this way (rather than burying logic in event handlers without exposure) makes it simple to write Node tests that call these methods and verify outcomes. If you haven't already, consider writing a small integration test that uses a real or mock MCP server to go through the motions – this will give you confidence that the implementation truly matches the spec in practice.

By double-checking each of these points, you can be confident your MCP client consumer is 100% up-to-date with the current spec and that it will pass any reasonable tests (both HTTP and WebSocket) for standard tool listing and invocation. This alignment with the official spec means your client will behave as expected in the broader MCP ecosystem, and any developer using or interacting with it will find it working with the "standard" features out-of-the-box [1] [26]. Good luck with your Node tests – they should be much easier now that the implementation is clean and spec-compliant!

Sources:

- Model Context Protocol – *Tools Specification (2025-06-18)* [12] [16] [17] (example structure of tools list and call in the latest spec)
- Model Context Protocol – *Base Protocol (Transports)* [19] [22] (official transports and HTTP request/response requirements)
- Model Context Protocol – *Lifecycle and Capabilities* [7] [2] (handshake exchange showing `tools` capability and note on experimental features)
- MCP Spec Changelog (2025-06-18) [3] (removal of JSON-RPC batching in latest spec)

- Community Discussion on WebSocket Transport [25] (WebSocket support is being explored as a non-standard extension, not in spec yet)

---

[1] [11] [12] [13] [14] [15] [16] [17] [18] Tools - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/server/tools

[2] [4] [5] [6] [7] [8] [9] [10] [26] Lifecycle - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/basic/lifecycle

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

[19] [20] [21] [22] [23] Transports - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/basic/transports

[24] [25] WebSocket Support for MCP + Proposal for an MCP Hosting WG : r/modelcontextprotocol
https://www.reddit.com/r/modelcontextprotocol/comments/1jin0iw/websocket_support_for_mcp_proposal_for_an_mcp/