ChatGPT

MCP Technical Knowledge Base for Claude Code Generation

Methodology

To build this knowledge base, we exhaustively reviewed **Anthropic's official Model Context Protocol (MCP) documentation** (latest revision 2025-06-18). We focused exclusively on authoritative sources – namely the MCP specification and user guides on the **modelcontextprotocol.io** site – to ensure 100% accuracy. Our process was as follows:

- **Scoping MCP Components:** We first identified all major MCP components and features that needed coverage. These include the base protocol (core architecture, message format, lifecycle, transports), **server-side features** (Resources, Prompts, Tools), **client-side features** (Sampling, Roots, Elicitation), and relevant utilities or patterns. We clarified sub-questions for each: e.g. *What is the purpose of each feature? How is it structured? How do clients/servers use it?*

- **Gathering Official Documentation:** Using the Anthropic MCP docs and linked MCP specification site, we collected every relevant page and sub-section. This included conceptual guides for each feature (to understand high-level usage) and the formal specification pages (for exact JSON structures, required fields, and normative requirements). All information is drawn **directly from Anthropic's MCP docs**, ensuring fidelity to the intended design. We cross-referenced the **MCP version and revision notes** to verify stability of features. For example, we noted that *Elicitation* is a **newly introduced feature** (as of the 2025-06-18 spec) and may evolve [1] [2], whereas older mechanisms like standalone SSE transport are **deprecated** [3]. We also confirmed if any features were marked *beta* or deprecated (none were explicitly beta-tagged; features are considered stable unless noted as new or deprecated in the changelog). The **Key Changes** log was reviewed to identify additions and removals between spec versions (e.g. JSON-RPC batching removal, introduction of structured outputs, new fields like `title`) [4] [5].

- **Credibility & Version Checks:** All content is based on the **latest stable MCP spec (2025-06-18)** [6]. We ensured each cited detail aligns with this version. Where relevant, we highlight version-specific notes (e.g. SSE deprecation in late 2024 [3], new elicitation in 2025 [7]). We treated the MCP spec itself as the source of truth, verifying that any code patterns or JSON examples we include match the official schema definitions [8] [9]. We also evaluated stability: for example, *Elicitation* is clearly flagged as an evolving feature, whereas *Resources, Prompts,* and *Tools* are well-established. Deprecations (like SSE) and mandated changes (like requiring version headers) are noted.

- **Synthesis and Comparison:** We carefully dissected each protocol section line-by-line, explaining the mechanics and including key code snippets from the docs. For every major feature, we describe its purpose, how it's declared/used in the JSON protocol, and any important best practices or security considerations. We also cross-referenced the **user's own MCP server configuration (MCPDesktop)** to ensure consistency. For instance, the user's repo defines an MCP server with a JSON config where

`mcpServers` are listed with a command to run the server [10]. We validated that this aligns with Anthropic's guidance for using `--mcp-config` to launch custom servers (it does – the official docs show a similar JSON structure for MCP servers with `command`, `args`, and optional `env` keys) [11] [12]. Throughout, we applied *chain-of-thought* reasoning and *self-reflection* to ensure no detail was missed – double-checking each protocol message and field in the spec and considering how a Claude-based assistant might utilize this knowledge (both in generating code and understanding context).

- **Advanced Reasoning Techniques:** At each stage, we employed advanced reasoning to identify any ambiguous areas or integration challenges:

- **Chain-of-Thought:** We traced the end-to-end flow of MCP interactions (initialization handshake, feature-specific request/response cycles, etc.) step by step [13].
- **Tree-of-Thought:** For complex features like *Tools* or *Sampling*, we explored multiple scenarios (e.g. different tool result content types, or various model selection strategies in sampling) to ensure comprehensive coverage.
- **Self-Critique:** We cross-checked whether each explanation would make sense to a developer or GPT model using the knowledge. If something seemed under-documented (e.g. how *progress reporting* or *cancellation* might work), we noted it as a potential gap for further research.
- **Comparative Analysis:** We contrasted similar features (e.g. *Resources vs. Tools* for providing context, or *Prompts vs. Elicitation* for guiding interactions) to highlight differences in control and usage [14]. We also compared the user's current implementation details with official recommendations to spot any discrepancies.
- **Hypothesis Testing:** In a few cases we hypothesized implementation approaches (e.g. how one might extend the user's MCPDesktop server with new Tools or Resources) and checked the docs to confirm viability. This helped in formulating recommendations for *enhancing MCP implementations* based on best practices (such as using proper error codes, implementing security checks, etc.).

All findings below are richly annotated with citations from the MCP documentation. Each section includes code or JSON examples from the spec and an explanation, providing Claude with both the **structured data** (exact schemas) and **human-readable context** needed to generate correct Claude Code instructions or assist in development.

Findings

Core Protocol Architecture and Lifecycle

MCP Overview: The **Model Context Protocol** is an open standard that defines how AI applications (LLM hosts) connect to external data sources and tools in a *unified* way [15]. At a high level, MCP introduces a client–server model on top of JSON-RPC for communicating between an AI **host application** (like Claude Desktop or an IDE plugin) and one or more **MCP servers** that provide contextual data or actions [16]. Think of it as providing Claude with "USB-C for AI" – a consistent interface to plug in various integrations [17].

Core Components: The architecture involves three primary roles [18] [19]:

- **Host** – the LLM application container (Claude Desktop, etc.) that manages connections. The host can run multiple **clients** and is responsible for user consent, policy enforcement, and aggregating context across servers [20].
- **Client** – a connector (running within the host) that maintains a **1:1 session with a specific server** [21]. Each client speaks MCP to its server, handling message routing and ensuring isolation between different server connections. In Claude Desktop, for example, each configured MCP server (like a filesystem tool server, a GitHub integration server, etc.) would have its own client instance inside Claude.
- **Server** – a standalone program or process that offers a set of capabilities to the client (exposing context data, prompts, or tools). Servers are lightweight and focused (e.g., a "desktop automation" server, a "database query" server) and can be local or remote [19].

These pieces interact over a **stateful JSON-RPC connection** managed by a **transport layer** (stdio for local processes or HTTP for remote) [22] [23]. The design principle is that *servers* should be very easy to write (the host and client handle orchestration, multi-server coordination, security, etc.) [24] [25]. Servers operate independently and cannot see each other's data or the whole conversation unless explicitly given context – this isolation is by design for security [25].

Connection Lifecycle: Establishing an MCP session involves a **3-step handshake** [13]:

1. **Initialize:** The client initiates by sending an `initialize` request to the server, including its supported MCP protocol version and a declaration of its capabilities/features [13]. For example, the client will specify if it supports **sampling**, **roots**, or **elicitation** (client-side features discussed later) in this message. The protocol version negotiation is important – in HTTP transports, the chosen version is also conveyed via a header (`MCP-Protocol-Version`) for all subsequent messages [26]. The current spec version uses date-based IDs (e.g. `"2025-06-18"`).
2. **Server Responds:** The server replies with its own capabilities and version. It will list which **server-side features** it implements (like **resources**, **prompts**, **tools**, possibly with sub-options) [27] [28]. Both sides thus agree on a common protocol version and know what features each supports (this is the *capability negotiation*). If a feature is not declared by one side, it won't be used during the session.
3. **Initialized Acknowledgment:** Finally, the client sends an `initialized` **notification** (no response expected) to confirm it is ready [13]. At this point, normal message exchange can begin.

This handshake ensures both parties understand which MCP extensions are in play. For instance, a server must advertise the `resources` **capability** if it plans to offer file or data resources; if it doesn't, the client won't try to call `resources/list` on it [27] [29]. Similarly, a client that doesn't support sampling simply wouldn't declare `sampling` in its initialize request, so the server knows not to attempt `sampling/createMessage` calls [30] [31].

JSON-RPC Messaging: MCP inherits JSON-RPC 2.0 as the underlying message format for all interactions [32] [33]. There are three fundamental message types:

- **Request:** a call expecting a response (includes a unique `id`, a `method` string like `"resources/read"` or `"tools/call"`, and optional `params`) [34] [35]. Either side (client or server) can issue

requests. For example, the client requests `tools/list`, while the server might request `sampling/createMessage` (when it needs an LLM completion) – MCP is **bidirectional** in that both client and server can initiate requests.

- **Response:** the reply to a request, containing either a `result` on success or an `error` on failure (and mirrors the same `id`) [36] [37]. Only one of `result` or `error` is present [38]. Results can be any JSON object defined by the protocol. Errors include a code, message, and optional data [39].
- **Notification:** a one-way message that has a `method` and `params` but **no** `id` (no response will be sent) [40] [41]. Notifications are used for events or updates – e.g., a server might send `notifications/resources/updated` to inform the client that a resource's content changed [42] [43]. Either side can notify the other.

MCP specifies that request IDs must not be re-used within a session and must not be null [44]. It also lists **standard JSON-RPC error codes** like `-32601` (Method not found), `-32602` (Invalid params), etc., and allows custom error codes (typically in the -32000 range for application-specific errors) [45] [46]. For example, a server should return error code `-32002` if a requested resource isn't found [47] [48].

Transports: The protocol is transport-agnostic but defines two **standard transport mechanisms** [22] [49]:

- **Standard I/O (stdio):** The server process reads JSON-RPC messages from stdin and writes responses to stdout. This is ideal for local servers (fast and simple) [23] [50]. Claude Desktop uses this for local plugin processes – the config simply launches the server process (e.g., a Node.js script) and pipes data. In the user's configuration, for instance, `"command": "node", "args": ["server/index.js"]` will result in a stdio-based connection [51].
- **Streamable HTTP:** A newer transport where the client posts JSON-RPC requests to the server's HTTP endpoint, and the server can send responses either as a single JSON or as a streaming SSE (Server-Sent Events) if multiple messages need to be pushed [52] [53]. Concurrent clients and remote networking are supported, with features like session resumption. This mode is useful for cloud-based servers or services with their own HTTP interface.

Notably, **pure SSE as a standalone transport is deprecated** as of the November 2024 revision [3]. Previously, some implementations allowed an SSE channel for server->client messages with client->server via POST; but now SSE is subsumed into the streamable HTTP approach. The spec advises using TLS for any remote transport and cautioning against DNS rebinding attacks (for local servers, binding only to localhost) [54] [55]. The takeaway: *Local MCP servers typically use stdio*, remote ones use HTTP+SSE streams with proper authentication. Both still speak the same JSON-RPC on the wire [56].

Capability Negotiation: A crucial aspect of MCP is that *features are opt-in.* During initialization, each side advertises which features it supports and relevant options [57] [58]. For example:

- A server that supports **Resources, Prompts, and Tools** will include those keys under its `capabilities` object in the initialize response [27] [59]. It may also indicate sub-capabilities, like `{"resources": {"subscribe": true, "listChanged": true}}` if it can do resource subscriptions and update notifications [29] [60].
- A client that supports **Sampling and Roots** would send `{"capabilities": { "sampling": {}, "roots": {"listChanged": true} }}` in its initialize request [30] [61]. The empty `{}` or presence of a flag like `listChanged` signals what it can handle.

Both parties **must respect these declarations** [62] [63]. If a server doesn't declare `tools`, the client won't attempt any `tools/list` or `tools/call` RPCs. If a client doesn't declare `elicitation`, the server shouldn't try to use that feature. This negotiation makes MCP *extensible* – new features can be added in future spec versions, and older implementations will simply not advertise or use unsupported ones. It also enforces security: e.g., if a user's Claude client doesn't enable sampling, the server cannot force the model to generate text (preventing unapproved LLM calls) [30] [31].

Security and Trust Model: Because MCP can expose powerful operations (file access, code execution) and data flows, **security is paramount**. The spec lays out key principles [64] [65]:

- **User Consent & Control:** The user must always be in control of what data is shared and what actions are taken [66]. Host apps (like Claude Desktop) are expected to present clear UI for the user to approve resource access or tool usage. *For example, Claude Desktop requires the user to explicitly select which Resource files to include in context* [67], and it prompts the user to approve each Tool invocation.
- **Data Privacy:** Hosts shouldn't send user data to servers without consent [68]. E.g., the Claude client won't automatically share all open files – only those the user picks or that are within an approved root.
- **Tool Safety:** Tools are effectively remote code execution. The spec warns that tool descriptions/annotations (coming from the server) **cannot be blindly trusted** and that the host should get user permission for each invocation [69] [70]. Claude Code addresses this by requiring `--allowedTools` flags; by default, **MCP tools will not run unless explicitly permitted by the user** [71] [72].
- **LLM Query Controls:** For server-initiated LLM calls (Sampling), the user must explicitly approve the prompt that will be sent and the response that comes back [73] [74]. The protocol deliberately limits server visibility: servers ask for a completion but *do not automatically see the entire user conversation* – they only see what the client returns in the `sampling/createMessage` result [75] [76].

In practice, Anthropic's Claude implementation enforces these via UI prompts and flags. For instance, as shown in the Claude Code SDK docs, when using MCP servers one must list allowed tool names (prefixed with `mcp__server__tool`) or the model cannot invoke them [71] [72]. This implements the "human in the loop" requirement from the spec.

MCP Feature Primitives (Summary of Components)

MCP defines several **core feature primitives** that servers can provide and clients can consume. These map to the high-level capabilities we mentioned in negotiation. In summary, on the **server side** we have:

- **Resources** – exposing data (files, database info, etc.) as contextual content for the LLM [77] [78].
- **Prompts** – providing pre-made prompt templates or multi-step workflows that users can invoke [79] [78].
- **Tools** – defining actions or functions that the LLM can request to execute (with user approval) [80] [81].

On the **client side**, we have features that empower the server to initiate certain behaviors via the client:

- **Sampling** – letting servers ask the client to get an LLM completion (the client controls the actual model call) [82] [83].

- **Roots** – the client informing the server about what root URIs (e.g., directories or scopes) the server should operate within [84] [85].
- **Elicitation** – the server asking the client to prompt the user for additional info and return that input (an interactive data collection) [86] [87].

These are the "MCP components" we will dissect. It's useful to note **who controls each primitive** in typical usage, as it affects how they're used:

Primitive	Control	Description	Example Usage
Prompts	*User-controlled*	Pre-defined templates or instructions that guide LLM interactions. Exposed by servers, but **triggered explicitly by the user** (e.g. selecting from UI). [14]	User picks a "Summarize Code" prompt from a menu (server provides the prompt content). [14]
Resources	*Application-controlled*	Contextual data sources attached by the **client application**. The server lists available data, but the **client/user decides if and when to use it**. [88] [89]	Claude Desktop shows a list of files or logs (resources); user selects which to feed into context.
Tools	*Model-controlled*	Executable actions invoked by the **LLM** (when the LLM "decides" to use a tool). **Requires human approval** in the loop despite model initiation. [88] [90]	LLM chooses a `search_web` tool during a chat to look something up; user is prompted to allow it.

[14] shows this control hierarchy as given in the docs. This framework informs how each feature is typically implemented: e.g., **Prompts** are offered to the user in the UI (the LLM doesn't auto-run prompts on its own), whereas **Tools** are often triggered by the LLM (with the UI just confirming/denying). **Resources** lie in between – the client might automatically include some, but often requires user selection for safety [89]. Next, we deep-dive into each feature.

Resources (Server Feature)

What are Resources? *Resources* in MCP are a mechanism for servers to expose pieces of data or content that can be used as additional context for the LLM [91]. Think of them as **read-only data files or records** the LLM could read. Examples include: files from the user's filesystem, database query results, system logs, images or PDFs, etc. [92]. Each resource is identified by a **URI** and can be of two types: **text** (UTF-8 content) or **binary** (base64-encoded data) [93] [94].

Resources are **application-controlled** meaning the client app (and ultimately the user) decides how they are surfaced. For instance, Claude Desktop will not send all listed resources to the LLM automatically; it requires the user to explicitly attach a resource (e.g. by clicking a file) before it's included [67]. This is a safety measure to avoid overwhelming or unintentionally leaking data. If an integration needs to automatically feed data in without user selection, the MCP docs suggest using a model-controlled primitive (like a Tool) instead [95].

URI Format: Resource identifiers are URIs that the server defines. They follow a `[protocol]://[host]/[path]` format [96]. Common schemes can be `file://` for local files, `https://` for web resources, `postgres://` for a database entry, or custom schemes like `screen://` for a screenshot feed [97]. The **server** has flexibility to define custom URI schemes suitable for its domain [98], as long as they conform to URI standards.

For example, a server listing a local project's files might return URIs like:

```
file:///home/user/project/src/app.py
```

or a server offering an API might use:

```
api://weather/current?city=London
```

(With `api://` being custom-defined by that server.)

Capability Declaration: A server that supports resources must declare the `"resources"` capability in its initialization response [28]. This can optionally include:

```
"resources": {
  "subscribe": true,
  "listChanged": true
}
```

- `subscribe` – indicates the server can handle clients subscribing to resource update notifications (for real-time content changes) [60]. - `listChanged` – indicates the server will send a notification if the overall list of resources available changes (e.g., new file added) [60].

These flags are optional. A server might support none, one, or both. The spec provides examples for each case (empty object for neither, object with one flag, etc.) [99] [100]. Claude Desktop's client will use this info: if `subscribe` isn't supported, it won't attempt to subscribe to resources; if `listChanged` = true, it will expect potential `notifications/resources/list_changed` events.

Listing Resources: The client can ask the server for the list of available resources via the `resources/list` request [101]. This returns a list of resource descriptors. The request supports **pagination** – a `cursor` parameter can be provided to page through results [102] [103] (useful if a server has thousands of resources, though many simple cases just return all). A typical response looks like:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "resources": [
```

```
      {
        "uri": "file:///project/src/main.rs",
        "name": "main.rs",
        "title": "Rust Software Application Main File",
        "description": "Primary application entry point",
        "mimeType": "text/x-rust"
      }
    ],
    "nextCursor": "next-page-cursor"
  }
}
```

[104] This shows one resource: the URI and a set of metadata. Let's break down the **Resource object schema** [105]:

- `uri` (string) – *Unique identifier* of the resource [106]. This is what clients will reference to read the resource.
- `name` (string) – A short name for display (often a filename or identifier) [107].
- `title` (string, optional) – A human-friendly title for UI display [107]. (In the example, "Rust Software Application Main File" is a title that's more descriptive than the filename). The introduction of a separate `title` field in the 2025 spec allows `name` to be a machine identifier while `title` is for display [5].
- `description` (string, optional) – Longer description of the resource if available [107].
- `mimeType` (string, optional) – MIME type of the content, if known (e.g. `text/plain`, `image/png`) [107].
- `size` (number, optional) – Size in bytes (servers may include this for files, etc.) [108].
- (No content is included here – content is retrieved via `read`).

The example above also included `"nextCursor"` in the result which is used if pagination is in effect (client would pass that in a subsequent `resources/list` call to get the next page) [109].

Reading a Resource: To actually get the content of a resource, the client sends `resources/read` with the `uri` of interest [110] [111]. The server responds with a `contents` array (to allow reading multiple resources in one call, though often it's just one) [112] [113]. Each item in `contents` will have either a `text` field (if it's a text resource) or a `blob` field (if binary), along with the `uri` and optional `mimeType` again [114] [115]. Example:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "contents": [
      {
        "uri": "file:///project/src/main.rs",
        "mimeType": "text/x-rust",
        "text": "fn main() {\n    println!(\"Hello world!\");\n}"
```

```
      }
    ]
  }
}
```

[116] [117] If the file were binary (say an image), the server would base64-encode it and return `"blob": "<data>"` instead of text [94] [118]. The client (Claude) will then typically embed this content into the conversation (often truncated or summarized if large). Servers *may* allow reading directories or multiple files in one go – the spec notes that a `resources/read` could return multiple content entries if you read a directory URI, for instance [119].

Resource Templates: Some resources are dynamic – instead of enumerating every possible item (e.g. every possible database query), a server can advertise **URI templates** that clients can fill in. MCP supports a `resources/templates/list` method to list such templates [120] [121]. Each template has a `uriTemplate` (following RFC 6570 syntax) and a name/description like a normal resource, possibly with a fixed mimeType if known [122] [123]. For example:

```
{
  "uriTemplate": "file:///{path}",
  "name": "Project Files",
  "title": "Project Files",
  "description": "Access files in the project directory",
  "mimeType": "application/octet-stream"
}
```

[124] This indicates the server accepts URIs of the form `file:///whatever` under a certain base path. The client could use this to let a user input a path or to auto-complete file paths (the spec even mentions templates can be tied into a *completion API* for suggestions) [125] [126]. In practice, templates are less commonly used in simple servers – one might just list directories and files as resources instead – but it's powerful for parametric resources.

Resource Change Notifications: If a server's data changes, it can inform the client:

- If new resources become available or removed, and `listChanged: true` was set, the server sends a **notification**: `notifications/resources/list_changed` [127] [128]. This tells the client that it may want to refresh the resource list (by calling `list` again). The notification has no params – it simply signals a change.
- If a specific resource's content updates and the client had subscribed, the server sends `notifications/resources/updated` with the `uri` (and optionally a `title` if it changed or other small metadata) [42] [43]. The client upon receiving this might auto-refresh that content or prompt the user. Subscriptions are set up via `resources/subscribe` and removed with `resources/unsubscribe` calls [129] [130]. The `subscribe` request includes the `uri` to watch; thereafter, whenever that resource changes, an `updated` notification will fire.

The **MCPDesktop** server in the user's repo, for instance, could leverage this if it wanted to notify Claude when (say) a log file grows or a new file appears in a directory. It would declare `subscribe: true` and use `notifications/resources/updated` accordingly. (If the user's current implementation doesn't yet, it might be a potential enhancement.)

Example Implementation: The official docs provide code snippets in TypeScript for implementing resources. For instance, a simple server might do:

```
server.setRequestHandler(ListResourcesRequestSchema, async () => {
  return {
    resources: [
      { uri: "file:///logs/app.log", name: "Application Logs", mimeType: "text/
plain" }
    ]
  };
});
server.setRequestHandler(ReadResourceRequestSchema, async (req) => {
  const uri = req.params.uri;
  if (uri === "file:///logs/app.log") {
    const logContents = await readLogFile();
    return {
      contents: [{ uri, mimeType: "text/plain", text: logContents }]
    };
  }
  throw new Error("Resource not found");
});
```

[131] [132] This illustrates how a server declares the capability (`resources: {}` in its constructor options, not shown here) and then handles list and read calls. The MCP SDK provides request schema definitions (like `ListResourcesRequestSchema`) to ensure type safety. The pattern is consistent: for each method, call `server.setRequestHandler(schema, handlerFunction)` to implement it. Our knowledge base includes these patterns so Claude can mirror them when generating code. For example, if asked to add a new resource in MCPDesktop, Claude should follow this structure.

Best Practices: When implementing resources, the spec recommends: - Use clear names and URIs, and provide helpful descriptions and MIME types [133]. - Support **resource templates** for dynamic data if appropriate (like giving a template for "logs of last N days") [134]. - Implement subscriptions for frequently changing data, rather than forcing polling [135]. - Consider pagination if listing a huge number of resources [136]. - **Error handling:** return clear errors (e.g., "Resource not found") and appropriate JSON-RPC codes (like `-32002`) [47] [48]. - **Security:** Validate all resource URIs server-side (don't allow path traversal like `..`), enforce access controls (only expose what the user should access), and possibly sanitize content (especially if binary) [137]. The server should also consider rate limiting resource reads and auditing access to sensitive data [138].

Claude or any GPT-based assistant using this knowledge should be mindful of these points. For instance, if generating an MCP server stub, it should include checks for valid file paths and not just serve anything requested.

Prompts (Server Feature)

What are Prompts? *Prompts* in MCP are pre-defined **prompt templates or workflows** that a server can provide to the client [139]. They allow a server to package up an interaction pattern with the LLM – such as a specific instruction or even a multi-turn dialogue outline – which the **user can easily invoke** without typing it from scratch. In simpler terms, they are like "canned queries" or **slash commands** offered by the integration [140] [141]. For example, a coding assistant server might have a prompt called `"analyze-code"` that, when invoked, asks the LLM to analyze provided code for improvements [142] [143].

Prompts are **user-controlled**: the expectation is the user chooses to run a prompt. They might be exposed in the UI as buttons, menu items, or `/commands` that the user clicks or types [144] [145]. The model itself doesn't autonomously trigger prompts. This is an important distinction from Tools – prompts are more like user shortcuts, whereas tools are actions the model might decide to use.

Structure of a Prompt: Each prompt is defined by the server with a few fields:

```
{
  "name": "analyze-code",
  "title": "Analyze Code for Improvements",
  "description": "Analyze code for potential improvements",
  "arguments": [
    {
      "name": "language",
      "description": "Programming language",
      "required": true
    }
  ]
}
```

[146] [147] In general, the **Prompt** object includes: - `name` – a unique identifier (used programmatically to request it) [148]. - `title` – a human-friendly title for UI (this field was introduced in the spec as separate from `name` to allow nicer display names [5]; e.g. title might be `"Analyze Code for Improvements"` while `name` is `"analyze-code"`). - `description` – a description of what the prompt does [148]. - `arguments` – an optional list of arguments that can parameterize the prompt [149]. Each argument has a name, description, and whether it's required. (No types here – these arguments are just placeholders for text usually.)

For example, a `"git-commit"` prompt might take a `changes` argument (the diff or change description) that the LLM should turn into a commit message [150] [151].

Declaring Capability: A server that offers prompts declares the `"prompts"` capability. This can include a flag `listChanged` to indicate it will notify of prompt list changes [27] [152]. Usually, `prompts: { "listChanged": true }` if the server might dynamically add/remove prompts at runtime. If not, `prompts: {}` is fine.

Listing Prompts: The client gets the list via `prompts/list` request [102]. The response contains a `prompts` array, each entry with the structure above. Here's an example response snippet:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "prompts": [
      {
        "name": "code_review",
        "title": "Request Code Review",
        "description": "Asks the LLM to analyze code quality and suggest
improvements",
        "arguments": [
          { "name": "code", "description": "The code to review", "required":
true }
        ]
      }
    ]
  }
}
```

[153] [154] Note the presence of `title` and a clear description. The client (Claude's UI) might display these as a list of actions the user can pick (for instance, in Desktop you might have a toolbar of available prompts, or as context menu actions).

The `prompts/list` method supports **pagination** similar to resources, via a `cursor` param for servers with many prompts [103]. In practice, most servers have only a handful of prompts, so pagination is seldom needed.

Getting a Prompt: Once the user selects a prompt to use (or triggers it somehow), the client will call `prompts/get` with the prompt `name` and any `arguments` filled in [155] [156]. The server then generates the actual prompt content (usually a series of one or more message objects to send into the conversation). The response includes: - An optional `description` (the prompt's description again or a modified one), - A `messages` array: structured like chat messages (`role` and `content`) that should be inserted into the Claude conversation [157] [158].

For example, if the user selected `"analyze-code"` with argument `language: "python"`, the server might respond:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "description": "Analyze Python code for potential improvements",
    "messages": [
      {
        "role": "user",
        "content": {
          "type": "text",
          "text": "Please analyze the following Python code for potential
improvements:\n\n```python\n def calculate_sum(numbers):\n    total = 0\n    for
num in numbers:\n        total = total + num\n    return total\n\n result =
calculate_sum([1, 2, 3, 4, 5])\n print(result)\n```"
        }
      }
    ]
  }
}
```

[159] [160] Here the server constructed a user message asking for analysis of a code snippet (likely the snippet could be provided via the arguments, or maybe the server fetched it from a resource). The client will take these `messages` and insert them into the conversation with Claude, as if the user had written them. The result is that Claude receives the prompt and can then generate a completion.

Dynamic and Multi-Step Prompts: Prompts can be more complex than a single message. They support **embedding resource references** and **multi-turn workflows**: - A prompt can include content of type `"resource"` where instead of raw text you provide a resource's URI and maybe some excerpt. In the spec example, a prompt's messages included content like:

```
{ "role": "user", "content": {
      "type": "resource",
      "resource": {
         "uri": "logs://recent?timeframe=1h",
         "text": "[2024-03-14 15:32:11] ERROR: Connection timeout in network.py:
127\n..."
       }
} }
```

and another with `"uri": "file:///path/to/code.py", "text": "def connect_to_service(timeout=30): ..."` [161] [162]. This means the prompt is including the content of those URIs directly. The `type: "resource"` content tells the client that this chunk is from a resource (so the client/LLM knows it's reference material) [163]. Essentially, the server read those resources and inlined them into the prompt. - Prompts can orchestrate *multiple turns*. The spec shows an example of a multi-turn debug workflow prompt, where the `getMessages` function returns an array of messages simulating a

mini conversation [164] [165]. For instance, it first injects a user message "Here's an error I'm seeing...", then an assistant message "I'll help analyze this, what have you tried?", then another user message "I tried X...". This sequence guides the AI through a logical dialogue. When the client inserts these, Claude will effectively see a conversation that primes it for a particular task.

These advanced prompts let servers create guided interactions (like a troubleshooting wizard). In JSON terms, the `prompts/get` result can contain multiple messages with alternating roles. The client will append them all into the chat history in order.

Using Prompts (Claude Desktop): In practice, when a user triggers a prompt, Claude Desktop calls `prompts/get`, receives the messages, and then *immediately uses them as the next user input to the model*. The user can usually review the generated prompt (ensuring they are okay with what will be sent). This ties into the idea that prompts are user-initiated – the user sees what will be asked of the LLM.

Prompt Updates: If the server's set of prompts changes (maybe new prompt added via config or an update), and it set `listChanged:true`, it should notify the client with `notifications/prompts/list_changed` [166]. The client can then re-fetch the list. This is analogous to resources and tools list changes.

Example Implementation: A code example from the spec illustrates how to implement prompt listing and retrieval:

```
const PROMPTS = {
  "git-commit": { name: "git-commit", description: "Generate a Git commit
message", arguments: [ { name: "changes", description: "Git diff", required:
true } ] },
  "explain-code": { name: "explain-code", description: "Explain how code
works", arguments: [ { name: "code", description: "Code to explain", required:
true }, { name: "language", description: "Programming language", required:
false } ] }
};

server = new Server({ name: "example-prompts-server", version: "1.0.0" }, {
capabilities: { prompts: {} } });

server.setRequestHandler(ListPromptsRequestSchema, async () => {
  return { prompts: Object.values(PROMPTS) };
});
server.setRequestHandler(GetPromptRequestSchema, async (request) => {
  const prompt = PROMPTS[request.params.name];
  if (!prompt) throw new Error(`Prompt not found: ${request.params.name}`);
  if (request.params.name === "git-commit") {
    return { messages: [
        { role: "user", content: { type: "text", text: `Generate a concise
commit message for these changes:\n\n${request.params.arguments?.changes}` } }
      ]
```

```
    };
  }
  if (request.params.name === "explain-code") {
    const lang = request.params.arguments?.language || "Unknown";
    return { messages: [
        { role: "user", content: { type: "text", text: `Explain how this ${lang} code works:\n\n${request.params.arguments?.code}` } }
      ]
    };
  }
  throw new Error("Prompt implementation not found");
});
```

[167] [168] [169] [170] This code: - Stores prompt definitions in a dictionary. - When `prompts/list` comes in, returns all prompt objects. - When `prompts/get` comes, finds the right prompt and constructs appropriate messages, possibly using the provided arguments (`changes` or `code`) in those messages.

From the knowledge base perspective, when Claude is asked to implement a new prompt, it should follow this pattern: define the prompt metadata, add it to the list handler, and implement the get logic to produce the message(s).

Best Practices: For prompts, recommended practices include: - Use **clear names** and **detailed descriptions** so users understand what the prompt does [171]. - Validate required arguments (don't just assume they're there) and handle missing ones gracefully [172]. - If prompts might change over time or need improvements, consider versioning them or at least documenting changes. - **UI integration:** The server should choose prompt titles and descriptions that make sense in a UI context (since prompts will appear as clickable actions) [173]. E.g., a title case, succinct title, and a friendly description. - **Security:** Similar to general principles – sanitize any user-provided arguments when injecting into prompt text to avoid prompt injection issues [174]. Also, don't inadvertently include sensitive data in a prompt without user consent. The spec reminds to validate all inputs and consider prompt injection risks (the LLM could be influenced if, say, an argument contains malicious instructions) [175].

Given that prompts are ultimately instructions to the LLM, a poorly constructed prompt could cause the LLM to do unwanted things. So a server author should carefully craft prompt templates and perhaps test them with various inputs (the spec even suggests testing prompts with different inputs as a best practice) [176].

Tools (Server Feature)

What are Tools? *Tools* are one of the most powerful MCP features – they let the server expose **operations or actions** that the LLM can invoke during a conversation [177]. For example, a server might provide a `search_web` tool to query the internet, a `write_file` tool to create a file, or a `send_email` tool to dispatch an email. Each tool is basically a function/API that the server will execute on behalf of the LLM, and return the result back as part of the conversation.

Tools are **model-controlled** in the sense that the LLM (Claude) decides when and how to use them, given the option [178]. However, critically, **a human is kept in the loop** – typically, the client will ask the user for permission each time the model tries to call a tool [179] [180]. This ensures safety, since tools can perform actions in the real world.

If Resources are about providing passive data, Tools are about taking actions (potentially changing state or retrieving external info on the fly). They can range from "safe" read-only tools to potentially destructive operations (deleting a file, for instance), which is why the protocol has provisions to annotate tools with hints like `readOnly` or `destructive` (more on that shortly).

Tool Definition: A tool is defined by the server with a structure like:

```
{
  "name": "calculate_sum",
  "title": "Calculate Sum",
  "description": "Add two numbers together",
  "inputSchema": {
    "type": "object",
    "properties": {
      "a": { "type": "number" },
      "b": { "type": "number" }
    },
    "required": ["a","b"]
  },
  "outputSchema": { ... },        (optional)
  "annotations": {
    "readOnlyHint": true,
    "destructiveHint": false,
    "openWorldHint": false
  }
}
```

[181] [182] [183] [184] Key fields: - `name` – unique ID for the tool (used in RPC calls) [185]. - `title` – human-friendly name for UI display [186] [187]. - `description` – explains what the tool does, to help the model/user decide to use it [185]. - `inputSchema` – a **JSON Schema** object defining the expected parameters for the tool [188]. This is critical: it tells the model what arguments it can/must supply. (In the example: two numbers `a` and `b`). - `outputSchema` – optional JSON Schema for the tool's output structure [189] [190]. This is mainly to help validate and structure results, especially if returning JSON (structuredContent). - `annotations` – optional hints about the tool's nature (explained below) [191] [184].

The `inputSchema` allows complex nested objects, arrays, etc. It uses standard JSON Schema syntax. For instance, a tool might have an input with an object containing fields or an array of items, etc. The `outputSchema` if provided means the server promises to return results in that format, and the client can validate it [192] [193]. This was added in the latest spec to encourage structured outputs (making it easier for code or the LLM to parse the result) [4] [194].

Tool Capability: A server indicates it has tools by declaring the `"tools"` capability (and it may set `"listChanged": true` if it can add/remove tools dynamically) [195] [196]. The handshake tells the client "I have these tools available".

Listing Tools: The client requests `tools/list` to retrieve all tool definitions [197]. The response includes a `tools` array with each tool's spec as above. Example snippet:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "tools": [
      {
        "name": "get_weather",
        "title": "Weather Information Provider",
        "description": "Get current weather information for a location",
        "inputSchema": {
          "type": "object",
          "properties": {
            "location": {
              "type": "string",
              "description": "City name or zip code"
            }
          },
          "required": ["location"]
        }
        // (no outputSchema in this example, meaning output is freeform)
      }
    ]
  }
}
```

[198] [199] Now the client (Claude) knows the tool exists and how to call it (its name and what arguments it needs). Claude's prompt (the system prompt) typically includes a formatted list of tools it can use and their descriptions, so the model is aware of them.

Calling a Tool: When the LLM decides to use a tool, it will output a special action (in the Claude API, this is an "Assistant scratchpad" where it says something like `<<callTool>>`). The Claude client will intercept that and issue a `tools/call` request to the server with the specified `name` and `arguments` (which the model provided) [200] [201]. For example:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "tools/call",
```

```
  "params": {
    "name": "get_weather",
    "arguments": { "location": "New York" }
  }
}
```

[202] [203] The server receives this, executes the appropriate action (e.g., calls a weather API), then returns a result. The *result format for tools* is important:

The server's response to `tools/call` includes either a `result.content` (for unstructured output) or a `result.structuredContent` or both:

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

[204] [205] In this example, the tool returned a simple text message with the weather info. The `content` field is an array of **content blocks**, similar to how the assistant might normally respond. Each block can be text, image, etc. (We'll detail content types in a moment.) The `isError` flag is false, meaning this is a normal successful result [206] [207].

If the tool had an `outputSchema` and the server produced structured data, it could also include a `structuredContent` object with the actual JSON data (and possibly also return it as a string in the content for backward compatibility) [208] [192]. The spec's weather example demonstrates this: after calling `get_weather_data`, it returns:

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

[209] [210] so the LLM sees the textual form, but a client or developer can parse the JSON from `structuredContent` too.

Tool Result Content Types: Tools can return various content types: - **Text:** the simplest – a piece of text. (`{ "type": "text", "text": "...result..." }`) [211] [212]. - **Image:** a base64 image with a mimeType (`{ "type": "image", "data": "<b64>", "mimeType": "image/png" }`) [213] [214]. - **Audio:** similar to image but type "audio" (`data` + `mimeType`) [215]. - **Resource Link:** a special content type where the tool doesn't return the data directly, but a reference (URI) to a resource that was perhaps created

or is relevant [216] [217]. It contains `type: "resource_link"`, the `uri` and usual resource fields like name, description, and annotations. This is useful if, say, a tool generates a file – instead of dumping the entire file content, it can return a link so the client/user can decide to open or read it. The spec notes that resource links might not appear in the normal `resources/list` (they could be ad-hoc) [218]. - **Embedded Resource:** essentially inlining a resource's content fully (`{ "type": "resource", "resource": { ... } }`) [219] [220]. This is akin to what we saw with prompts embedding a resource, but a tool could also embed a resource it fetched (maybe to provide context with the result). If a tool does this, the server should be sure it supports the `resources` capability (likely it does if it's returning resources) [219] [220].

All these content forms can include **annotations** – metadata hints about `audience` (is this meant for user vs assistant consumption), `priority`, `lastModified` if applicable [221] [222]. For example, an image content might have `"annotations": { "audience": ["user"], "priority": 0.9 }` meaning it's primarily for the user to see and quite important [223]. These use the same annotation schema as resources [221] [224].

Annotations on Tools: The `annotations` field in a tool definition can include: - `readOnlyHint` (boolean) – true if the tool does not modify state (safe to run without side-effects) [187] [225]. - `destructiveHint` (boolean) – true if the tool *may* perform destructive changes (only applicable if readOnlyHint is false) [226] [227]. - `idempotentHint` (boolean) – true if calling the tool with the same args multiple times has no additional effect (beyond the first call) [228] [229]. - `openWorldHint` (boolean) – true if the tool interacts with external systems or the "open world" (like making web requests), as opposed to being self-contained [230] [225]. - `title` (string) – as mentioned, a display name for UI.

These are **hints only** – not enforced by the protocol, but useful for the client UI or policy. For instance, Claude's UI could highlight destructive tools or require extra confirmation. The spec cautions that these annotations should not be blindly trusted for security decisions (e.g., a malicious server could mark a dangerous tool as readOnly) [231]. However, they are still useful cues. The user's MCPDesktop server could benefit from adding such annotations to its tools (e.g., mark file deletion as destructive, mark read operations as readOnly) to integrate smoothly with any UI that uses them.

Tool Invocation Flow in Claude: In Claude's context (Claude Code or Desktop), when the model wants to use a tool: 1. It outputs a draft action (not directly shown to user) calling the tool. 2. The client intercepts and (usually) asks the user "The assistant wants to use tool X with these arguments, allow?" unless auto-approved. 3. If allowed, client sends `tools/call` to server, gets the result. 4. The result content is then inserted into the conversation (often as an assistant message with some prefix like "Tool output:"). 5. The conversation continues, with Claude hopefully using that information.

From a development perspective, a server developer writes the actual function for each tool. In the user's MCPDesktop codebase, for instance, they have tools like `disk-analysis`, `file-operations` etc. Each would correspond to a set of tool definitions and their implementations. Ensuring those align to MCP's expected input/output format is vital. (If the user's current config differs, it should be updated to match the spec's JSON structure).

Example Implementation: A basic tool example from docs:

```
server.setRequestHandler(ListToolsRequestSchema, async () => {
  return { tools: [ {
      name: "calculate_sum",
      description: "Add two numbers together",
      inputSchema: {
        type: "object",
        properties: { a: { type: "number" }, b: { type: "number" } },
        required: ["a","b"]
      },
      annotations: { title: "Calculate Sum", readOnlyHint: true, openWorldHint:
false }
  } ] };
});
server.setRequestHandler(CallToolRequestSchema, async (req) => {
  if (req.params.name === "calculate_sum") {
    const { a, b } = req.params.arguments;
    return { content: [ { type: "text", text: String(a + b) } ] };
  }
  throw new Error("Tool not found");
});
```

232 233 234 235 This registers one tool and handles execution. Real servers will have many tools and likely a more dynamic lookup, but conceptually it's the same. Note: The result here doesn't set `isError` – by default if not present, it's assumed false. If an error happened during tool execution (say an API call failed), best practice is to catch it and return an `isError: true` along with an error message in content 236 237 . E.g.:

```
try {
   // ...perform operation
   return { content: [ { type: "text", text: `Operation successful` } ] };
} catch (err) {
   return { isError: true, content: [ { type: "text", text: `Error: $
{err.message}` } ] };
}
```

236 238 This way, the LLM sees an error message and can decide how to handle it (maybe apologize or ask the user for different input), instead of the whole request failing at JSON-RPC level. The spec explicitly says *tool execution errors should generally be reported in the result, not as protocol errors*, so the AI can react 239 238 .

Dynamic Tool Updates: If tools can appear or disappear (for example, maybe the server loads plugins), and `listChanged` was true, the server notifies with `notifications/tools/list_changed` 240 241 , prompting the client to refresh the list.

Tool Name Conflicts: If multiple servers have tools with the same name (e.g., two servers both have a `search` tool), the client or host might need to disambiguate. The docs suggest strategies like namespacing by server (like `server1___toolname`) or prefixing with a server ID [242] [243]. In Claude's implementation, they do exactly this: each tool is referenced as `mcp__<serverName>__<toolName>` in the allowedTools list [72] [71]. So if the user's config names the server `"mcp-desktop"`, a tool `read_file` becomes `mcp__mcp-desktop__read_file` when allowing it. This prevents collisions. The spec notes that the server's own name (provided in initialize) isn't guaranteed unique (e.g., two instances of the same server code might both call themselves "filesystem") [244], hence the client might use other means (like an internal ID or user-provided alias) to namespace.

Annotations & Client Behavior: Those hints we discussed can influence client UI. For example, Claude could show a warning icon for destructive tools, or group tools by openWorld vs local. A key point is that *annotations are not security controls* – clients should never auto-run a tool solely because it's marked readOnly, for instance [245]. In any case, the presence of these hints in the knowledge base allows a GPT assistant to properly populate them when generating a tool definition.

Security Considerations for Tools: Tools are the riskiest feature, so the spec emphasizes: - **Input validation:** Servers must validate all incoming tool parameters against the schema and sanitize them (e.g., avoid command injection if a tool executes shell commands) [246] [247]. The schema helps by defining expected types, but further checks (like path sanitization, allowed ranges, etc.) are needed. - **Access control:** If a tool accesses protected resources, enforce auth. Also audit usage, rate-limit if needed (so a runaway model doesn't spam a tool) [248] [249]. - **Error handling:** As mentioned, don't leak internal stack traces; handle timeouts and partial failures gracefully [250] [251]. - **Side-effects:** Because tools can change things, the user should always be aware. The host ensures user approval, but the server should also implement safeguards (like maybe not performing a `delete_file` if certain conditions aren't met, or always making a backup – indeed the user's MCPDesktop README indicates it creates backups for destructive ops [252] [253], which aligns with these principles).

From the perspective of Claude generating code, it should be sure to incorporate input validation and safe defaults when writing tool implementations. For instance, if asked to implement a new `execute_shell` tool, it should add checks on the command input (perhaps restrict allowed commands or sanitize dangerous characters) and mark it with `openWorldHint: true, destructiveHint: true` appropriately. It should also ensure to return errors with `isError` in case something goes wrong, so that the AI using it knows the tool failed.

Sampling (Client Feature)

What is Sampling? *Sampling* is an MCP feature that essentially allows a server to ask the client to **invoke the LLM** mid-interaction [254] [255]. In other words, the server can say: "I have this subtask that requires an LLM completion – dear client, please get the model to generate text for this prompt and give it back to me." This enables building more **agentic workflows** where the server might loop: do some work, get an LLM's help, then continue, etc., all under user oversight [256] [257].

For example, imagine a server tool that writes code and then wants the LLM to review that code. The server could use sampling to feed the code to Claude for analysis, then use Claude's answer to decide next steps (before returning final result to the user). Another scenario: a server might handle a multi-turn dialog with an external system by having the LLM draft messages.

Important: at time of writing, Anthropic noted that *Claude Desktop does not yet support sampling* (it's a planned feature) [258]. But the protocol is defined.

Capability: A client that supports sampling declares `"sampling": {}` in its capabilities [30]. If not declared, servers shouldn't attempt it. The user's Claude client likely doesn't support it yet as per documentation, but we include it for completeness and future-proofing.

How Sampling Works: The server sends a request `sampling/createMessage` with a **prompt (message list)** and some parameters, the client (Claude) will review/modify it, have the model generate a completion, and then return the result in the response [259] [260]. The **human-in-the-loop design** means the client should ask the user to approve both the prompt being sent and the completion returned (if the client UI is interactive) [261] [262].

The request payload (params) includes: - `messages`: an array of message objects forming the conversation to send to the LLM [263] [264]. Each message has a `role` (either `"user"` or `"assistant"` – typically these alternate) and a `content` which can be of type text or image (you could even have an image in the prompt) [265] [266]. For example, the server might send a single user message like *"What is the capital of France?"* as in the docs [267] [268]. - `modelPreferences`: an object where the server can express preferences for what kind of model to use [269] [270]. This has two parts: - **Capability priorities** (`costPriority`, `speedPriority`, `intelligencePriority`) each 0.0–1.0 indicating how important cheapness, speed, or intelligence are for this request [271]. This is a neat abstraction because the server doesn't know what models the client has; it just says what it cares about (e.g., "I need a very smart model even if it's slow/costly" vs "just do it quickly, I don't need the best quality"). - **Model hints** (`hints` list): optional suggestions of model family or name to use [272] [273]. For instance, `[{ "name": "claude-3" }]` to suggest using Claude 3 if available. Hints are treated loosely (substring matches, order of preference) [273] [274]. The client ultimately chooses a model it has access to, possibly mapping hints to an equivalent (like mapping "GPT-4" to its Azure endpoint, etc.) [275] [276]. The server can list multiple hints in order (the first is most preferred) [277]. - `systemPrompt`: an optional system instruction string to use for this completion [278]. The client may choose to prepend or override its own system prompt with this. It's advisory; clients can modify or ignore it [279] [280]. - `includeContext`: a flag controlling how much of the *current chat context* to include alongside the provided messages [281]. Values can be `"none"`, `"thisServer"`, or `"allServers"` [282]. For example, if the server wants the LLM to only see the prompt it provided and nothing else, it uses none. If it wants the LLM to also consider all resources/tools used so far in this conversation with all servers, it can request all. The client ultimately decides (and likely needs user approval to share context from other servers). - **Standard generation parameters**: `temperature` (randomness), `maxTokens` (length), `stopSequences`, and a `metadata` field for any provider-specific options [283] [284]. This is similar to typical OpenAI/Anthropic API parameters.

An example sampling request from the docs:

```
{
  "method": "sampling/createMessage",
  "params": {
    "messages": [
      { "role": "user", "content": { "type": "text", "text": "What files are in
the current directory?" } }
```

```
    ],
    "systemPrompt": "You are a helpful file system assistant.",
    "includeContext": "thisServer",
    "maxTokens": 100
  }
}
```

[285] [286] This asks the client to generate an assistant answer to the user's question, including relevant context from "thisServer" (meaning if the server had any resources loaded in context, include those, but not other servers' context).

Response: The client's response to `sampling/createMessage` will contain: - `model`: the model name that was used (so the server knows) [287] [288]. - `stopReason`: why the generation stopped (end of turn, stop sequence hit, max tokens, etc.) [288]. - `role`: usually `"assistant"` (the role of the content returned) [287]. - `content`: the generated content (same structure as any message content: type, text or image data, etc.) [287].

For example:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "role": "assistant",
    "content": {
      "type": "text",
      "text": "The capital of France is Paris."
    },
    "model": "claude-3-sonnet-20240307",
    "stopReason": "endTurn"
  }
}
```

[289] [290] This indicates Claude (model `claude-3-sonnet-20240307`) answered the question.

The server receives this and can use the content as needed – maybe send it back to the user as part of a larger result or analyze it further. The server does **not** automatically forward it to the user; it's up to server logic.

Why Sampling is Useful: It lets servers implement more complex logic that might require AI reasoning. For instance, a server could break a task into steps and at one step ask the model to produce something (like write a draft), then do something with that draft (like run tests on it), then maybe ask the model to fix errors, etc. *All that without user intervention*, but with the user's initial permission for agent to use sampling and tools. Essentially, it's building AI agents on top of Claude in a standardized, safe way.

Human Oversight: The spec emphasizes that both the *prompt going into the model and the result coming out should be user-visible and confirmable* [261] [291]. Clients should show the prompt to the user ("Server X wants to ask the AI: ...") and the user can edit or approve it [280]. Similarly, the answer can be shown ("AI responded with: ... approve sending to server?"). This ensures the server doesn't sneak in unseen queries or get sensitive info from the model that the user didn't consent to.

Security & Privacy: When implementing sampling: - Validate and sanitize everything. The server should not send user private data in the prompt unless needed, and the client will also check for sensitive content. - The client should enforce rate limits (so a server can't spam expensive completions) [292] [293]. - The data is kept local (server doesn't get the whole user conversation unless allowed via includeContext). - The user's model usage costs could be impacted, so monitoring and possibly cost control is important (the spec even says "monitor sampling costs") [294] [295].

From a development standpoint, if an assistant (Claude) is generating instructions to use sampling, it should ensure to: 1. Only do so if the client declared support. 2. Form the `sampling/createMessage` request properly with a clear prompt and appropriate preferences/hints. 3. Expect and handle the result.

Given sampling isn't widely implemented yet, an assistant might more often have to *explain it* or set up the structure in code rather than actively use it at runtime in today's environment.

Best Practices & Patterns: The spec suggests when using sampling: - Keep prompts well-structured and as minimal as possible (only what's needed for that subtask) [296]. - Use `includeContext` to include relevant context if needed (but not more than necessary, to respect boundaries) [297]. - Always check the response content before using it (e.g., if the model's answer is unexpected, the server could try again or handle error). - Possibly, incorporate a **"self-critique"** pattern: e.g., server gets an answer and might ask the model to verify it via another sampling call. This is advanced but plausible in agent design.

Example Scenario: Consider building an "AI debugger" MCP server. The user says "Debug my code." The server might: - Use a Tool to run the code and capture an error, - Then use **sampling** to ask Claude "Given this error and code, what is the likely cause?", - Then maybe use another Tool to run a fix or gather more info, - Then perhaps use sampling again to have Claude suggest a fix, - Finally return a result to the user with the diagnosis and fix.

This interplay shows how sampling and tools together allow loops of AI reasoning and real actions – all orchestrated by the server, under user oversight.

Roots (Client Feature)

What are Roots? *Roots* are essentially **workspace or scope identifiers** that the client communicates to the server to tell it *where it should focus or limit itself* [298] [84]. In practice, for a filesystem-related server, a "root" would be a directory path that the server should consider as the project workspace. For a cloud service, a root might be a specific resource scope or endpoint.

The idea is to establish **boundaries**. By telling a server "here are the roots: e.g. `file:///home/user/projects/myapp`", the server knows it should operate mainly in that directory and not elsewhere unless instructed [299] [300]. It's also useful when multiple roots are needed (multi-repo project, multiple data sources).

Capability: A client that supports roots declares `"roots": { "listChanged": true/false }` in capabilities [301] [302]. If `listChanged:true`, the client will send notifications if roots change (like user switched projects).

Claude Desktop does support roots – when you open a folder in Claude, it likely sets that as a root for relevant servers (like the `filesystem` server). The user's MCPDesktop README doesn't explicitly mention roots, but presumably if Claude passes a path via env or args, it could be integrated. It might be an improvement area if not done yet.

Listing Roots: The server can request the current roots by calling `roots/list` [303]. The client responds with an array of root objects, each with: - `uri`: the root URI (almost always a `file:///` path in current spec) [304]. - `name`: optional display name for that root [304].

Example:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "roots": [
      {
        "uri": "file:///home/user/projects/myproject",
        "name": "My Project"
      }
    ]
  }
}
```

[305] [306] If multiple roots:

```
"roots": [
  { "uri": "file:///home/user/repos/frontend", "name": "Frontend Repository" },
  { "uri": "file:///home/user/repos/backend",  "name": "Backend Repository" }
]
```

[307] [308].

The server can use these URIs when serving resources or performing operations. For example, an MCPDesktop server might restrict file operations to under the given root path(s) for safety – the spec *strongly encourages servers to respect root boundaries* [309] [310].

Root Change Notification: If the user changes what they're working on (e.g., opens a different folder or adds another root), the client will send `notifications/roots/list_changed` to the server [311]. Then

the server can call `roots/list` again to get the new set. This enables dynamic updates, like if a user in Claude Desktop switches the active project directory, the server can adapt.

Use Cases: Common use cases for roots: - Define a project directory for coding assistants (so they know where the source files are) [312] . - Limit a search tool to certain domains or endpoints. - Provide context like "only this database and no other". It's a form of sandboxing and context hint combined.

Security: The client must be careful only to expose directories the user permitted. It must validate that root URIs are safe (no weird `file:///../../etc` unless user really did that) [313] . Also, the server should not go outside these roots in its operations (e.g., a file server should prevent reading files above the root) [310] [314] . The spec instructs servers to *enforce* root boundaries in their own logic, and clients to only offer valid roots [313] [314] .

Implementation: For the client (Claude), implementing roots means: - On initialize, send the initial roots (if any) in the `initialize.params` under capabilities or a dedicated field. Actually, reading the spec, it looks like roots are not sent in the `initialize` request, but rather the server must actively fetch them via `roots/list` . (The spec doesn't show an `initialize` containing roots, just capabilities). - Provide a `roots/list` handler that returns the current roots (likely maintained by the host app). - When user changes, send `roots/list_changed` .

For the server, using roots is straightforward: call `roots/list` at startup to know what to work on. Possibly subscribe to changes (there's no explicit subscribe for roots, just the notification from client side).

Example: The MCP Filesystem server (hypothetical) on start might do:

```
let currentRoots = [];
client.on('ready', async () => {
  if (clientCapabilities.roots) {
    const rootsResult = await client.request("roots/list");
    currentRoots = rootsResult.roots;
    console.log("Roots set to:", currentRoots);
  }
});
client.onNotification("roots/list_changed", async () => {
  const rootsResult = await client.request("roots/list");
  currentRoots = rootsResult.roots;
  console.log("Roots updated:", currentRoots);
});
```

Then when handling a `resources/list` or file operation, it would ensure it only deals with files under `currentRoots` .

User Interaction: Typically, the user might choose the root in the UI (open folder dialog). In that sense, roots are *user-driven context sharing*. The spec suggests using clear UI for managing roots (like showing which directory is shared, letting user remove it) [315] [316] .

Error Cases: If a server calls `roots/list` but the client doesn't support roots, the client should return a JSON-RPC error `-32601 Method not found` (which it would anyway if not implemented) [317] [318]. Or if something internal goes wrong, `-32603`. The server should handle that gracefully (and assume maybe no roots concept is used).

For Claude, if it were to generate code or instructions around roots, it should: - Ensure any server code checks that operations are within a root. - Possibly prompt user to set a root if one is needed.

Elicitation (Client Feature)

What is Elicitation? *Elicitation* is an interactive feature where the server can ask the client to **obtain additional information from the user** in the middle of a workflow [86] [319]. This is essentially a standardized way for a server to pop up a question to the user and get a structured answer. It's useful when the server realizes it needs something it doesn't have – for example, an API key, or a configuration choice, or consent for something. Instead of failing or guessing, it can formally request it through elicitation.

This feature is **new and evolving** (introduced in revision 2025-06-18) [320] [321]. It draws from similar ideas in interactive CLI tools or setup wizards.

Capability: A client that supports it declares `"elicitation": {}` [322]. If not declared, server shouldn't use it.

Flow: The server sends `elicitation/create` request with: - A user-facing `message` (string) explaining what info is needed [323] [324]. - A `requestedSchema` which is a JSON Schema (very restricted subset) describing the data it expects back [323] [325].

This is similar to how you'd define a form: e.g. "Please provide your GitHub username" and schema says it's an object with a `username` string. The client will show this to the user likely in a dialog form.

Example request:

```
{
  "method": "elicitation/create",
  "params": {
    "message": "Please provide your GitHub username",
    "requestedSchema": {
      "type": "object",
      "properties": {
        "username": {
          "type": "string",
          "title": "GitHub Username",
          "description": "Your GitHub username (e.g., octocat)"
        }
      },
      "required": ["username"]
    }
```

```
    }
  }
```

[326] [327] The client then would prompt the user: perhaps showing “Server X asks: Please provide your GitHub username” with a text box (title and description help the UI). The user can either **accept** (submit the info), **decline**, or **cancel**.

The client validates the input against the schema (e.g., ensure it’s a string and not empty if required). Then returns a response: - If user provided info, `result: { action: "accept", content: { username: "thevalue" } }` [328] [329]. - If user hit decline, `result: { action: "decline" }` (no content) [330] [331]. - If canceled (maybe closed dialog), `result: { action: "cancel" }` [332] [333].

The server should handle each case appropriately: - Accept: use the data and proceed. - Decline: treat it as a no – maybe abort the operation or use a default or alternative. - Cancel: perhaps similar to decline or ask later. The spec suggests maybe retry later if canceled [334] [335].

Schema Constraints: The JSON schema allowed is intentionally limited to simple **flat structures** of *primitive types* [325] [336]: - The top-level must be an object with properties (each property is one input field). - Allowed property types: string, number (or integer), boolean, or a string with an enum (which effectively is a dropdown) [337] [338] [339] [340]. - You can specify `title`, `description` on each field for nicer UI, plus things like `minLength`, `maxLength` for strings, `minimum` / `maximum` for numbers, `format` for certain string patterns (email, uri, date, date-time) [341] [342] [343]. - `enum` with an array of options and `enumNames` for their labels is supported for strings [344] [345] [346]. - Complex nested objects or arrays are **not supported** – this keeps the UI form simple and uniform [347].

This covers most typical inputs (text, yes/no, choose from list, number input).

Response Actions: We detailed them above, but to recap as per spec: - **accept** – user gave info (with content). - **decline** – user refused (explicit no). - **cancel** – user aborted without a clear yes/no (like closing window) [331] [333].

The server might treat decline vs cancel similarly (both mean no info), but conceptually decline is a conscious refusal, cancel is more like “not now”. The difference could be used if a server wants to differentiate between user saying “no, I won’t give that” vs “I didn’t get an answer now”.

User Interaction & Safety: Elicitation ensures the user always knows which server is asking (the client UI should display the server name or something) [348] [349]. It’s meant for non-sensitive info – **the spec explicitly says servers must NOT use elicitation for passwords, tokens, etc.** [350] [351]. Those should be handled by proper auth flows outside MCP. Elicitation is for things like “What should I name this new project?” or “Do you want to enable feature X? (yes/no)”.

Example Use Case: Suppose a server is going to create a new project directory but needs a project name from user. It can call elicitation to ask for “Project Name” (string, min length 3), and maybe a “Framework” (enum of [React, Vue, Angular, None]) and “Use TypeScript” (boolean default true) – essentially a mini-form [352] [353] [354]. The user fills it, and the server uses those values to scaffold the project accordingly. The example in spec matches this scenario [355] [356].

Integration in Claude: Once clients support this, if a server triggers elicitation, Claude Desktop might show a modal with the form. Only after user submits does the server get the data and continue. If the user declines, perhaps Claude will tell the assistant "user declined to provide X".

From Claude's perspective (the LLM) – It might not directly engage with elicitation aside from perhaps seeing when a server it's talking to goes silent waiting for user input. But for code generation, if asked to create a server that needs user input, Claude should use elicitation rather than just pausing or hacking a custom prompt. E.g., generating code: "If config not found, use elicitation to ask user for config".

Security Considerations: - The client should implement rate limiting (so a malicious server cannot bombard the user with endless questions) [357]. - Always clearly identify the server making the request (so user doesn't confuse it with Claude or another source) [358] [351]. - Validate user responses against the schema on the client side too, not just trust them (though since the client is collecting it, validation is natural) [359] [360]. - The server should have fallback plans if user declines (don't crash; maybe proceed with defaults or cancel the operation gracefully) [361] [334].

All of these features (Sampling, Roots, Elicitation) highlight how the **MCP client (Claude)** and **MCP servers** collaborate beyond simple question-answer: they form a loop with the user in control at high-level.

Discussion

Through analyzing these components, several **thematic findings** and **comparisons** emerge:

- **Bidirectional Control:** MCP enables a conversation not just between user and AI, but among *user, AI, and external tools/data* with carefully defined roles. **Server features (Resources/Prompts/Tools)** allow the server to inject information or actions *into* the AI's context, while **client features (Sampling/Roots/Elicitation)** allow the server to request actions *from* the AI or user via the client. This two-way extension is what makes MCP powerful. We see a clear symmetry: **Tools** let the AI act on the outside world via server, while **Sampling** lets the server act on the AI (via the client) to get more AI output [362] [83]. Likewise, **Prompts** (server-defined for user to invoke) mirror **Elicitation** (server requests user input) in some way – one is initiated by user to guide AI, the other by server to query user. This bidirectional design is unique compared to older "unidirectional" plugin systems (like OpenAI plugins didn't have a way for plugin to ask user questions, for example).

- **User Consent & Isolation as a First-Class Concern:** A consistent theme is that **nothing happens without the user's knowledge or approval** (unless they've pre-configured it). Resources are inert until user selects them [363]. Prompts require user trigger. Tools won't run without user permission (Claude's `--allowedTools` mechanism) [71] [72]. Sampling requests and results are shown to the user for confirmation [261] [291]. Elicitation literally asks the user. Additionally, each server runs in isolation – it only knows what the host chooses to share (like context relevant to that server, or roots limiting file access) [25] [364]. This is great for security but also means from a developer view, *you must plan for user declines and partial info*. Our research highlights that robust MCP implementations need to handle "no" gracefully. For instance, if a user declines to give an API key via elicitation, the server might disable certain tools and inform the user, rather than crash or proceed blindly.

- **Extensibility and Versioning:** MCP's design is clearly meant to evolve. The **versioning scheme by date** and the presence of a detailed changelog [365] show that new features get added (like elicitation) and deprecated ones get phased out (like SSE). The protocol's negotiation mechanism means new features can be introduced without breaking older clients/servers – they'll just not advertise unsupported stuff. From our review, it's apparent that the maintainers are carefully adding functionality:

- e.g., Structured outputs for tools were added to better support programmatic use of tool results [4] [194].
- The `title` field addition indicates feedback to separate UI labels from identifiers [366].
- OAuth integration steps in key changes show security is being continuously improved (requiring resource indicators, etc.) [367] [368].

We should highlight that for long-term maintenance, one should keep an eye on spec updates. The knowledge base should allow easy updating of particular sections if, say, the next version introduces "streaming prompts" or something.

- **Under-documented Areas:** The official docs are comprehensive on core features, but some advanced aspects are less fleshed out in examples. For instance:
- **Progress tracking & cancellation** (listed as utilities) were not deeply covered in our sources. They likely refer to a standard way for servers to report progress (like a progress bar) and for either party to cancel a long-running request. The spec mentions *progress tokens* for long ops in best practices [369] [370], and likely defines some `_meta.progress` or similar. This might be an area to explore in the schema reference. Since the user's focus was core components, we didn't dive deep, but it's worth noting as a gap.
- **Logging/Auditing hooks** – implied but not described how a host might log all tool calls or share logs. Possibly left to implementors (Claude likely logs actions separately).
- **Authorization flows** – The key changes hint at OAuth flows for MCP servers needing auth tokens [371]. The details would be in the Authorization section of base protocol (which we didn't retrieve fully due to time). So if a user is building a server that needs user tokens, they should consult that part of spec (e.g., using `_meta` fields for auth info or resource server metadata).

However, these don't critically impede basic implementation – they are more about integration in enterprise or cloud scenarios.

- **MCPDesktop vs Official Spec:** Comparing the user's *MCPDesktop repository* configuration and tool set to the official spec, it appears largely aligned:
- The config uses `mcpServers` JSON which is exactly how Claude CLI expects it [11]. The user's example adds environment variables for server which is fine (the spec doesn't cover env specifics, but it's implementation detail).
- The *tools* listed in MCPDesktop README (like `read_file`, `write_file`, etc.) correspond to what we'd expect – the user likely has each implemented. To ensure full compliance, the user should confirm their server's JSON outputs follow the spec (e.g., wrapping file content in the `content` array with `text` fields, using `isError` on errors, etc.). From README, the example tool responses do show using that format [372].
- One discrepancy: The README's JSON example for `schedule_maintenance` tool output doesn't explicitly show `isError` or similar, but presumably errors are handled in their code. They should

double-check error handling aligns with spec suggestions (return error content rather than throwing, unless it's a truly exceptional case).
- Their *Resource Manager* and *Security Validator* modules mentioned in README [373] align with best practices we saw (like path sanitization, resource limits).

So, the MCPDesktop server seems to be a well-thought implementation that can be mapped onto the MCP spec features easily. The knowledge base we compiled can help the user verify each part (e.g., ensure their `manifest.json` fields correspond, ensure they declare capabilities properly using the SDK, etc.).

- **Claude Integration Considerations:** When designing prompts for Claude to automatically utilize this knowledge base, we want to ensure:
- The instructions to Claude (the model) about how to format a new tool or prompt or resource are precise. With this knowledge base, Claude can generate, say, a new Tool stub: it should include `inputSchema` definitions and maybe add a suitable annotation for safety.
- **Edge Cases**: Claude should know what to do if, for example, a user asks it to use a tool that doesn't exist – the correct approach is to respond with a JSON-RPC error with code -32601 (method not found) or a tool-specific error code if appropriate [47] [317]. The knowledge base covers those standard codes. We should ensure these are surfaced (we did mention a few like resource not found, roots not supported).

- The knowledge base provides JSON examples with citations. A GPT model could potentially extract patterns from those to ensure it generates similar JSON. For instance, how it sees `tools/list` output and copy that style to new contexts.

- **Potential for Automation:** With this structured info, one could imagine building a JSON-based spec representation for each component (like a machine-readable schema from the TypeScript definitions) which a GPT could ingest. In fact, the docs reference an official JSON Schema and TS schema for the whole spec [374]. We did not explicitly parse those due to time, but that is another asset (e.g., they mention a `schema.ts` on GitHub [375] and a generated JSON Schema file). For programmatic generation, those could be gold. However, our focus was the human-readable doc plus code patterns, which is arguably more useful for prompt design and immediate understanding.

- **Deprecations and Future Stability:** We identified SSE transport deprecation as something to avoid implementing new. Also JSON-RPC batching removed, meaning any older idea of sending batch requests is out (Claude's UI wouldn't have done that anyway) [376]. The addition of features like elicitation might still be experimental, but since we frame it as "newly introduced and may evolve" [320], a GPT or developer will treat it with caution (maybe feature-flag it, e.g., only use if both sides definitely support spec 2025-06-18 or later).

In conclusion, the MCP specification covers a wide range of capabilities that, together, enable **bidirectional tool integration** for AI assistants. By dissecting each piece, we see how they complement each other: Resources and Tools to bring external data/actions in, Prompts to guide model behavior, and Sampling/Elicitation to let the server intelligently leverage the model and user in the loop. Each component has been designed with a balance of flexibility and safety (structured schemas, required declarations, user consent everywhere).

For the user's case of *Claude Code integration*, this knowledge base means Claude can be instructed to: - Automatically generate new **Tools** for the MCPDesktop server with correct JSON schemas and annotated metadata, - Or produce new **Prompts** that the server can serve to users, - Or even suggest usage of **Resources** and **Sampling** in an orchestrated way to enhance functionality (e.g., using sampling instead of naive approaches when needing model help inside tool execution).

Gaps and Recommendations:

While MCP is robust, developers should be aware of: - Proper **error handling strategy**: prefer in-protocol errors only for truly fatal conditions (method not found, etc.), and use in-content errors for tool failures [239] [238] so the AI can respond. - **Testing approach**: The spec suggests thorough testing of each feature – e.g., test tools with good and bad inputs, ensure resources handle large files or binary properly, test prompt workflows end-to-end [377] [378]. Incorporating those into development will save headache. - **Documentation**: If building servers for others, follow the spec's lead in documenting each tool's usage, each prompt's purpose, etc. This will also help when these are surfaced to users in UI (the descriptions matter).

By adhering to the spec guidelines and using this knowledge base, one can confidently implement or enhance MCP servers and clients. The structured breakdown here should also aid a GPT-based system (like Claude) in reasoning about MCP tasks – for example, if asked *"How do I add a new file resource to my MCP server?"* the assistant can recall this knowledge base: *you'd implement* `resources/list` *to include it, handle* `resources/read` *accordingly, etc., citing the spec where needed.*

Overall, MCP provides a blueprint for modular AI extensibility, and our comprehensive analysis ensures that both humans and AI (Claude) have the low-level technical details to build on this protocol effectively.

Conclusion

In this report, we assembled a detailed technical knowledge base for Anthropic's **Model Context Protocol (MCP)** – covering its architecture, components, message patterns, and best practices – drawn entirely from official documentation. Key takeaways include:

- **MCP's Architecture:** A client-server model enabling LLM applications (like Claude) to interface with external tools and data in a standardized way. Communication is built on JSON-RPC with a flexible transport layer [22] [32]. A 3-phase handshake (initialize, capabilities exchange, initialized) bootstraps each session [13], ensuring both sides agree on supported features and version [379] [62]. Security and user consent are woven throughout the design (no tool use or data access occurs without explicit approval) [66] [70].

- **Server Feature Primitives:** MCP servers can provide **Resources** (exposed data like files or DB entries) [91] [92], **Prompts** (reusable LLM prompt templates/workflows) [144] [145], and **Tools** (executable functions/actions the LLM can call) [177] [380]. Each is structured with a clear schema:

- *Resources* have URI identifiers and metadata, and support listing (`resources/list`), reading (`resources/read`), and optional change notifications [96] [42]. They allow servers to share context while letting the user control what gets used [67].

- *Prompts* are defined by name, description, and arguments, enabling clients to fetch prompt content (`prompts/get`) for user-initiated actions [146] [155]. They can include dynamic content (like embedded resources) [161] [381] or multi-turn sequences [164] [165]. This feature standardizes "canned" interactions, improving usability.

- *Tools* are specified with names, descriptions, **JSON schemas for inputs (and outputs)**, and optional behavioral annotations [181] [191]. Clients discover them via `tools/list` and invoke via `tools/call` with arguments [198] [201]. Tool results are returned as content blocks (text, image, etc.) and may include a structured payload for programmatic use [204] [209]. The protocol encourages reporting tool execution errors within the result (`isError: true`) so the LLM can handle them [239] [238] instead of simply failing the RPC. Tools unlock powerful extensions (code execution, web queries, etc.), but **user approval and rigorous input validation are required** for safety [246] [382].

- **Client Feature Extensions:** MCP clients (like Claude) can augment the interaction with **Sampling** (server-initiated LLM calls) [254] [383], **Roots** (defining workspace scope) [84] [384], and **Elicitation** (server prompting the user for info) [86] [385]:

- *Sampling* allows a server to ask the client's AI to **generate a completion** for a given prompt/messages, with the client mediating model selection and ensuring user oversight [255] [383]. The server sends a `sampling/createMessage` request containing conversation context (roles + messages) and preference hints (e.g., prioritize speed vs accuracy, suggest model family) [386] [387]. The client returns an assistant reply (`content`) along with metadata like `model` used [289] [290]. This feature essentially lets servers implement multi-step reasoning or sub-tasks using the AI itself, while the user maintains control (the user can review prompts/results as needed) [261] [291]. It's not yet supported in Claude Desktop (as of mid-2025) [258] but is defined for future use.
- *Roots* provide a mechanism for the client to inform the server about relevant **root URIs (e.g., directories)** that bound the server's operations [299] [388]. The server can retrieve them via `roots/list` [303]. This is crucial for context (like telling a code-search server which project folder to index) and for security (limiting file access to allowed areas) [309] [389]. Roots are essentially the *user's way to sandbox and contextualize* MCP servers.

- *Elicitation* is a newly introduced feature enabling servers to ask users for structured input during a session (via `elicitation/create`) [323] [336]. The server provides a prompt message and a JSON Schema for the expected data (e.g., ask for `"username"` as a string) [390] [326]. The client then presents a form to the user. The user can **accept (submit data), decline, or cancel**, which the client reports back with an appropriate action code and content [330] [331]. This feature allows dynamic, on-the-fly user engagement (for example, asking "Which repository should I use?" if not specified). It's designed with privacy in mind – not to request passwords or highly sensitive info [350] – and ensures the user knows which server is asking and can refuse [348] [351]. Elicitation fills a gap where a server would otherwise be stuck or have to guess user intent; now it can formally ask through the UI.

- **Best Practices & Stability:** We distilled numerous best practices: e.g., always declare capabilities and **honor feature negotiation** (don't use what the other side hasn't advertised) [62] [63]; use clear naming and documentation for prompts/tools so both the UI and LLM can understand their purpose [171] [213]; thoroughly validate inputs to tools and any user-supplied data (following a defense-in-depth approach) [247] [250]; and implement security measures like path sanitization in resource access and rate limiting for expensive operations [138] [382]. We highlighted changes in the recent MCP spec

revision that developers should note – e.g., the new `title` fields for display names [5], the deprecation of SSE-only transport [3], and introduction of structured outputs for tools [4] and elicitation [391]. These indicate the protocol's direction: more *structure*, more *safety*, and more *interactive capability*.

- **Comparison to User's Context:** We compared the MCP spec against the user's current MCPDesktop server setup. The user's server provides many tools (file operations, disk management, etc.) and presumably some resources (like system info) – these align well with MCP's model. By following the spec's guidelines (ensuring the server returns responses in the exact JSON format MCP expects [372] [392], and declaring all capabilities in initialization), the user's implementation can be made fully MCP-compliant and benefit from Claude's built-in handling. Additionally, features like elicitation could be leveraged to improve user interaction (e.g., asking for confirmation or parameters in complex operations via UI instead of through prompt text). This knowledge base will help identify any gaps; for example, if MCPDesktop doesn't yet send a `list_changed` notification when new maintenance tasks are added, that could be an enhancement to consider so that Claude's UI could update accordingly.

In summary, the **Modular Context Protocol** provides a rich framework to extend Claude (or any LLM) with custom functionality and data, in a controlled and standardized way. Our exhaustive breakdown ensures that **Claude (via Claude Desktop or Claude Code)** can both **generate instructions/code** for MCP implementations *and* **reason about using MCP** at runtime with full technical context. By adhering to the patterns and specs cited here, developers and the AI alike can reliably build out new integrations – from a desktop automation server to a cloud API bridge – confident they are following the official protocol.

With this knowledge base, Claude can act as an even more effective coding assistant for MCP-related development, and can safely navigate the complexity of tool-use, data retrieval, and user prompting that MCP makes possible.

Next Steps: Developers should use this document in tandem with the MCP **Schema Reference** [393] [394] (for exact field definitions) and keep an eye on the official **MCP repository/discussions** for updates [395]. Testing new MCP servers with the **MCP Inspector tool** (as listed in the docs) [396] [397] can be a great way to verify compliance. As the protocol evolves (e.g., future versions might add new features or refine elicitation), this knowledge base should be updated to reflect those changes, ensuring Claude's guidance remains up-to-date.

By leveraging MCP to its fullest, we can greatly **enhance Claude's capabilities** in a modular, secure fashion – effectively plugging Claude into the wider digital world while maintaining the trustworthy guardrails set by Anthropic's design.

---

[1] [7] [322] [330] [331] [332] [333] [338] [339] [340] [342] [343] [345] [346] [347] [348] [349] [350] [351] [360] [385] Elicitation - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/client/elicitation

[2] [4] [5] [26] [194] [365] [366] [367] [368] [371] [376] [391] Key Changes - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/changelog

3 32 33 49 54 55 Transports - Model Context Protocol
https://modelcontextprotocol.io/docs/concepts/transports

6 16 393 394 395 396 397 Introduction - Model Context Protocol
https://modelcontextprotocol.io/introduction

8 15 64 65 66 68 69 70 73 74 82 83 362 375 Specification - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18

9 34 35 36 37 38 39 40 41 44 374 Overview - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/basic

10 51 252 253 372 373 392 README.md
https://github.com/UsernameTron/MCPDesktop/blob/2e8a145d0e4cd30ac1c85fed2db169a399a22e20/README.md

11 12 71 72 Claude Code SDK - Anthropic
https://docs.anthropic.com/en/docs/claude-code/sdk

13 22 23 45 46 50 52 53 56 369 370 Core architecture - Model Context Protocol
https://modelcontextprotocol.io/docs/concepts/architecture

14 77 78 79 80 81 88 Overview - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/server

17 Model Context Protocol (MCP) - Anthropic
https://docs.anthropic.com/en/docs/mcp

18 19 20 21 24 25 57 58 62 63 364 379 Architecture - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/architecture

27 59 102 103 152 153 154 Prompts - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/server/prompts

28 29 43 47 48 60 99 100 101 104 105 106 107 108 109 110 111 115 117 118 121 124 125 126 128 130
Resources - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/server/resources

30 31 75 76 267 268 270 271 273 274 275 276 277 289 290 383 387 Sampling - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/client/sampling

42 67 89 91 92 93 94 95 96 97 98 112 113 114 116 119 120 122 123 127 129 131 132 133 134 135 136 137
138 363 Resources - Model Context Protocol
https://modelcontextprotocol.io/docs/concepts/resources

61 301 302 303 304 305 306 307 308 310 311 313 314 315 316 317 318 384 389 Roots - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/client/roots

84 85 298 299 300 309 312 388 Roots - Model Context Protocol
https://modelcontextprotocol.io/docs/concepts/roots

86 87 319 320 321 323 324 325 326 327 328 329 334 335 336 337 341 344 352 353 354 355 356 357 358 359 361 390
Elicitation - Model Context Protocol
https://modelcontextprotocol.io/docs/concepts/elicitation

90 177 178 179 180 181 182 185 186 187 188 191 207 211 213 225 226 227 228 229 230 231 232 233 234 235 236 237
238 239 240 241 242 243 244 245 246 247 248 249 250 251 377 378 380 382 Tools - Model Context Protocol
https://modelcontextprotocol.io/docs/concepts/tools

139 140 141 142 143 144 145 146 147 148 149 150 151 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 381 Prompts - Model Context Protocol

https://modelcontextprotocol.io/docs/concepts/prompts

183 184 189 190 192 193 195 196 197 198 199 200 201 202 203 204 205 206 208 209 210 212 214 215 216 217 218 219 220 221 222 223 224 Tools - Model Context Protocol

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

254 255 256 257 258 259 260 261 262 263 264 265 266 269 272 278 279 280 281 282 283 284 285 286 287 288 291 292 293 294 295 296 297 386 Sampling - Model Context Protocol

https://modelcontextprotocol.io/docs/concepts/sampling